Oracle® Database

SQL Language Reference

12*c* Release 2 (12.2)

E83703-08

January 2023

Oracle Database SQL Language Reference, 12*c* Release 2 (12.2)

E83703-08

Primary Authors: Usha Krishnamurthy, Mary Beth Roeser

Contributors: Drew Adams, David Alpern, Angela Amor, Geeta Arora, Vikas Arora, Lance Ashdown, Hermann Baer, Prasad Bagal, Nigel Bayliss, Eric Belden, Timothy Chien, Eugene Chong, George Eadon, Amit Ganesh, Barb Glover, Naveen Gopal, Mike Hallas, Beda Hammerschmidt, Min-Hank Ho, Patricia Huey, Chandrasekharan Iyer, Peter Knaggs, Srinath Krishnaswamy, Vasudha Krishnaswamy, Andre Kruglikov, Tom Kyte, Huagang Li, Yunrui Li, Bryn Llewellyn, Rich Long, Scott Lynn, Vineet Marwah, David McDermid, Robert McGuirk, Jan Michels, Rahil Mir, Gopal Mulagund, Sujatha Muthulingam, Padmaja Potineni, Hanlin Qian, Ananth Raghavan, Vivek Raja, Ashish Ray, Jia Shi, Wayne Smith, Sachin Sonawane, Jim Stenoish, Sankar Subramanian, Sarika Surampudi, Randy Urbano, Peter Wahl, Charles Wetherell, Andy Witkowski, Sergiusz Wolicki, Tsae-feng Yu, Mohamed Zait, Fred Zemke, Wei Zhang, Weiran Zhang

Contents

Preface

Audience xxvi
Documentation Accessibility xxvi
Related Documents xxvi
Conventions xxvii

Changes in This Release for Oracle Database SQL Language Reference

Changes in Oracle Database 12c Release 2 (12.2.0.1) xxviii
Changes in Oracle Database 12c Release 1 (12.1.0.2) xxxix
Changes in Oracle Database 12c Release 1 (12.1.0.1) xli

1 Introduction to Oracle SQL

History of SQL 1-1
SQL Standards 1-1
- How SQL Works 1-2
- Common Language for All Relational Databases 1-2

Using Enterprise Manager 1-2
Lexical Conventions 1-3
Tools Support 1-3

2 Basic Elements of Oracle SQL

Data Types 2-1
- Oracle Built-in Data Types 2-6
  - Character Data Types 2-9
  - Numeric Data Types 2-14
  - LONG Data Type 2-18
  - Datetime and Interval Data Types 2-20
  - RAW and LONG RAW Data Types 2-28
  - Large Object (LOB) Data Types 2-29

Extended Data Types 2-32
Rowid Data Types 2-33
ROWID Data Type 2-33
UROWID Data Type 2-34
ANSI, DB2, and SQL/DS Data Types 2-34
User-Defined Types 2-36
Object Types 2-36
REF Data Types 2-37
Varrays 2-37
Nested Tables 2-37
Oracle-Supplied Types 2-38
Any Types 2-38
ANYTYPE 2-38
ANYDATA 2-38
ANYDATASET 2-39
XML Types 2-39
XMLType 2-39
URI Data Types 2-40
URIFactory Package 2-41
Spatial Types 2-41
SDO_GEOMETRY 2-41
SDO_TOPO_GEOMETRY 2-42
SDO_GEORASTER 2-42
Media Types 2-42
Data Type Comparison Rules 2-44
Numeric Values 2-44
Datetime Values 2-44
Binary Values 2-44
Character Values 2-45
Object Values 2-48
Varrays and Nested Tables 2-48
Data Type Precedence 2-48
Data Conversion 2-49
Implicit and Explicit Data Conversion 2-49
Implicit Data Conversion 2-49
Implicit Data Conversion Examples 2-52
Explicit Data Conversion 2-52
Security Considerations for Data Conversion 2-54
Literals 2-55
Text Literals 2-56
Numeric Literals 2-57

Integer Literals 2-57
NUMBER and Floating-Point Literals 2-58
Datetime Literals 2-60
Interval Literals 2-64
INTERVAL YEAR TO MONTH 2-64
INTERVAL DAY TO SECOND 2-66
Format Models 2-67
Number Format Models 2-68
Number Format Elements 2-69
Datetime Format Models 2-71
Datetime Format Elements 2-72
Datetime Format Elements and Globalization Support 2-77
ISO Standard Date Format Elements 2-77
The RR Datetime Format Element 2-78
Datetime Format Element Suffixes 2-79
Format Model Modifiers 2-79
Format Model Examples 2-80
String-to-Date Conversion Rules 2-82
XML Format Model 2-83
Nulls 2-84
Nulls in SQL Functions 2-85
Nulls with Comparison Conditions 2-85
Nulls in Conditions 2-85
Comments 2-86
Comments Within SQL Statements 2-86
Comments on Schema and Nonschema Objects 2-87
Hints 2-87
Alphabetical Listing of Hints 2-93
ALL_ROWS Hint 2-94
APPEND Hint 2-94
APPEND_VALUES Hint 2-95
CACHE Hint 2-95
CHANGE_DUPKEY_ERROR_INDEX Hint 2-96
CLUSTER Hint 2-96
CLUSTERING Hint 2-97
CONTAINERS Hint 2-97
CURSOR_SHARING_EXACT Hint 2-98
DISABLE_PARALLEL_DML Hint 2-98
DRIVING_SITE Hint 2-98
DYNAMIC_SAMPLING Hint 2-99
ENABLE_PARALLEL_DML Hint 2-100

FACT Hint 2-100
FIRST_ROWS Hint 2-100
FRESH_MV Hint 2-101
FULL Hint 2-101
GATHER_OPTIMIZER_STATISTICS Hint 2-102
GROUPING Hint 2-102
HASH Hint 2-103
IGNORE_ROW_ON_DUPKEY_INDEX Hint 2-103
INDEX Hint 2-104
INDEX_ASC Hint 2-104
INDEX_COMBINE Hint 2-105
INDEX_DESC Hint 2-105
INDEX_FFS Hint 2-106
INDEX_JOIN Hint 2-106
INDEX_SS Hint 2-106
INDEX_SS_ASC Hint 2-107
INDEX_SS_DESC Hint 2-107
INMEMORY Hint 2-108
INMEMORY_PRUNING Hint 2-108
LEADING Hint 2-108
MERGE Hint 2-109
MODEL_MIN_ANALYSIS Hint 2-109
MONITOR Hint 2-110
NATIVE_FULL_OUTER_JOIN Hint 2-110
NOAPPEND Hint 2-110
NOCACHE Hint 2-111
NO_CLUSTERING Hint 2-111
NO_EXPAND Hint 2-111
NO_FACT Hint 2-112
NO_GATHER_OPTIMIZER_STATISTICS Hint 2-112
NO_INDEX Hint 2-112
NO_INDEX_FFS Hint 2-113
NO_INDEX_SS Hint 2-113
NO_INMEMORY Hint 2-114
NO_INMEMORY_PRUNING Hint 2-114
NO_MERGE Hint 2-114
NO_MONITOR Hint 2-115
NO_NATIVE_FULL_OUTER_JOIN Hint 2-115
NO_PARALLEL Hint 2-115
NOPARALLEL Hint 2-116
NO_PARALLEL_INDEX Hint 2-116

NOPARALLEL_INDEX Hint 2-116
NO_PQ_CONCURRENT_UNION Hint 2-116
NO_PQ_SKEW Hint 2-117
NO_PUSH_PRED Hint 2-117
NO_PUSH_SUBQ Hint 2-117
NO_PX_JOIN_FILTER Hint 2-118
NO_QUERY_TRANSFORMATION Hint 2-118
NO_RESULT_CACHE Hint 2-118
NO_REWRITE Hint 2-118
NOREWRITE Hint 2-119
NO_STAR_TRANSFORMATION Hint 2-119
NO_STATEMENT_QUEUING Hint 2-119
NO_UNNEST Hint 2-120
NO_USE_BAND Hint 2-120
NO_USE_CUBE Hint 2-120
NO_USE_HASH Hint 2-120
NO_USE_MERGE Hint 2-121
NO_USE_NL Hint 2-121
NO_XML_QUERY_REWRITE Hint 2-121
NO_XMLINDEX_REWRITE Hint 2-122
NO_ZONEMAP Hint 2-122
OPT_PARAM Hint 2-123
ORDERED Hint 2-123
PARALLEL Hint 2-124
PARALLEL_INDEX Hint 2-126
PQ_CONCURRENT_UNION Hint 2-127
PQ_DISTRIBUTE Hint 2-127
PQ_FILTER Hint 2-130
PQ_SKEW Hint 2-130
PUSH_PRED Hint 2-130
PUSH_SUBQ Hint 2-131
PX_JOIN_FILTER Hint 2-131
QB_NAME Hint 2-131
RESULT_CACHE Hint 2-132
RETRY_ON_ROW_CHANGE Hint 2-132
REWRITE Hint 2-133
STAR_TRANSFORMATION Hint 2-133
STATEMENT_QUEUING Hint 2-134
UNNEST Hint 2-134
USE_BAND Hint 2-135
USE_CONCAT Hint 2-135

USE_CUBE Hint 2-136
USE_HASH Hint 2-136
USE_MERGE Hint 2-136
USE_NL Hint 2-137
USE_NL_WITH_INDEX Hint 2-137
Database Objects 2-138
Schema Objects 2-138
Nonschema Objects 2-139
Database Object Names and Qualifiers 2-139
Database Object Naming Rules 2-140
Schema Object Naming Examples 2-144
Schema Object Naming Guidelines 2-144
Syntax for Schema Objects and Parts in SQL Statements 2-145
How Oracle Database Resolves Schema Object References 2-145
References to Objects in Other Schemas 2-147
References to Objects in Remote Databases 2-147
Creating Database Links 2-147
References to Database Links 2-148
References to Partitioned Tables and Indexes 2-149
References to Object Type Attributes and Methods 2-151

3 Pseudocolumns

Hierarchical Query Pseudocolumns 3-1
CONNECT_BY_ISCYCLE Pseudocolumn 3-1
CONNECT_BY_ISLEAF Pseudocolumn 3-2
LEVEL Pseudocolumn 3-2
Sequence Pseudocolumns 3-3
Where to Use Sequence Values 3-4
How to Use Sequence Values 3-4
Version Query Pseudocolumns 3-6
COLUMN_VALUE Pseudocolumn 3-7
OBJECT_ID Pseudocolumn 3-8
OBJECT_VALUE Pseudocolumn 3-8
ORA_ROWSCN Pseudocolumn 3-9
ROWID Pseudocolumn 3-10
ROWNUM Pseudocolumn 3-11
XMLDATA Pseudocolumn 3-12

4 Operators

About SQL Operators 4-1
- Unary and Binary Operators 4-1
- Operator Precedence 4-2

Arithmetic Operators 4-2
COLLATE Operator 4-3
Concatenation Operator 4-4
Hierarchical Query Operators 4-5
- PRIOR 4-5
- CONNECT_BY_ROOT 4-6

Set Operators 4-6
Multiset Operators 4-6
- MULTISET EXCEPT 4-7
- MULTISET INTERSECT 4-8
- MULTISET UNION 4-9

User-Defined Operators 4-10

5 Expressions

About SQL Expressions 5-1
Simple Expressions 5-3
Compound Expressions 5-3
Calculated Measure Expressions 5-4
- Analytic View Measure Expressions 5-6
- Analytic View Simple Expressions 5-16
- Single Row Function Expression 5-16
- Examples of Calculated Measure Expressions 5-16

CASE Expressions 5-18
Column Expressions 5-20
CURSOR Expressions 5-21
Datetime Expressions 5-23
Function Expressions 5-24
Interval Expressions 5-25
JSON Object Access Expressions 5-26
Model Expressions 5-28
Object Access Expressions 5-30
Placeholder Expressions 5-30
Scalar Subquery Expressions 5-31
Type Constructor Expressions 5-31

Expression Lists 5-33

6 Conditions

About SQL Conditions 6-1
Condition Precedence 6-3
Comparison Conditions 6-4
Simple Comparison Conditions 6-5
Group Comparison Conditions 6-7
Floating-Point Conditions 6-8
Logical Conditions 6-9
Model Conditions 6-10
IS ANY Condition 6-10
IS PRESENT Condition 6-11
Multiset Conditions 6-12
IS A SET Condition 6-12
IS EMPTY Condition 6-13
MEMBER Condition 6-14
SUBMULTISET Condition 6-14
Pattern-matching Conditions 6-15
LIKE Condition 6-15
REGEXP_LIKE Condition 6-19
Null Conditions 6-21
XML Conditions 6-22
EQUALS_PATH Condition 6-22
UNDER_PATH Condition 6-23
SQL/JSON Conditions 6-23
IS JSON Condition 6-24
JSON_EXISTS Condition 6-26
JSON_TEXTCONTAINS Condition 6-29
Compound Conditions 6-31
BETWEEN Condition 6-32
EXISTS Condition 6-33
IN Condition 6-33
IS OF type Condition 6-36

7 Functions

About SQL Functions 7-2
Single-Row Functions 7-4
Numeric Functions 7-4

Character Functions Returning Character Values 7-5
Character Functions Returning Number Values 7-5
Character Set Functions 7-6
Collation Functions 7-6
Datetime Functions 7-6
General Comparison Functions 7-7
Conversion Functions 7-7
Large Object Functions 7-8
Collection Functions 7-8
Hierarchical Functions 7-8
Data Mining Functions 7-9
XML Functions 7-9
JSON Functions 7-10
Encoding and Decoding Functions 7-11
NULL-Related Functions 7-11
Environment and Identifier Functions 7-11
Aggregate Functions 7-12
Analytic Functions 7-14
Object Reference Functions 7-21
Model Functions 7-21
OLAP Functions 7-21
Data Cartridge Functions 7-21
ABS 7-22
ACOS 7-22
ADD_MONTHS 7-23
APPENDCHILDXML 7-24
APPROX_COUNT_DISTINCT 7-25
APPROX_COUNT_DISTINCT_AGG 7-26
APPROX_COUNT_DISTINCT_DETAIL 7-27
APPROX_MEDIAN 7-30
APPROX_PERCENTILE 7-33
APPROX_PERCENTILE_AGG 7-36
APPROX_PERCENTILE_DETAIL 7-37
ASCII 7-41
ASCIISTR 7-42
ASIN 7-42
ATAN 7-43
ATAN2 7-44
AVG 7-45
BFILENAME 7-46
BIN_TO_NUM 7-47

BITAND	7-48
CARDINALITY	7-50
CAST	7-51
CEIL	7-56
CHARTOROWID	7-56
CHR	7-57
CLUSTER_DETAILS	7-59
CLUSTER_DISTANCE	7-62
CLUSTER_ID	7-65
CLUSTER_PROBABILITY	7-67
CLUSTER_SET	7-70
COALESCE	7-72
COLLATION	7-74
COLLECT	7-75
COMPOSE	7-76
CON_DBID_TO_ID	7-77
CON_GUID_TO_ID	7-78
CON_NAME_TO_ID	7-79
CON_UID_TO_ID	7-79
CONCAT	7-80
CONVERT	7-81
CORR	7-83
CORR_*	7-84
CORR_S	7-86
CORR_K	7-86
COS	7-87
COSH	7-87
COUNT	7-88
COVAR_POP	7-90
COVAR_SAMP	7-92
CUBE_TABLE	7-93
CUME_DIST	7-95
CURRENT_DATE	7-97
CURRENT_TIMESTAMP	7-97
CV	7-98
DATAOBJ_TO_MAT_PARTITION	7-100
DATAOBJ_TO_PARTITION	7-100
DBTIMEZONE	7-101
DECODE	7-102
DECOMPOSE	7-103
DELETEXML	7-105

DENSE_RANK 7-106
DEPTH 7-108
DEREF 7-108
DUMP 7-109
EMPTY_BLOB, EMPTY_CLOB 7-111
EXISTSNODE 7-112
EXP 7-113
EXTRACT (datetime) 7-114
EXTRACT (XML) 7-116
EXTRACTVALUE 7-117
FEATURE_COMPARE 7-119
FEATURE_DETAILS 7-121
FEATURE_ID 7-124
FEATURE_SET 7-126
FEATURE_VALUE 7-129
FIRST 7-132
FIRST_VALUE 7-134
FLOOR 7-136
FROM_TZ 7-137
GREATEST 7-138
GROUP_ID 7-139
GROUPING 7-140
GROUPING_ID 7-141
HEXTORAW 7-142
INITCAP 7-142
INSERTCHILDXML 7-143
INSERTCHILDXMLAFTER 7-145
INSERTCHILDXMLBEFORE 7-146
INSERTXMLAFTER 7-147
INSERTXMLBEFORE 7-148
INSTR 7-149
ITERATION_NUMBER 7-151
JSON_ARRAY 7-153
JSON_ARRAYAGG 7-154
JSON_DATAGUIDE 7-156
JSON_OBJECT 7-157
JSON_OBJECTAGG 7-159
JSON_QUERY 7-161
JSON_TABLE 7-167
JSON_VALUE 7-176
LAG 7-181

LAST 7-182
LAST_DAY 7-183
LAST_VALUE 7-184
LEAD 7-187
LEAST 7-188
LENGTH 7-189
LISTAGG 7-190
LN 7-194
LNNVL 7-195
LOCALTIMESTAMP 7-196
LOG 7-197
LOWER 7-197
LPAD 7-198
LTRIM 7-199
MAKE_REF 7-200
MAX 7-200
MEDIAN 7-202
MIN 7-204
MOD 7-206
MONTHS_BETWEEN 7-207
NANVL 7-207
NCHR 7-208
NEW_TIME 7-209
NEXT_DAY 7-210
NLS_CHARSET_DECL_LEN 7-211
NLS_CHARSET_ID 7-211
NLS_CHARSET_NAME 7-212
NLS_COLLATION_ID 7-212
NLS_COLLATION_NAME 7-213
NLS_INITCAP 7-214
NLS_LOWER 7-216
NLS_UPPER 7-216
NLSSORT 7-217
NTH_VALUE 7-220
NTILE 7-222
NULLIF 7-223
NUMTODSINTERVAL 7-224
NUMTOYMINTERVAL 7-225
NVL 7-226
NVL2 7-227
ORA_DM_PARTITION_NAME 7-228

ORA_DST_AFFECTED 7-230
ORA_DST_CONVERT 7-230
ORA_DST_ERROR 7-231
ORA_HASH 7-232
ORA_INVOKING_USER 7-233
ORA_INVOKING_USERID 7-233
PATH 7-234
PERCENT_RANK 7-235
PERCENTILE_CONT 7-237
PERCENTILE_DISC 7-239
POWER 7-241
POWERMULTISET 7-242
POWERMULTISET_BY_CARDINALITY 7-243
PREDICTION 7-244
PREDICTION_BOUNDS 7-248
PREDICTION_COST 7-250
PREDICTION_DETAILS 7-254
PREDICTION_PROBABILITY 7-258
PREDICTION_SET 7-261
PRESENTNNV 7-265
PRESENTV 7-266
PREVIOUS 7-267
RANK 7-268
RATIO_TO_REPORT 7-270
RAWTOHEX 7-271
RAWTONHEX 7-272
REF 7-273
REFTOHEX 7-273
REGEXP_COUNT 7-274
REGEXP_INSTR 7-279
REGEXP_REPLACE 7-283
REGEXP_SUBSTR 7-288
REGR_ (Linear Regression) Functions 7-292
REMAINDER 7-297
REPLACE 7-298
ROUND (date) 7-299
ROUND (number) 7-299
ROW_NUMBER 7-301
ROWIDTOCHAR 7-302
ROWIDTONCHAR 7-303
RPAD 7-304

RTRIM 7-305
SCN_TO_TIMESTAMP 7-306
SESSIONTIMEZONE 7-307
SET 7-308
SIGN 7-308
SIN 7-309
SINH 7-310
SOUNDEX 7-310
SQRT 7-311
STANDARD_HASH 7-312
STATS_BINOMIAL_TEST 7-313
STATS_CROSSTAB 7-314
STATS_F_TEST 7-316
STATS_KS_TEST 7-317
STATS_MODE 7-318
STATS_MW_TEST 7-319
STATS_ONE_WAY_ANOVA 7-321
STATS_T_TEST_* 7-322
    STATS_T_TEST_ONE 7-324
    STATS_T_TEST_PAIRED 7-324
    STATS_T_TEST_INDEP and STATS_T_TEST_INDEPU 7-324
STATS_WSR_TEST 7-326
STDDEV 7-327
STDDEV_POP 7-328
STDDEV_SAMP 7-329
SUBSTR 7-331
SUM 7-332
SYS_CONNECT_BY_PATH 7-334
SYS_CONTEXT 7-335
SYS_DBURIGEN 7-342
SYS_EXTRACT_UTC 7-343
SYS_GUID 7-344
SYS_OP_ZONE_ID 7-345
SYS_TYPEID 7-346
SYS_XMLAGG 7-347
SYS_XMLGEN 7-348
SYSDATE 7-349
SYSTIMESTAMP 7-350
TAN 7-351
TANH 7-351
TIMESTAMP_TO_SCN 7-352

TO_APPROX_COUNT_DISTINCT 7-353
TO_APPROX_PERCENTILE 7-354
TO_BINARY_DOUBLE 7-355
TO_BINARY_FLOAT 7-357
TO_BLOB (bfile) 7-358
TO_BLOB (raw) 7-359
TO_CHAR (bfile|blob) 7-359
TO_CHAR (character) 7-360
TO_CHAR (datetime) 7-361
TO_CHAR (number) 7-363
TO_CLOB (bfile|blob) 7-365
TO_CLOB (character) 7-366
TO_DATE 7-367
TO_DSINTERVAL 7-369
TO_LOB 7-371
TO_MULTI_BYTE 7-372
TO_NCHAR (character) 7-372
TO_NCHAR (datetime) 7-373
TO_NCHAR (number) 7-374
TO_NCLOB 7-375
TO_NUMBER 7-376
TO_SINGLE_BYTE 7-377
TO_TIMESTAMP 7-378
TO_TIMESTAMP_TZ 7-379
TO_YMINTERVAL 7-381
TRANSLATE 7-382
TRANSLATE ... USING 7-383
TREAT 7-385
TRIM 7-386
TRUNC (date) 7-387
TRUNC (number) 7-388
TZ_OFFSET 7-389
UID 7-390
UNISTR 7-390
UPDATEXML 7-391
UPPER 7-392
USER 7-393
USERENV 7-394
VALIDATE_CONVERSION 7-395
VALUE 7-398
VAR_POP 7-399

VAR_SAMP 7-400
VARIANCE 7-401
VSIZE 7-403
WIDTH_BUCKET 7-403
XMLAGG 7-405
XMLCAST 7-406
XMLCDATA 7-407
XMLCOLATTVAL 7-408
XMLCOMMENT 7-409
XMLCONCAT 7-409
XMLDIFF 7-410
XMLELEMENT 7-412
XMLEXISTS 7-415
XMLFOREST 7-415
XMLISVALID 7-416
XMLPARSE 7-417
XMLPATCH 7-418
XMLPI 7-419
XMLQUERY 7-420
XMLROOT 7-421
XMLSEQUENCE 7-422
XMLSERIALIZE 7-424
XMLTABLE 7-425
XMLTRANSFORM 7-428
ROUND and TRUNC Date Functions 7-429
About User-Defined Functions 7-431
    Prerequisites 7-432
    Name Precedence 7-432
        Naming Conventions 7-433

8 Common SQL DDL Clauses

allocate_extent_clause 8-1
constraint 8-3
deallocate_unused_clause 8-30
file_specification 8-32
logging_clause 8-41
parallel_clause 8-44
physical_attributes_clause 8-47
size_clause 8-50

storage_clause 8-51

9 SQL Queries and Subqueries

About Queries and Subqueries 9-1
Creating Simple Queries 9-2
Hierarchical Queries 9-2
    Hierarchical Query Examples 9-5
The UNION [ALL], INTERSECT, MINUS Operators 9-8
Sorting Query Results 9-11
Joins 9-12
    Join Conditions 9-12
    Equijoins 9-12
    Band Joins 9-13
    Self Joins 9-13
    Cartesian Products 9-13
    Inner Joins 9-13
    Outer Joins 9-14
    Antijoins 9-15
    Semijoins 9-16
Using Subqueries 9-16
Unnesting of Nested Subqueries 9-17
Selecting from the DUAL Table 9-18
Distributed Queries 9-18

10 SQL Statements: ADMINISTER KEY MANAGEMENT to ALTER JAVA

Types of SQL Statements 10-1
    Data Definition Language (DDL) Statements 10-2
    Data Manipulation Language (DML) Statements 10-3
    Transaction Control Statements 10-3
    Session Control Statements 10-4
    System Control Statement 10-4
    Embedded SQL Statements 10-4
How the SQL Statement Chapters are Organized 10-4
ADMINISTER KEY MANAGEMENT 10-5
ALTER ANALYTIC VIEW 10-29
ALTER ATTRIBUTE DIMENSION 10-30
ALTER AUDIT POLICY (Unified Auditing) 10-31
ALTER CLUSTER 10-36
ALTER DATABASE 10-40

ALTER DATABASE LINK 10-91
ALTER DIMENSION 10-92
ALTER DISKGROUP 10-96
ALTER FLASHBACK ARCHIVE 10-130
ALTER FUNCTION 10-133
ALTER HIERARCHY 10-134
ALTER INDEX 10-135
ALTER INDEXTYPE 10-159
ALTER INMEMORY JOIN GROUP 10-161
ALTER JAVA 10-163

11 SQL Statements: ALTER LIBRARY to ALTER SESSION

ALTER LIBRARY 11-1
ALTER LOCKDOWN PROFILE 11-2
ALTER MATERIALIZED VIEW 11-14
ALTER MATERIALIZED VIEW LOG 11-35
ALTER MATERIALIZED ZONEMAP 11-44
ALTER OPERATOR 11-48
ALTER OUTLINE 11-51
ALTER PACKAGE 11-52
ALTER PLUGGABLE DATABASE 11-54
ALTER PROCEDURE 11-73
ALTER PROFILE 11-75
ALTER RESOURCE COST 11-78
ALTER ROLE 11-80
ALTER ROLLBACK SEGMENT 11-82
ALTER SEQUENCE 11-85
ALTER SESSION 11-87
    Initialization Parameters and ALTER SESSION 11-95
    Session Parameters and ALTER SESSION 11-96

12 SQL Statements: ALTER SYNONYM to COMMENT

ALTER SYNONYM 12-1
ALTER SYSTEM 12-3
ALTER TABLE 12-29
ALTER TABLESPACE 12-171
ALTER TABLESPACE SET 12-189
ALTER TRIGGER 12-191
ALTER TYPE 12-193

ALTER USER 12-195
ALTER VIEW 12-206
ANALYZE 12-209
ASSOCIATE STATISTICS 12-218
AUDIT (Traditional Auditing) 12-222
AUDIT (Unified Auditing) 12-238
CALL 12-243
COMMENT 12-247

13 SQL Statements: COMMIT to CREATE JAVA

COMMIT 13-1
CREATE ANALYTIC VIEW 13-6
CREATE ATTRIBUTE DIMENSION 13-14
CREATE AUDIT POLICY (Unified Auditing) 13-23
CREATE CLUSTER 13-33
CREATE CONTEXT 13-42
CREATE CONTROLFILE 13-44
CREATE DATABASE 13-52
CREATE DATABASE LINK 13-69
CREATE DIMENSION 13-75
CREATE DIRECTORY 13-81
CREATE DISKGROUP 13-84
CREATE EDITION 13-93
CREATE FLASHBACK ARCHIVE 13-96
CREATE FUNCTION 13-99
CREATE HIERARCHY 13-101
CREATE INDEX 13-105
CREATE INDEXTYPE 13-142
CREATE INMEMORY JOIN GROUP 13-146
CREATE JAVA 13-148

14 SQL Statements: CREATE LIBRARY to CREATE SCHEMA

CREATE LIBRARY 14-1
CREATE LOCKDOWN PROFILE 14-3
CREATE MATERIALIZED VIEW 14-4
CREATE MATERIALIZED VIEW LOG 14-38
CREATE MATERIALIZED ZONEMAP 14-49
CREATE OPERATOR 14-58
CREATE OUTLINE 14-62

CREATE PACKAGE 14-65
CREATE PACKAGE BODY 14-67
CREATE PFILE 14-69
CREATE PLUGGABLE DATABASE 14-71
CREATE PROCEDURE 14-93
CREATE PROFILE 14-95
CREATE RESTORE POINT 14-103
CREATE ROLE 14-106
CREATE ROLLBACK SEGMENT 14-110
CREATE SCHEMA 14-113

15 SQL Statements: CREATE SEQUENCE to DROP CLUSTER

CREATE SEQUENCE 15-1
CREATE SPFILE 15-8
CREATE SYNONYM 15-12
CREATE TABLE 15-17
CREATE TABLESPACE 15-141
CREATE TABLESPACE SET 15-162
CREATE TRIGGER 15-164
CREATE TYPE 15-166
CREATE TYPE BODY 15-168
CREATE USER 15-169
CREATE VIEW 15-179
DELETE 15-195
DISASSOCIATE STATISTICS 15-204
DROP ANALYTIC VIEW 15-206
DROP ATTRIBUTE DIMENSION 15-207
DROP AUDIT POLICY (Unified Auditing) 15-207
DROP CLUSTER 15-209

16 SQL Statements: DROP CONTEXT to DROP JAVA

DROP CONTEXT 16-1
DROP DATABASE 16-2
DROP DATABASE LINK 16-3
DROP DIMENSION 16-4
DROP DIRECTORY 16-5
DROP DISKGROUP 16-6
DROP EDITION 16-8
DROP FLASHBACK ARCHIVE 16-9

DROP FUNCTION 16-10
DROP HIERARCHY 16-11
DROP INDEX 16-12
DROP INDEXTYPE 16-14
DROP INMEMORY JOIN GROUP 16-15
DROP JAVA 16-16

17 SQL Statements: DROP LIBRARY to DROP SYNONYM

DROP LIBRARY 17-1
DROP LOCKDOWN PROFILE 17-2
DROP MATERIALIZED VIEW 17-3
DROP MATERIALIZED VIEW LOG 17-5
DROP MATERIALIZED ZONEMAP 17-7
DROP OPERATOR 17-8
DROP OUTLINE 17-9
DROP PACKAGE 17-10
DROP PLUGGABLE DATABASE 17-11
DROP PROCEDURE 17-13
DROP PROFILE 17-14
DROP RESTORE POINT 17-15
DROP ROLE 17-16
DROP ROLLBACK SEGMENT 17-17
DROP SEQUENCE 17-18
DROP SYNONYM 17-19

18 SQL Statements: DROP TABLE to LOCK TABLE

DROP TABLE 18-1
DROP TABLESPACE 18-5
DROP TABLESPACE SET 18-8
DROP TRIGGER 18-9
DROP TYPE 18-10
DROP TYPE BODY 18-12
DROP USER 18-13
DROP VIEW 18-15
EXPLAIN PLAN 18-16
FLASHBACK DATABASE 18-20
FLASHBACK TABLE 18-23
GRANT 18-30
INSERT 18-61

LOCK TABLE 18-80

19 SQL Statements: MERGE to UPDATE

MERGE 19-1
NOAUDIT (Traditional Auditing) 19-10
NOAUDIT (Unified Auditing) 19-15
PURGE 19-20
RENAME 19-22
REVOKE 19-24
ROLLBACK 19-35
SAVEPOINT 19-38
SELECT 19-39
SET CONSTRAINT[S] 19-125
SET ROLE 19-126
SET TRANSACTION 19-129
TRUNCATE CLUSTER 19-132
TRUNCATE TABLE 19-134
UPDATE 19-138

A How to Read Syntax Diagrams

Graphic Syntax Diagrams A-1
    Required Keywords and Parameters A-2
    Optional Keywords and Parameters A-3
    Syntax Loops A-4
    Multipart Diagrams A-4
Backus-Naur Form Syntax A-5

B Automatic and Manual Locking Mechanisms During SQL Operations

Automatic Locks in DML Operations B-1
Automatic Locks in DDL Operations B-4
    Exclusive DDL Locks B-4
    Share DDL Locks B-5
    Breakable Parse Locks B-5
Manual Data Locking B-5

C Oracle and Standard SQL

ANSI Standards C-1
ISO Standards C-2

Oracle Compliance to Core SQL C-3
Oracle Support for Optional Features of SQL/Foundation C-8
Oracle Compliance with SQL/CLI C-26
Oracle Compliance with SQL/PSM C-26
Oracle Compliance with SQL/MED C-26
Oracle Compliance with SQL/OLB C-26
Oracle Compliance with SQL/JRT C-26
Oracle Compliance with SQL/XML C-26
Oracle Compliance with FIPS 127-2 C-32
Oracle Extensions to Standard SQL C-33
Oracle Compliance with Older Standards C-33
Character Set Support C-34

D Oracle Regular Expression Support

Multilingual Regular Expression Syntax D-1
Regular Expression Operator Multilingual Enhancements D-2
Perl-influenced Extensions in Oracle Regular Expressions D-3

E Oracle SQL Reserved Words and Keywords

Oracle SQL Reserved Words E-1
Oracle SQL Keywords E-4

F Extended Examples

Using Extensible Indexing F-1
Using XML in SQL Statements F-8

Index

Preface

This reference contains a complete description of the Structured Query Language (SQL) used to manage information in an Oracle Database. Oracle SQL is a superset of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) SQL standard.

This Preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

The *Oracle Database SQL Language Reference* is intended for all users of Oracle SQL.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see these Oracle resources:

- *Oracle Database PL/SQL Language Reference* for information on PL/SQL, the procedural language extension to Oracle SQL
- *Pro*C/C++ Programmer's Guide* and *Pro*COBOL Programmer's Guide* for detailed descriptions of Oracle embedded SQL

Many of the examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database installation. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle Database SQL Language Reference

This preface contains:

- Changes in Oracle Database 12*c* Release 2 (12.2.0.1)
- Changes in Oracle Database 12*c* Release 1 (12.1.0.2)
- Changes in Oracle Database 12*c* Release 1 (12.1.0.1)

Changes in Oracle Database 12*c* Release 2 (12.2.0.1)

The following are changes in *Oracle Database SQL Language Reference* for Oracle Database 12*c* Release 2 (12.2.0.1).

New Features

The following features are new in this release:

Long Identifiers

The maximum length for most database object names has increased from 30 bytes to 128 bytes.

See Database Object Naming Rules .

Data-Bound Collation and Case-Insensitive Databases

Data-bound collation allows you to declare character comparison rules at the column level. The collation declared for a column is automatically applied to all collation-sensitive SQL operations that reference the column. This enables applications to consistently apply language-specific comparison rules to the exact data that requires these rules. Data-bound collation also allows you to declare a case-insensitive collation for a table or a schema, so that all columns in a table or schema can be compared in a case-insensitive manner. This enables you to create a case-insensitive database.

See Data-Bound Collation.

Features that Introduce New SQL Statements

The following features introduce new SQL statements:

- **Analytic views** are metadata objects that provide a fast and efficient way to create and store analytic queries of data in existing database tables and views. With analytic views you can easily create complex analytic queries on large amounts of hierarchical and dimensional data. Attribute dimensions, hierarchies, and analytic views are new database schema objects.

See the following new statements:

  – CREATE ANALYTIC VIEW
  – CREATE ATTRIBUTE DIMENSION
  – CREATE HIERARCHY
  – ALTER ANALYTIC VIEW
  – ALTER ATTRIBUTE DIMENSION
  – ALTER HIERARCHY
  – DROP ANALYTIC VIEW
  – DROP ATTRIBUTE DIMENSION
  – DROP HIERARCHY

- **Join groups** allow you to optimize join queries for table columns that are populated in the In-Memory Column Store.

  See the following new statements:

  – CREATE INMEMORY JOIN GROUP
  – ALTER INMEMORY JOIN GROUP
  – DROP INMEMORY JOIN GROUP

- **Oracle Sharding** supports distribution and replication of data across a pool of discrete Oracle databases that share no hardware or software.

  See the following new statements:

  – CREATE TABLESPACE SET
  – ALTER TABLESPACE SET
  – DROP TABLESPACE SET

  See the new clauses SHARDED and DUPLICATED of `CREATE TABLE`.

- You can use **PDB lockdown profiles** in a multitenant container database (CDB) to restrict user operations in PDBs.

  See the following new statements:

  – CREATE LOCKDOWN PROFILE
  – ALTER LOCKDOWN PROFILE
  – DROP LOCKDOWN PROFILE

ADMINISTER KEY MANAGEMENT Enhancements

- You can instruct the database to force a keystore open for certain operations.

  See Notes on the FORCE KEYSTORE Clause.

- You can use keystore passwords stored in an external store.

  See the new clause `EXTERNAL STORE` in Notes on Specifying Keystore Passwords.

- You can specify an encryption algorithm for a master key.

  See the new clause `USING ALGORITHM` of *set_key* and *create_key*.

ALTER SESSION Enhancement

- When you switch to a different container in a CDB, you can specify the service you would like to use in the new container.

  See the new clause SERVICE.

AUDIT and NOAUDIT (Unified Auditing) Enhancements

- You can enable or disable unified audit policies for users who have been directly granted specific roles.

  See the new clauses *by_users_with_roles* of `AUDIT` and `NOAUDIT`.

CREATE DATABASE and ALTER DATABASE Enhancements

- You can specify a local temporary tablespace for your database.

  See the clause *default_temp_tablespace* of `CREATE DATABASE` and the DEFAULT [LOCAL] TEMPORARY TABLESPACE Clause of `ALTER DATABASE`.

- You can specify local undo mode or shared undo mode for a CDB.

  See the new *undo_mode_clause* of `CREATE DATABASE` and *undo_mode_clause* of `ALTER DATABASE`.

- For Oracle Real Application Clusters (Oracle RAC) or Oracle RAC One Node databases, you can control the number of instances on a physical standby that Redo Apply uses.

  See the new clause USING INSTANCES of `ALTER DATABASE`.

- You can perform offline encryption or decryption of a data file using Transparent Data Encryption (TDE).

  See the new clauses ENCRYPT | DECRYPT of `ALTER DATABASE`.

CREATE DIRECTORY Enhancement

- You can create directories that are application common objects. Application common objects can be shared by application PDBs in an application container.

  See the new clause SHARING of `CREATE DIRECTORY`.

CREATE DISKGROUP and ALTER DISKGROUP Enhancements

- You can create Oracle ASM flex disk groups, which support quota groups and file groups. Flex disk groups enable you to define a quota limit for the files of a group of databases within a disk group.

  See:

  - The new `FLEX` keyword in the REDUNDANCY Clause of `CREATE DISKGROUP`.
  - The new *quotagroup_clauses* of `ALTER DISKGROUP`.
  - The new *filegroup_clauses* of `ALTER DISKGROUP`.

- You can pause, restart, and change the power of active disk group rebalance operations. You can also specify which phases of a rebalance operation to perform.

  See the *rebalance_diskgroup_clause* of `ALTER DISKGROUP`.

- The following new Oracle ASM disk group attributes are listed in Table 13-2:
  - LOGICAL_SECTOR_SIZE allows you to set the logical sector size of a disk group.
  - PREFERRED_READ.ENABLED allows you to specify whether preferred read functionality is enabled for a disk group in an Oracle extended cluster.

CREATE INDEX, ALTER INDEX, and DROP INDEX Enhancements

- You can specify whether to invalidate dependent cursors while creating an index, rebuilding an index, marking an index UNUSABLE, or dropping an index.

  See the new clause { DEFERRED | IMMEDIATE } INVALIDATION of CREATEINDEX, ALTERINDEX, and DROPINDEX.
- Advanced index compression provides a HIGH compression level.

  See the new HIGH keyword of the clause advanced_index_compression of CREATE INDEX.

CREATE JAVA Enhancement

- You can create Java schema objects that are application common objects. Application common objects can be shared by application PDBs in an application container.

  See the new clause SHARING of CREATE JAVA.

CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW Enhancements

- A real-time materialized view provides fresh data to user queries even when the materialized view is not in sync with its base tables due to data changes.

  See the new clause { ENABLE | DISABLE } ON QUERY COMPUTATION of CREATEMATERIALIZEDVIEW and ALTERMATERIALIZEDVIEW.
- The ON STATEMENT refresh mode causes an automatic refresh to occur every time a DML operation is performed on any of a materialized view's base tables.

  See the new ON STATEMENT Clause of CREATE MATERIALIZED VIEW.

CREATE PLUGGABLE DATABASE and ALTER PLUGGABLE DATABASE Enhancements

- An **application container** is a component of a CDB that stores data for one or more applications. It consists of an application root, an optional application seed, and application PDBs.

  See the new clauses AS APPLICATION CONTAINER and AS SEED of CREATE PLUGGABLE DATABASE.
- You can perform the following operations in an application container:
  - Install, patch, upgrade, and uninstall applications
  - Register application versions and patch numbers
  - Keep applications in sync between the application root and application PDBs

  See the new *application_clauses* of ALTER PLUGGABLE DATABASE.
- You can create a **refreshable PDB** when cloning a PDB. Changes in the source PDB can be propagated to the refreshable PDB, either manually or automatically.

  See the new *pdb_refresh_mode_clause* of CREATE PLUGGABLE DATABASE and the new clauses REFRESH and *pdb_refresh_mode_clause* of ALTER PLUGGABLE DATABASE.

- A **proxy PDB** references a PDB in a different CDB and provides fully functional access to the referenced PDB.

  See the new clauses AS PROXY FROM and HOST and PORT of `CREATE PLUGGABLE DATABASE`.

- You can relocate a PDB from one CDB to another.

  See the new clause RELOCATE of `CREATE PLUGGABLE DATABASE`.

- When cloning a PDB, you can instruct the database to clone a tablespace using storage snapshots or clone the data model definition of a tablespace, but not the tablespace's data.

  See the new clauses { SNAPSHOT COPY | NO DATA } of `CREATE PLUGGABLE DATABASE`.

- When plugging in a PDB, you can instruct the database to copy or move tablespace files to a new location.

  See the new clauses { COPY | MOVE | NOCOPY } of `CREATE PLUGGABLE DATABASE`.

- In earlier releases, you could specify a permanent default tablespace only when creating a PDB from seed. You can now also specify a permanent default tablespace when cloning a PDB or plugging in a PDB.

  See the clause *default_tablespace* of `CREATE PLUGGABLE DATABASE`.

- You can use parallel execution servers to copy a new PDB's data files to a new location. This may result in faster creation of the PDB.

  See the new *parallel_pdb_creation_clause* of `CREATE PLUGGABLE DATABASE`.

CREATE PROFILE and ALTER PROFILE Enhancements

- In profiles, you can specify the permitted number of consecutive days of no logins after which an account will be locked.

  See the new INACTIVE_ACCOUNT_TIME setting of `CREATE PROFILE` and `ALTER PROFILE`.

CREATE RESTORE POINT Enhancement

- You can create restore points for a PDB.

  See the new clauses CLEAN and FOR PLUGGABLE DATABASE of `CREATE RESTORE POINT`.

CREATE SEQUENCE Enhancement

- You can create sequences that are application common objects. Application common objects can be shared by application PDBs in an application container.

  See the new clause SHARING of `CREATE SEQUENCE`.

CREATE SYNONYM Enhancement

- You can create synonyms that are application common objects. Application common objects can be shared by application PDBs in an application container.

  See the new clause SHARING of `CREATE SYNONYM`.

CREATE TABLE and ALTER TABLE Enhancements

- You can create partitioned external tables.

  See the *external_table_clause* of `CREATE TABLE` and the clause *alter_external_table* of `ALTER TABLE`.

- You can specify constraints on external tables.

  See External Table Constraints .

- You can specify up to 16 partitioning key columns for list-partitioned tables, and up to 16 subpartitioning key columns for composite-partitioned tables that use list subpartitioning. In earlier releases, you could specify only one partitioning or subpartitioning key column.

  See the *list_values_clause* of `CREATE TABLE` and `ALTER TABLE`.

- You can create an automatic list-partitioned table. This type of table enables the database to create additional list partitions on demand.

  See the new clauses AUTOMATIC of `CREATE TABLE` and *alter_automatic_partitioning* of `ALTER TABLE`.

- You can change a nonpartitioned table to a partitioned table.

  See the new clause *modify_to_partitioned* of `ALTER TABLE`.

- You can create a table that matches the structure of an existing partitioned table. The two tables are then eligible for exchanging partitions and subpartitions.

  See the new clause FOR EXCHANGE WITH TABLE of `CREATE TABLE`.

- You can specify which rows to preserve during the following operations: moving, splitting, or merging table partitions or subpartitions; moving a table; or converting a nonpartitioned table to a partitioned table.

  See the new clause *filter_condition* of `ALTER TABLE`.

- You can specify read-only mode for a table, partition, or subpartition.

  See the new *read_only_clause* of `CREATE TABLE` and `ALTER TABLE`.

- DML operations on a table are allowed while splitting its partitions and subpartitions.

  See the new `ONLINE` keyword of the clauses *split_table_partition* and *split_table_subpartition* of `ALTER TABLE`.

- Nonpartitioned tables can be moved as an online operation without blocking any concurrent DML operations. A table move operation now also supports automatic index maintenance as part of the move.

  See the *move_table_clause* of `ALTER TABLE`.

- You can specify whether to invalidate dependent cursors while performing table partition maintenance operations.

  See the new clause { DEFERRED | IMMEDIATE } INVALIDATION of `ALTER TABLE`.

- You can create tables that are application common objects. Application common objects can be shared by application PDBs in an application container.

  See the new clause SHARING of `CREATE TABLE`.

- Table column encryption supports the following additional algorithms: `ARIA192`, `ARIA256`, `GOST256`, and `SEED128`.

See the clause *encryption_spec* of CREATE TABLE and ALTER TABLE.

- When specifying default In-Memory Column Store (IM column store) settings for a table, you can specify the Oracle RAC or Oracle Active Data Guard instances in which the table is eligible to be populated in the IM column store.

  See the new clause FOR SERVICE of CREATE TABLE and ALTER TABLE.

- You can create Automatic Data Optimization policies that enable, disable, or recompress tables in the IM column store.

  See the new clause *ilm_inmemory_policy* of CREATE TABLE and ALTER TABLE.

- Automatic Data Optimization compression policies support an additional compression method: COLUMN STORE COMPRESS FOR QUERY.

  See the clause *ilm_compression_policy* of CREATE TABLE and ALTER TABLE.

CREATE TABLESPACE and ALTER TABLESPACE Enhancements

- Tablespace encryption enhancements: You can encrypt both offline and online tablespaces. You can encrypt undo tablespaces, temporary tablespaces, and the SYSTEM and SYSAUX tablespaces. Tablespace encryption supports the following additional algorithms: ARIA192, ARIA256, GOST256, and SEED128.

  See the new *tablespace_encryption_clause* of CREATE TABLESPACE and the new clause *alter_tablespace_encryption* of ALTER TABLESPACE.

- You can create local temporary tablespaces, which are useful for Oracle Real Application Clusters and Oracle Flex Clusters. They store a separate, nonshared temp file for each database instance, which can improve I/O performance.

  See the new LOCAL TEMPORARY TABLESPACE clause of the *temporary_tablespace_clause* of CREATE TABLESPACE.

- You can specify a default index compression method for a tablespace.

  See the new clause *default_index_compression* of CREATE TABLESPACE and the clause *default_tablespace_params* of ALTER TABLESPACE.

- In earlier releases of Oracle Database, you could specify a default table compression method for a tablespace. You can still do this in Oracle Database 12*c* Release 2 (12.2), however, the syntax now includes the TABLE keyword to differentiate it from the new default index compression syntax.

  See the new clause *default_table_compression* of CREATE TABLESPACE and the clause *default_tablespace_params* of ALTER TABLESPACE.

CREATE USER and ALTER USER Enhancements

- You can assign a local temporary tablespace to a user.

  See the new LOCAL keyword for the TEMPORARY TABLESPACE clause of CREATEUSER and ALTERUSER.

CREATE VIEW Enhancement

- You can create views that are application common objects. Application common objects can be shared by application PDBs in an application container.

  See the new clause SHARING of CREATE VIEW.

FLASHBACK DATABASE Enhancement

- You can flash back a PDB.

  See the new clause PLUGGABLE of `FLASHBACK DATABASE`.

SELECT Enhancement

- The *`query_table_expression`* clause of the `FROM` clause of the `SELECT` statement now accepts a hierarchy or an analytic view in a subquery.

  See the clause *query_table_expression* of `SELECT`.

New Operator

- The new `COLLATE` operator determines the collation for an expression. This operator enables you to override the collation that the database would have derived for the expression using standard collation derivation rules.

  See COLLATE Operator.

New or Enhanced Expressions

- The new calculated measure expression defines a calculated measure in an analytic view.

  See Calculated Measure Expressions.

- JSON object access expressions have been enhanced to let you access specific elements of a JSON array.

  See the clause *`array_step`* of JSON Object Access Expressions.

Enhanced Condition

- The `JSON_EXISTS` condition now lets you pass values to the path expression.

  See the new clause *JSON_passing_clause* of `JSON_EXISTS`.

New or Enhanced Functions

- **Approximate Query Processing Functions**

  The following new functions return approximate results with negligible deviation from the exact result:

  - APPROX_MEDIAN takes a numeric or datetime value and returns an approximate median value.
  - APPROX_PERCENTILE takes a percentile value and a sort specification, and returns the value that would fall into that percentile value with respect to the sort specification.

  The following new functions support materialized view-based summary aggregation strategies for approximate distinct value counts:

  - APPROX_COUNT_DISTINCT_DETAIL calculates information about the approximate number of rows that contain a distinct value for an expression and returns a `BLOB` value, called a detail, which contains that information in a special format.

  - APPROX_COUNT_DISTINCT_AGG takes as its input a column of details containing information about approximate distinct value counts, and enables you to perform aggregations of those counts.
  - TO_APPROX_COUNT_DISTINCT takes as its input a detail containing information about an approximate distinct value count, and converts it to a `NUMBER` value.

  The following new functions support materialized view-based summary aggregation strategies for approximate percentile values:

  - APPROX_PERCENTILE_DETAIL calculates approximate percentile information for the values and returns a `BLOB` value, called a detail, which contains that information in a special format.
  - APPROX_PERCENTILE_AGG takes as its input a column of details containing approximate percentile information, and enables you to perform aggregations of that information.
  - TO_APPROX_PERCENTILE takes as its input a detail containing approximate percentile information, a percentile value, and a sort specification, and returns an approximate value that would fall into that percentile value with respect to the sort specification.

- **Collation Functions**

  The following new functions return information about collation settings:

  - COLLATION returns the name of the derived collation for an expression.
  - NLS_COLLATION_ID takes as its argument a collation name and returns the corresponding collation ID number.
  - NLS_COLLATION_NAME takes as its argument a collation ID number and returns the corresponding collation name.

- **Conversion Functions**

  The following conversion functions now allow you to specify a value to be returned if a conversion error occurs:

  - CAST - In addition, this function now lets you to specify a format model and NLS parameters to be applied for the conversion.
  - TO_BINARY_DOUBLE
  - TO_BINARY_FLOAT
  - TO_DATE
  - TO_DSINTERVAL
  - TO_NUMBER
  - TO_TIMESTAMP
  - TO_TIMESTAMP_TZ
  - TO_YMINTERVAL

  The following new function lets you determine whether an expression can be converted to a specified data type:

  - VALIDATE_CONVERSION

  The following new functions allow additional data types to be converted to `BLOB`, character, and `CLOB` values:

  - TO_BLOB (bfile) converts a BFILE value to a BLOB value.
  - TO_CHAR (bfile|blob) converts BFILE or BLOB data to the database character set.
  - TO_CLOB (bfile|blob) converts BFILE or BLOB data to the database character set and returns the data as a CLOB value.

- **Data Mining Functions**

  The data mining functions are enhanced so they can be applied to models built using the native algorithms of Oracle, as well as those built using R through the extensibility mechanism of Oracle Advanced Analytics. See Data Mining Functions .

  The following are new data mining functions:

  - FEATURE_COMPARE uses a Feature Extraction model to compare two different documents.
  - ORA_DM_PARTITION_NAME returns the name of the partition associated with the input row.

  The syntax of the following functions is enhanced so that the functions can use the GROUPING hint when scoring a partitioned model:

  - CLUSTER_DETAILS
  - CLUSTER_DISTANCE
  - CLUSTER_ID
  - CLUSTER_PROBABILITY
  - CLUSTER_SET
  - FEATURE_COMPARE
  - FEATURE_DETAILS
  - FEATURE_ID
  - FEATURE_SET
  - FEATURE_VALUE
  - ORA_DM_PARTITION_NAME
  - PREDICTION
  - PREDICTION_BOUNDS
  - PREDICTION_COST
  - PREDICTION_DETAILS
  - PREDICTION_PROBABILITY
  - PREDICTION_SET

- **JSON Functions**

  The following new functions enable you to query and generate JavaScript Object Notation (JSON) data:

  - JSON_ARRAY takes as its input one or more SQL expressions, converts each expression to a JSON value, and returns a JSON array that contains those JSON values.

  - JSON_ARRAYAGG takes as its input a column of SQL expressions, converts each expression to a JSON value, and returns a single JSON array that contains those JSON values.
  - JSON_DATAGUIDE takes as its input a table column of JSON data. Each row in the column is referred to as a JSON document. For each JSON document in the column, this function returns a `CLOB` value that contains a flat data guide for that JSON document.
  - JSON_OBJECT takes as its input one or more property key-value pairs, and returns a JSON object that contains an object member for each of those key-value pairs.
  - JSON_OBJECTAGG takes as its input a property key-value pair, constructs an object member for each key-value pair, and returns a single JSON object that contains those object members.

  The following JSON functions have been enhanced to let you specify a return value when no match is found:

  - JSON_QUERY
  - JSON_VALUE
- **LISTAGG Function**
  - LISTAGG now allows you to control how the function behaves when the return value exceeds the maximum length of the return data type.

New or Enhanced Privileges

The following are new or enhanced system privileges and object privileges:

- The following system privileges, which are listed in Table 18-1, have been enhanced:
  - `CREATE JOB` and `CREATE ANY JOB` now allow you to manage resource objects and incompatibility resource objects.
- The following new system privileges are listed in Table 18-1:
  - These system privileges allow the creation and management of analytic views:

    `CREATE ANALYTIC VIEW`, `CREATE ANY ANALYTIC VIEW`, `ALTER ANY ANALYTIC VIEW`, `DROP ANY ANALYTIC VIEW`

    `CREATE ATTRIBUTE DIMENSION`, `CREATE ANY ATTRIBUTE DIMENSION`, `ALTER ANY ATTRIBUTE DIMENSION`, `DROP ANY ATTRIBUTE DIMENSION`

    `CREATE HIERARCHY`, `CREATE ANY HIERARCHY`, `ALTER ANY HIERARCHY`, and `DROP ANY HIERARCHY`
  - `CREATE LOCKDOWN PROFILE`, `ALTER LOCKDOWN PROFILE`, and `DROP LOCKDOWN PROFILE` allow you to manage PDB lockdown profiles.
  - `INHERIT ANY REMOTE PRIVILEGES` allows you to execute definer's rights procedures or functions that contain current user database links.
  - `USE ANY JOB RESOURCE` allows you to associate any schedule resource object with any program or job in the grantee's schema.
- The following new object privileges are listed in Table 18-2:

  - `INHERIT REMOTE PRIVILEGES` can be granted on a user to users and roles. It allows the user on whom this privilege is granted to execute definer's rights procedures or functions that contain current user database links and are owned by the grantee.
  - The `USE` privilege can be granted on job scheduler objects. It allows you to associate the specified scheduler resource object with programs and jobs.

New Hints

The following are new hints:

- The CONTAINERS Hint lets you pass a hint to the query of each PDB in a CDB or application container during a `SELECT` ... *`containers_clause`* ... query.
- The FRESH_MV Hint is part of the new real-time materialized view feature. This hint allows you to fetch up-to-date data from a stale real-time materialized view.
- The GROUPING Hint applies to data mining scoring functions when scoring partitioned models.
- The USE_BAND Hint and NO_USE_BAND Hint allow you to use or exclude band joins in a query. Band joins are new for this release. For more information, see Band Joins.

Deprecated Features

The following features are deprecated in this release, and may be desupported in a future release:

- The Oracle Multimedia support for object types that comply with the first edition of the ISO/IEC 13249-5:2001 SQLMM Part5:StillImage standard (commonly referred to as the SQL/MM Still Image standard) is deprecated.

  See *Oracle Multimedia Reference* for more information.
- The XMLROOT function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you instead use the SQL/XML function `XMLSERIALIZE` with a version number.

  See *Oracle XML DB Developer's Guide* for more information on the `XMLSERIALIZE` function.

Desupported Features

Some features previously described in this document are desupported in Oracle Database 12*c* Release 2 (12.2). See *Oracle Database Upgrade Guide* for a list of desupported features.

Changes in Oracle Database 12*c* Release 1 (12.1.0.2)

The following are changes in *Oracle Database SQL Language Reference* for Oracle Database 12*c* Release 1 (12.1.0.2).

New Features

The following features are new in this release:

- The **In-Memory Column Store** (IM column store) is an optional, static SGA pool that stores copies of tables and partitions in a special columnar format optimized for rapid scans.

  See the *inmemory_table_clause* of CREATE TABLE, the *inmemory_clause* of CREATE TABLESPACE, and the *inmemory_table_clause* of CREATE MATERIALIZED VIEW

  See the following hints:

  – INMEMORY Hint

  – NO_INMEMORY Hint

  – INMEMORY_PRUNING Hint

  – NO_INMEMORY_PRUNING Hint

- Oracle Database now supports **JavaScript Object Notation (JSON)**.

  See the following conditions:

  – IS JSON Condition

  – JSON_EXISTS Condition

  – JSON_TEXTCONTAINS Condition

  See the following functions:

  – JSON_QUERY

  – JSON_TABLE

  – JSON_VALUE

  See "JSON Object Access Expressions"

- **Attribute clustering** lets you cluster table data in close physical proximity based on the content of specified columns.

  See the *attribute_clustering_clause* of CREATE TABLE and the *attribute_clustering_clause* of ALTER TABLE

  See the following hints:

  – CLUSTERING Hint

  – NO_CLUSTERING Hint

- **Zone maps** let you reduce the I/O and CPU costs of table scans by tracking the sets of contiguous data blocks, or zones, in which certain column values are stored. You can use zone maps with or without attribute clustering.

  See the statements CREATE MATERIALIZED ZONEMAP, ALTER MATERIALIZED ZONEMAP, and DROP MATERIALIZED ZONEMAP, and the zonemap_clause of CREATE TABLE

  See the NO_ZONEMAP Hint and the function SYS_OP_ZONE_ID

- You can now create **range-partitioned hash clusters**.

  See the *cluster_range_partitions* clause of CREATE CLUSTER and the *allocate_extent_clause* of ALTER CLUSTER

- The new function **APPROX_COUNT_DISTINCT** returns the approximate number of distinct values for a column. This function is an alternative to the COUNT function. It processes large amounts of data significantly faster than COUNT, with negligible deviation from the exact result.

See APPROX_COUNT_DISTINCT

- A new type of index compression called **advanced index compression** lets you improve compression ratios significantly while still providing efficient access to indexes.

  See the advanced_index_compression clause of `CREATE INDEX`

- For tables compressed with Hybrid Columnar Compression, you can now control whether **row-level locking** is used during DML operations.

  See the [NO] ROW LEVEL LOCKING clause of `CREATE TABLE`

- The database now supports **force full database caching mode**, which allows you to designate the entire database, including NOCACHE LOBs, as eligible for caching in the buffer cache.

  See the [NO] FORCE FULL DATABASE CACHING clause of `ALTER DATABASE`

- When you grant a database role to a user who is responsible for CBAC grants, you can now specify **`WITH DELEGATE OPTION`** in the `GRANT` statement to prevent giving the grantee additional privileges on the role. `WITH DELEGATE OPTION` is an alternative to `WITH ADMIN OPTION`. It enables a role to be granted to program units, but it does not permit the granting of the role to other principals or the administration of the role itself.

  See the WITH DELEGATE OPTION clause of `GRANT`

- The new **`READ` object privilege** and **`READ ANY TABLE` system privilege** allow users to query tables, materialized views, views, and their synonyms.

  The `READ` object privilege is an alternative to the `SELECT` object privilege. In addition to querying objects, the `SELECT` object privilege allows users lock rows of a table with the `LOCK TABLE` and `SELECT` ... `FOR UPDATE` statements. The `READ` object privilege only allows users to query objects. See Table 18-2 for more information.

  The `READ ANY TABLE` system privilege is an alternative to the `SELECT ANY TABLE` system privilege. In addition to querying objects, the `SELECT ANY TABLE` privilege allows users to lock rows of a table with the `SELECT` ... `FOR UPDATE` statement. The `READ ANY TABLE` privilege only allows users to query objects. See Table 18-1 for more information.

Changes in Oracle Database 12*c* Release 1 (12.1.0.1)

The following are changes in *Oracle Database SQL Language Reference* for Oracle Database 12*c* Release 1 (12.1.0.1).

New Features

The following features are new in this release:

Features that Introduce New SQL Statements

The following features introduce new SQL statements:

- The **multitenant architecture** offers the capability that enables an Oracle database to function as a multitenant container database (CDB). A CDB is an Oracle database that includes one or more pluggable databases (PDBs). A PDB is a portable collection of schemas, schema objects, and nonschema objects that appears to an Oracle client as a non-CDB. You can unplug a PDB from a CDB and plug it into a different CDB.

  See the following new statements:

  – CREATE PLUGGABLE DATABASE
  – ALTER PLUGGABLE DATABASE
  – DROP PLUGGABLE DATABASE

- **Unified auditing** provides a full set of enhanced auditing features. It enables you to create named unified audit policies, enable or disable unified audit policies, apply users to or exclude users from policies, and set whether an audit record is created if the audited behavior fails, succeeds, or both. It also enables you to capture application context values in audit records.

  See the following new statements:
  – CREATE AUDIT POLICY (Unified Auditing)
  – ALTER AUDIT POLICY (Unified Auditing)
  – DROP AUDIT POLICY (Unified Auditing)
  – AUDIT (Unified Auditing)
  – NOAUDIT (Unified Auditing)

- A new unified key management interface for Transparent Data Encryption (TDE) eases key administration tasks, provides for better compliance and tracking, and improves separation of duty between the database administrator and security administrator.

  See the new ADMINISTER KEY MANAGEMENT statement.

ALTER DATABASE Enhancements

The following features provide enhancements to the `ALTER DATABASE` statement:

- **Storage Snapshot Optimization** enables you to use a third-party storage snapshot of the database taken without backup mode for all types of recovery operations, including point-in-time recovery. The `ALTER DATABASE` statement has been enhanced with the new `SNAPSHOT TIME` clause to enable you to recover the database using such a storage snapshot.

  See the new `SNAPSHOT TIME` clause of the `ALTER DATABASE` *full_database_recovery* clause.

- Move an online data file to a new location while the database is open and accessing the data file.

  See the new *move_datafile_clause* of `ALTER DATABASE`.

- Create a control file for a Data Guard far sync instance.

  See the enhanced *controlfile_clauses* of `ALTER DATABASE`.

- Performing switchovers and failovers to a physical standby database is simplified.

  See the new `ALTER DATABASE` clauses *switchover_clause* and *failover_clause*.

- Real-time apply is now enabled by default during Redo Apply on a physical standby database. You can disable real-time apply by specifying `USING ARCHIVED LOGFILE`.

  See the enhanced *managed_standby_recovery* clause of `ALTER DATABASE`.

ALTER SYSTEM Enhancements

The following features provide enhancements to the `ALTER SYSTEM` statement:

- Relocate a client to the least loaded Oracle ASM instance.

  See the new RELOCATE CLIENT clause of `ALTER SYSTEM`.
- Apply one-off patches released for Oracle ASM in a rolling manner.

  See the new *rolling_patch_clauses* of `ALTER SYSTEM`.

AUDIT and NOAUDIT (Traditional Auditing) Enhancements

The following feature provides enhancements to the `AUDIT` and `NOAUDIT` statements for traditional auditing:

- Audit operations on a SQL translation profile.

  See the new clause ON SQL TRANSLATION PROFILE of `AUDIT`.

CREATE DISKGROUP and ALTER DISKGROUP Enhancements

The following features provide enhancements to the `CREATE DISKGROUP` statement, `ALTER DISKGROUP` statement, or both:

- Check logical data corruptions and repair them automatically in normal and high redundancy Oracle ASM disks groups.

  See the new *scrub_clause* of `ALTER DISKGROUP`.
- Replace a user in an Oracle ASM disk group.

  See the enhanced *user_clauses* of `ALTER DISKGROUP`.
- Change the permissions, owner, and user group of an Oracle ASM disk group file while it is open.

  See the enhanced `ALTER DISKGROUP` clauses *file_permissions_clause* and the *file_owner_clause*.
- Replace one or more disks in an Oracle ASM disk group with a single operation.

  See the new *replace_disk_clause* of `ALTER DISKGROUP`.
- Rename a disk in an Oracle ASM disk group.

  See the new *rename_disk_clause* of `ALTER DISKGROUP`.
- The following are new Oracle ASM disk group attributes:
  - `CONTENT.CHECK` allows you to enable or disable content checking when performing data copy operations for rebalancing a disk group.
  - `FAILGROUP_REPAIR_TIME` allows you to specify a default repair time for the failure groups in the disk group.
  - `PHYS_META_REPLICATED` allows you to track the replication status of a disk group.
  - `THIN_PROVISIONED` allows you to enable or disable the functionality to discard unused storage space after a disk group rebalance is completed.

  See Table 13-2.

CREATE FLASHBACK ARCHIVE and ALTER FLASHBACK ARCHIVE Enhancements

The following feature provides enhancements to the CREATE FLASHBACK ARCHIVE and ALTER FLASHBACK ARCHIVE statements:

- Instruct the database to optimize the storage of data in history tables.

  See the new clause [NO] OPTIMIZE DATA of CREATE FLASHBACK ARCHIVE and the new clause [NO] OPTIMIZE DATA of ALTER FLASHBACK ARCHVE.

CREATE INDEX and ALTER INDEX Enhancements

The following features provide enhancements to the CREATE INDEX statement, ALTER INDEX statement, or both:

- Create partial indexes on a subset of the partitions of a table.

  See the new *partial_index_clause* of CREATE INDEX.
- Remove orphaned index entries for records that were previously dropped or truncated by a table partition maintenance operation.

  See the new keyword CLEANUP of ALTER INDEX ... COALESCE and the new keyword CLEANUP of ALTER INDEX ... MODIFY PARTITION ... COALESCE.
- Create multiple indexes on the same set of columns, column expressions, or both if the indexes are of different types, use different partitioning, or have different uniqueness properties.

  See the *index_expr* clause of CREATE INDEX.

CREATE INDEXTYPE and ALTER INDEXTYPE Enhancements

The following feature provides enhancements to the CREATE INDEXTYPE and ALTER INDEXTYPE statements:

- Create domain indexes on hash- and interval-partitioned tables.

  See CREATE INDEXTYPE and ALTER INDEXTYPE .

CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW Enhancements

The following feature provides enhancements to the CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW statements:

- Materialized views, which are noneditioned objects, can depend on editioned objects.

  See:
  - The new clauses *evaluation_edition_clause* and *unusable_editions_clause* of CREATE MATERIALIZED VIEW
  - The new clauses *evaluation_edition_clause* and *unusable_editions_clause* of ALTER MATERIALIZED VIEW

CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG Enhancements

The following feature provides enhancements to the `CREATE MATERIALIZED VIEW LOG` and `ALTER MATERIALIZED VIEW LOG` statements:

- Specify the refresh method for which a materialized view log will be used. You can specify synchronous refresh, which is introduced in Oracle Database 12*c*, or fast refresh, which is also available in earlier releases.

  See:

  - The new *for_refresh_clause* of `CREATE MATERIALIZED VIEW LOG`
  - The new *for_refresh_clause* of `ALTER MATERIALIZED VIEW LOG`

CREATE SEQUENCE and ALTER SEQUENCE Enhancements

The following features provide enhancements to the `CREATE SEQUENCE` and `ALTER SEQUENCE` statements:

- Control whether the sequence pseudocolumn `NEXTVAL` retains its original value during replay for Application Continuity.

  See:

  - The new clauses KEEP and NOKEEP of `CREATE SEQUENCE`
  - The new clauses `KEEP` and `NOKEEP` of ALTER SEQUENCE
- Create a session sequence, which is a special type of sequence that is specifically designed to be used with global temporary tables that have session visibility.

  See:

  - The new clauses SESSION and GLOBAL of `CREATE SEQUENCE`
  - The new clauses `SESSION` and `GLOBAL` of ALTER SEQUENCE

CREATE TABLE and ALTER TABLE Enhancements

The following features provide enhancements to the `CREATE TABLE` statement, `ALTER TABLE` statement, or both:

- The maximum size for the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types is increased to 32767 bytes.

  See "Extended Data Types".
- **Temporal Validity** support enables you to associate a valid time dimension with a table. You can use Oracle Flashback Query to retrieve rows from that table based on whether they are considered valid as of a specified time or during a specified time period.

  See:

  - The new `CREATE TABLE` clause *period_definition*
  - The new `ALTER TABLE` clauses *add_period_clause* and *drop_period_clause*
  - The enhanced `SELECT` *flashback_query_clause*
- Virtual columns, which are noneditioned objects, can depend on editioned objects.

  See:

- The new clauses evaluation_edition_clause and unusable_editions_clause of CREATE TABLE
- The new clause *modify_virtcol_properties* of ALTER TABLE

- Performance has been improved when you specify a DEFAULT value for a nullable column.

  See the DEFAULT clause of ALTER TABLE.

- Specify a default column value that includes the sequence pseudocolumns CURRVAL and NEXTVAL.

  See:

  - The DEFAULT clause of CREATE TABLE
  - The DEFAULT clause of ALTER TABLE

- The DEFAULT clause has the new clause ON NULL, which instructs the database to assign a specified default column value when an INSERT statement attempts to assign a value that evaluates to NULL.

  See:

  - The ON NULL clause of CREATE TABLE
  - The ON NULL clause of ALTER TABLE

- Specify an identity column, which is assigned an increasing or decreasing integer value from a sequence generator.

  See:

  - The new clauses *identity_clause* of CREATE TABLE and identity_options of CREATE TABLE
  - The new clauses *identity_clause* of ALTER TABLE and *identity_options* of ALTER TABLE

- Hide and unhide columns in tables.

  See:

  - The new clauses VISIBLE | INVISIBLE of CREATE TABLE
  - The new clauses VISIBLE | INVISIBLE of CREATE TABLE for virtual columns
  - The new clause *modify_col_visibility* of ALTER TABLE

- Recursively cascade a truncate operation to child tables.

  See the new keyword CASCADE of the clause *truncate_partition_subpart* of ALTER TABLE.

- Recursively cascade an exchange operation to child tables.

  See the new keyword CASCADE of the clause *exchange_partition_subpart* of ALTER TABLE.

- Store XMLType data, and abstract data types that contain attributes of type XMLType, CLOB, BLOB, or NCLOB, in an ANYDATA column.

  See the new clause *modify_opaque_type* of ALTER TABLE.

- Enable a table for row archival for In-Database Archiving.

See the new ROW ARCHIVAL clause of CREATE TABLE.

- Manage policies for Automatic Data Optimization.

  See the new *ilm_clause* of CREATE TABLE and the new *ilm_clause* of ALTER TABLE.

- Create a reference-partitioned child table whose parent is an interval-partitioned table.

  See the enhanced clause *reference_partitioning* of CREATE TABLE.

- Specify multiple table partitions or table subpartitions for the following ALTER TABLE operations:
  - Add one or more range, list, or system partitions to a table. See *add_table_partition*.
  - Add one or more range subpartitions to a partition. See *add_range_subpartition*.
  - Add one or more list subpartitions to a partition. See *add_list_subpartition*.
  - Split one range or list partition into two or more partitions. See *split_table_partition*.
  - Split one range or list subpartition into two or more subpartitions. See *split_table_subpartition*.
  - Merge two or more range, list, or system partitions into one new partition. See *merge_table_partitions* .
  - Merge two or more range or list subpartitions into one new subpartition. See *merge_table_subpartitions* .
  - Truncate one or more partitions or subpartitions. See *truncate_partition_subpart*.
  - Drop one or more partitions. See *drop_table_partition*.
  - Drop one or more subpartitions. See *drop_table_subpartition* .
- In earlier releases, the following DDL operations required a DML-blocking lock. You can use the new ONLINE keyword to allow the execution of DML statements during the following DDL operations:
  - Dropping an index (using DROP INDEX ... ONLINE ...)
  - Marking an index as UNUSABLE (using ALTER INDEX ... UNUSABLE ONLINE)
  - Marking a column as UNUSED (using ALTER TABLE ... SET UNUSED ... ONLINE ...)
  - Dropping a constraint (using ALTER TABLE ... DROP ... ONLINE ...)
  - Moving a table partition (using ALTER TABLE ... MOVE PARTITION ... ONLINE)
  - Moving a table subpartition (using ALTER TABLE ... MOVE SUBPARTITION ... ONLINE)

CREATE VIEW Enhancements

The following features provide enhancements to the CREATE VIEW statement:

- Hide and unhide columns in views.

  See the new clause VISIBLE | INVISIBLE of CREATE VIEW.

- Specify whether functions referenced in the view are executed using the view invoker's rights or the view definer's rights.

  See the new clause BEQUEATH of CREATE VIEW.

GRANT and REVOKE Enhancements

The following features provide enhancements to the GRANT and REVOKE statements:

- Grant object privileges **on a user** to users and roles.

  See:

  – The new clause ON USER of GRANT

  – The new clause ON USER of REVOKE

- Grant object privileges on a SQL translation profile to users and roles.

  See:

  – The new clause ON SQL TRANSLATION PROFILE of GRANT

  – The new clause ON SQL TRANSLATION PROFILE of REVOKE

- Grant code based access control (CBAC) roles to program units.

  See:

  – The new clause *grant_roles_to_programs* of GRANT

  – The new clause *revoke_roles_from_programs* of REVOKE

SELECT Enhancements

The following features provide enhancements to the SELECT statement:

- **Pattern matching** enables you to recognize patterns found across multiple rows in a table.

  See the new *row_pattern_clause* of SELECT.

- Perform top-N queries by specifying an offset, and the number of rows or percentage of rows to return.

  See the new *row_limiting_clause* of SELECT.

- In a query that performs outer joins of more than two pairs of tables, a single table can now be the null-generated table for multiple tables.

  See "Outer Joins ".

- Perform a variation of an ANSI CROSS JOIN or an ANSI LEFT OUTER JOIN with left correlation support. You can specify a table reference or collection expression on the right side of the join clause.

  See the new *cross_outer_apply_clause* of SELECT.

- Specify a lateral inline view in a query expression.

  See the new keyword LATERAL of SELECT.

- Declare and define PL/SQL functions and procedures in the WITH clause of a query. You can then reference the PL/SQL functions in the query and its subqueries.

  See the new clause plsql_declarations on SELECT.

TRUNCATE TABLE Enhancements

The following feature provides enhancements to the TRUNCATE TABLE statement:

- Recursively truncate child tables.

  See the new clause CASCADE of TRUNCATE TABLE.

New or Enhanced Functions

The following are new or enhanced functions:

- CLUSTER_DETAILS is a new function that predicts cluster membership for each row. It can use a pre-defined clustering model or perform dynamic clustering. The function returns an XML string that describes the predicted cluster or a specified cluster.
- CLUSTER_DISTANCE is a new function that predicts cluster membership for each row. It can use a pre-defined clustering model or perform dynamic clustering. The function returns the raw distance between each row and the centroid of either the predicted cluster or a specified.
- CLUSTER_ID has been enhanced so that it can either use a pre-defined clustering model or perform dynamic clustering.
- CLUSTER_PROBABILITY has been enhanced so that it can either use a pre-defined clustering model or perform dynamic clustering. The data type of the return value has been changed from `NUMBER` to `BINARY_DOUBLE`.
- CLUSTER_SET has been enhanced so that it can either use a pre-defined clustering model or perform dynamic clustering. The data type of the returned probability has been changed from `NUMBER` to `BINARY_DOUBLE`
- FEATURE_DETAILS is a new function that predicts feature matches for each row. It can use a pre-defined feature extraction model or perform dynamic feature extraction. The function returns an XML string that describes the predicted feature or a specified feature.
- FEATURE_ID has been enhanced so that it can either use a pre-defined feature extraction model or perform dynamic feature extraction.
- FEATURE_SET has been enhanced so that it can either use a pre-defined feature extraction model or perform dynamic feature extraction. The data type of the returned probability has been changed from `NUMBER` to `BINARY_DOUBLE`.
- FEATURE_VALUE has been enhanced so that it can either use a pre-defined feature extraction model or perform dynamic feature extraction. The data type of the return value has been changed from `NUMBER` to `BINARY_DOUBLE`.
- ORA_INVOKING_USER is a new function that returns the name of the database user who invoked the current statement or view. This function takes into account the `BEQUEATH` property of intervening views referenced in the statement.
- ORA_INVOKING_USERID is a new function that returns the identifier of the database user who invoked the current statement or view. This function takes into account the `BEQUEATH` property of intervening views referenced in the statement.
- PREDICTION has been enhanced so that it can either use a pre-defined predictive model or perform dynamic prediction.
- PREDICTION_BOUNDS now returns the upper and lower bounds of the prediction as the `BINARY_DOUBLE` data type. It previously returned these values as the `NUMBER` data type.
- PREDICTION_COST has been enhanced so that it can either use a pre-defined predictive model or perform dynamic prediction. The data type of the returned cost has been changed from `NUMBER` to `BINARY_DOUBLE`.
- PREDICTION_DETAILS has been enhanced so that it can either use a pre-defined predictive model or perform dynamic prediction.

- PREDICTION_PROBABILITY has been enhanced so that it can either use a pre-defined predictive model or perform dynamic prediction. The data type of the returned probability has been changed from NUMBER to BINARY_DOUBLE.
- PREDICTION_SET has been enhanced so that it can either use a pre-defined predictive model or perform dynamic prediction. The data type of the returned probability has been changed from NUMBER to BINARY_DOUBLE.
- STANDARD_HASH is a new function that computes a hash value for a given expression using one of several standardized hash algorithms.
- SYS_CONTEXT enables you to query a new built-in namespace, SYS_SESSION_ROLES, which allows you to determine if a specified role is currently enabled for the session.

New or Enhanced Privileges

The following are new or enhanced system privileges and object privileges:

- The behavior has changed for the following system privilege, which is listed in Table 18-1:
  - SELECT ANY DICTIONARY now does not allow you to query the following objects in the SYS schema: DEFAULT_PWD$, ENC$, LINK$, USER$, USER_HISTORY$, and XS$VERIFIERS.
- The following new system privileges are listed in Table 18-1:
  - CREATE SQL TRANSLATION PROFILE, CREATE ANY SQL TRANSLATION PROFILE, ALTER ANY SQL TRANSLATION PROFILE, USE ANY SQL TRANSLATION PROFILE, and DROP ANY SQL TRANSLATION PROFILE allow you to manage SQL translation profiles.
  - EXEMPT REDACTION POLICY allows you to bypass any existing Oracle Data Redaction policies.
  - INHERIT ANY PRIVILEGES allows you to execute invoker's rights procedures with the privileges of the invoker.
  - KEEP DATE TIME allows the SYSDATE and SYSTIMESTAMP functions to return their original values during replay for Application Continuity.
  - KEEP SYSGUID allows the SYS_GUID function to return its original value during replay for Application Continuity.
  - LOGMINING allows you to perform LogMiner operations in a multitenant container database (CDB).
  - PURGE DBA_RECYCLEBIN allows you to remove all objects from the system-wide recycle bin.
  - SYSBACKUP allows you to perform backup and recovery tasks.
  - SYSDG allows you to manage Oracle Data Guard.
  - SYSKM allows you to perform encryption key management.
  - TRANSLATE ANY SQL allows you to translate SQL for any user.
- The following new object privileges are listed in Table 18-2:
  - The ALTER and USE privileges authorize operations on SQL translation profiles.

  - INHERIT PRIVILEGES is a new type of object privilege that can be granted **on a user** to users and roles. It allows invoker's rights procedures owned by the grantee to be executed with the privileges of the invoker when the invoker is the user on whom this privilege is granted.
  - KEEP SEQUENCE allows the sequence pseudocolumn NEXTVAL to retain its original value during replay for Application Continuity.
  - TRANSLATE SQL is a new type of object privilege that can be granted **on a user** to users and roles. It allows the grantee to translate SQL through the grantee's SQL translation profile for the user on whom this privilege is granted.

New Hints

The following are new hints:

- The GATHER_OPTIMIZER_STATISTICS Hint and NO_GATHER_OPTIMIZER_STATISTICS Hint allow you to enable and disable statistics gathering during bulk loads.
- The PQ_CONCURRENT_UNION Hint and NO_PQ_CONCURRENT_UNION Hint allow you to enable and disable concurrent processing of UNION and UNION ALL operations.
- The PQ_FILTER Hint allows you to instruct the optimizer on how to process rows when filtering correlated subqueries.
- The PQ_SKEW Hint and NO_PQ_SKEW Hint allow you to advise the optimizer of whether the distribution of the values of the join keys for a parallel join is skewed.
- The USE_CUBE Hint and NO_USE_CUBE Hint allow you to specify whether to use or exclude cube joins.

Deprecated Features

The following features are deprecated in this release, and may be desupported in a future release:

- Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. SQL plan management creates SQL plan baselines, which offer superior SQL performance stability compared with stored outlines.

  See *Oracle Database SQL Tuning Guide* for more information about SQL plan management.
- The use of PKI encryption with Transparent Data Encryption is deprecated. To configure Transparent Data Encryption, use the ADMINISTER KEY MANAGEMENT statement.

  See *Oracle Database Advanced Security Guide* for more information.

Desupported Features

Some features previously described in this document are desupported in Oracle Database 12*c* Release 1 (12.1). See *Oracle Database Upgrade Guide* for a list of desupported features.

1

Introduction to Oracle SQL

Structured Query Language (SQL) is the set of statements with which all programs and users access data in an Oracle Database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications in turn must use SQL when executing the user's request. This chapter provides background information on SQL as used by most database systems.

This chapter contains these topics:

- History of SQL
- SQL Standards
- Lexical Conventions
- Tools Support

History of SQL

Dr. E. F. Codd published the paper, "A Relational Model of Data for Large Shared Data Banks", in June 1970 in the Association of Computer Machinery (ACM) journal, *Communications of the ACM*. Codd's model is now accepted as the definitive model for relational database management systems (RDBMS). The language, Structured English Query Language (SEQUEL) was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel"). In 1979, Relational Software, Inc. (now Oracle) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

SQL Standards

Oracle strives to comply with industry-accepted standards and participates actively in SQL standards committees. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), which is affiliated with the International Electrotechnical Commission (IEC). Both ANSI and the ISO/IEC have accepted SQL as the standard language for relational databases. When a new SQL standard is simultaneously published by these organizations, the names of the standards conform to conventions used by the organization, but the standards are technically identical.

> **See Also:**
>
> Oracle and Standard SQL for a detailed description of Oracle Database conformance to the SQL standard

How SQL Works

The strengths of SQL provide benefits for all types of users, including application programmers, database administrators, managers, and end users. Technically speaking, SQL is a data sublanguage. The purpose of SQL is to provide an interface to a relational database such as Oracle Database, and all SQL statements are instructions to the database. In this SQL differs from general-purpose programming languages like C and BASIC. Among the features of SQL are the following:

- It processes sets of data as groups rather than as individual units.
- It provides automatic navigation to the data.
- It uses statements that are complex and powerful individually, and that therefore stand alone. Flow-control statements, such as begin-end, if-then-else, loops, and exception condition handling, were initially not part of SQL and the SQL standard, but they can now be found in ISO/IEC 9075-4 - Persistent Stored Modules (SQL/PSM). The PL/SQL extension to Oracle SQL is similar to PSM.

SQL lets you work with data at the logical level. You need to be concerned with the implementation details only when you want to manipulate the data. For example, to retrieve a set of rows from a table, you define a condition used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. You need not deal with the rows one by one, nor do you have to worry about how they are physically stored or retrieved. All SQL statements use the **optimizer**, a part of Oracle Database that determines the most efficient means of accessing the specified data. Oracle also provides techniques that you can use to make the optimizer perform its job better.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the preceding tasks in one consistent language.

Common Language for All Relational Databases

All major relational database management systems support SQL, so you can transfer all skills you have gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

Using Enterprise Manager

Many of the operations you can accomplish using SQL syntax can be done much more easily using Enterprise Manager. For more information, see the Oracle Enterprise Manager documentation set, *Oracle Database 2 Day DBA*, or any of the Oracle Database *2 Day* + books.

Lexical Conventions

The following lexical conventions for issuing SQL statements apply specifically to the Oracle Database implementation of SQL, but are generally acceptable in other SQL implementations.

When you issue a SQL statement, you can include one or more tabs, carriage returns, spaces, or comments anywhere a space occurs within the definition of the statement. Thus, Oracle Database evaluates the following two statements in the same manner:

```
SELECT last_name,salary*12,MONTHS_BETWEEN(SYSDATE,hire_date)
  FROM employees
  WHERE department_id = 30
  ORDER BY last_name;

SELECT last_name,
  salary * 12,
        MONTHS_BETWEEN( SYSDATE, hire_date )
FROM employees
WHERE department_id=30
ORDER BY last_name;
```

Case is insignificant in reserved words, keywords, identifiers, and parameters. However, case is significant in text literals and quoted names. Refer to Text Literals for a syntax description of text literals.

> **Note:**
>
> SQL statements are terminated differently in different programming environments. This documentation set uses the default SQL*Plus character, the semicolon (;).

Tools Support

Oracle provides a number of utilities to facilitate your SQL development process:

- Oracle SQL Developer is a graphical tool that lets you browse, create, edit, and delete (drop) database objects, edit and debug PL/SQL code, run SQL statements and scripts, manipulate and export data, and create and view reports. With SQL Developer, you can connect to any target Oracle Database schema using standard Oracle Database authentication. Once connected, you can perform operations on objects in the database. You can also connect to schemas for selected third-party (non-Oracle) databases, such as MySQL, Microsoft SQL Server, and Microsoft Access, view metadata and data in these databases, and migrate these databases to Oracle.
- SQL*Plus is an interactive and batch query tool that is installed with every Oracle Database server or client installation. It has a command-line user interface and a Web-based user interface called *i*SQL*Plus.
- Oracle JDeveloper is a multiple-platform integrated development environment supporting the complete lifecycle of development for Java, Web services, and SQL. It provides a graphical interface for executing and tuning SQL statements and a visual schema diagrammer (database modeler). It also supports editing, compiling, and debugging PL/SQL applications.

- Oracle Application Express is a hosted environment for developing and deploying database-related Web applications. SQL Workshop is a component of Oracle Application Express that lets you view and manage database objects from a Web browser. SQL Workshop offers quick access to a SQL command processor and a SQL script repository.

> **See Also:**
>
> *SQL*Plus User's Guide and Reference* and *Oracle Application Express App Builder User's Guide* for more information on these products

The Oracle Call Interface and Oracle precompilers let you embed standard SQL statements within a procedure programming language.

- The Oracle Call Interface (OCI) lets you embed SQL statements in C programs.
- The Oracle precompilers, Pro*C/C++ and Pro*COBOL, interpret embedded SQL statements and translate them into statements that can be understood by C/C++ and COBOL compilers, respectively.

> **See Also:**
>
> *Oracle C++ Call Interface Programmer's Guide*, *Pro*COBOL Programmer's Guide*, and *Oracle Call Interface Programmer's Guide* for additional information on the embedded SQL statements allowed in each product

Most (but not all) Oracle tools also support all features of Oracle SQL. This reference describes the complete functionality of SQL. If the Oracle tool that you are using does not support this complete functionality, then you can find a discussion of the restrictions in the manual describing the tool, such as *SQL*Plus User's Guide and Reference*.

2 Basic Elements of Oracle SQL

This chapter contains reference information on the basic elements of Oracle SQL. These elements are the simplest building blocks of SQL statements. Therefore, before using the SQL statements described in this book, you should familiarize yourself with the concepts covered in this chapter.

This chapter contains these sections:

- Data Types
- Data Type Comparison Rules
- Literals
- Format Models
- Nulls
- Comments
- Database Objects
- Database Object Names and Qualifiers
- Syntax for Schema Objects and Parts in SQL Statements

Data Types

Each value manipulated by Oracle Database has a **data type**. The data type of a value associates a fixed set of properties with the value. These properties cause Oracle to treat values of one data type differently from values of another. For example, you can add values of NUMBER data type, but not values of RAW data type.

When you create a table or cluster, you must specify a data type for each of its columns. When you create a procedure or stored function, you must specify a data type for each of its arguments. These data types define the domain of values that each column can contain or each argument can have. For example, DATE columns cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'. Each value subsequently placed in a column assumes the data type of the column. For example, if you insert '01-JAN-98' into a DATE column, then Oracle treats the '01-JAN-98' character string as a DATE value after verifying that it translates to a valid date.

Oracle Database provides a number of built-in data types as well as several categories for user-defined types that can be used as data types. The syntax of Oracle data types appears in the diagrams that follow. The text of this section is divided into the following sections:

- Oracle Built-in Data Types
- ANSI, DB2, and SQL/DS Data Types
- User-Defined Types
- Oracle-Supplied Types
- Data Type Comparison Rules

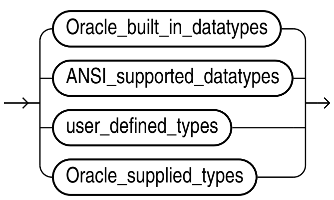
- Data Conversion

A data type is either scalar or nonscalar. A scalar type contains an atomic value, whereas a nonscalar (sometimes called a "collection") contains a set of values. A large object (LOB) is a special form of scalar data type representing a large scalar value of binary or character data. LOBs are subject to some restrictions that do not affect other scalar types because of their size. Those restrictions are documented in the context of the relevant SQL syntax.

> **See Also:**
>
> Restrictions on LOB Columns

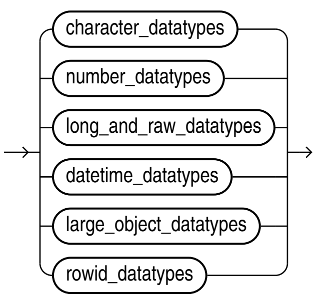
The Oracle precompilers recognize other data types in embedded SQL programs. These data types are called **external data types** and are associated with host variables. Do not confuse built-in data types and user-defined types with external data types. For information on external data types, including how Oracle converts between them and built-in data types or user-defined types, see *Pro*COBOL Programmer's Guide*, and *Pro*C/C++ Programmer's Guide*.

***datatypes*::=**

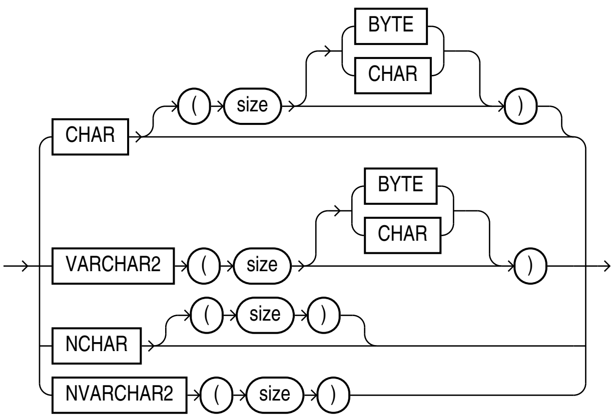


The Oracle built-in data types appear in the figures that follows. For descriptions, refer to Oracle Built-in Data Types.

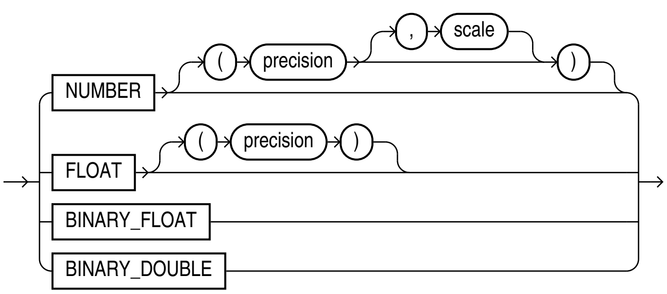
***Oracle_built_in_datatypes*::=**



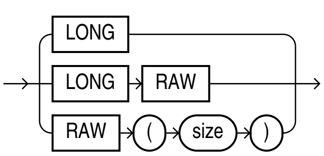
***character_datatypes*::=**



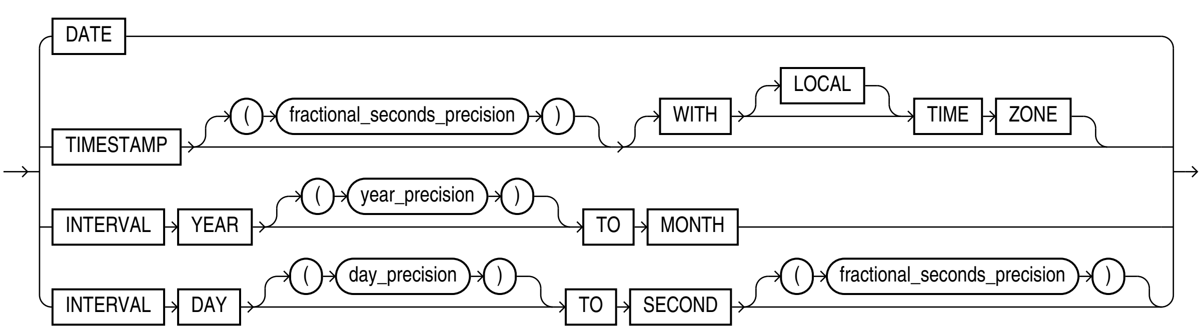
***number_datatypes*::=**



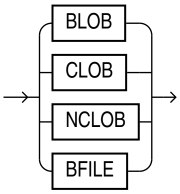
***long_and_raw_datatypes*::=**



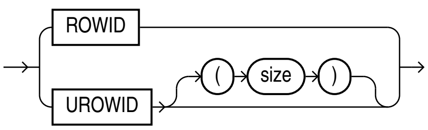
***datetime_datatypes*::=**



***large_object_datatypes*::=**

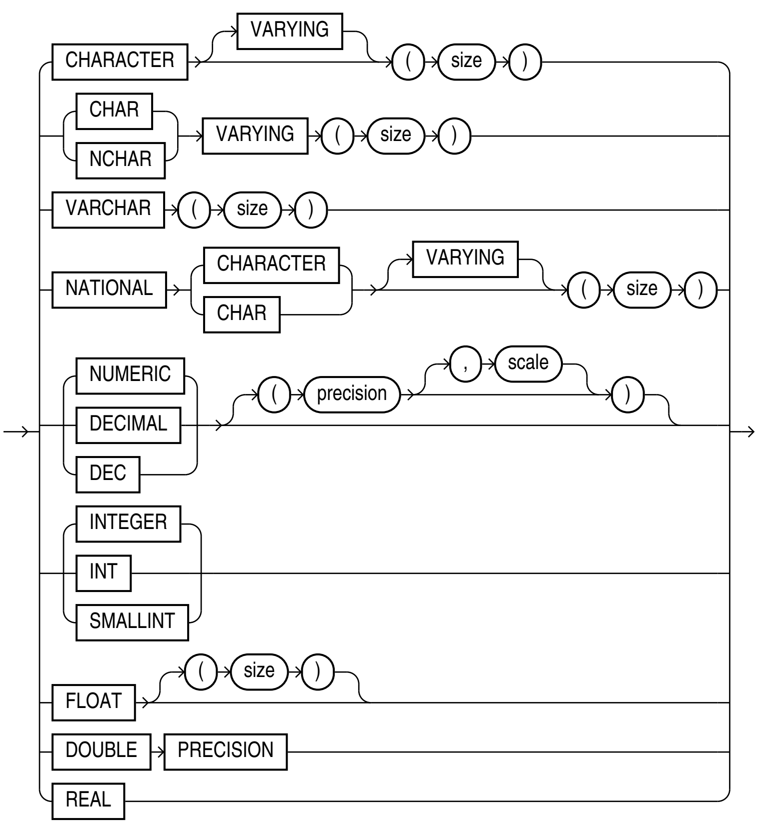


***rowid_datatypes*::=**



The ANSI-supported data types appear in the figure that follows. ANSI, DB2, and SQL/DS Data Types discusses the mapping of ANSI-supported data types to Oracle built-in data types.

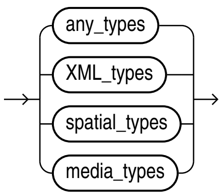
***ANSI_supported_datatypes*::=**



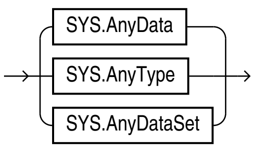
For descriptions of user-defined types, refer to User-Defined Types .

The Oracle-supplied data types appear in the figures that follows. For descriptions, refer to Oracle-Supplied Types .

***Oracle_supplied_types*::=**

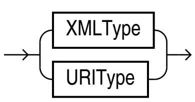


***any_types*::=**



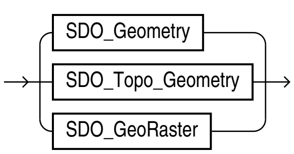
For descriptions of the `Any` types, refer to Any Types .

***XML_types*::=**



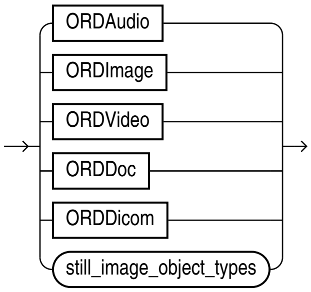
For descriptions of the XML types, refer to XML Types .

***spatial_types*::=**

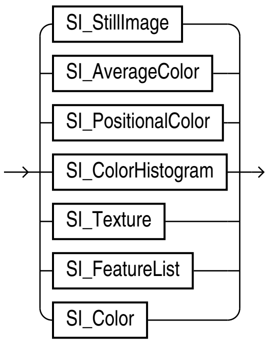


For descriptions of the spatial types, refer to Spatial Types .

***media_types*::=**



***still_image_object_types*::=**



For descriptions of the media types, refer to Media Types .

Oracle Built-in Data Types

The **Built-In Data Type Summary** table lists the built-in data types available. Oracle Database uses a code to identify the data type internally. This is the number in the **Code** column of the **Built-In Data Type Summary** table. You can verify the codes in the table using the `DUMP` function.

In addition to the built-in data types listed in the **Built-In Data Type Summary** table, Oracle Database uses many data types internally that are visible via the `DUMP` function.

Table 2-1 Built-In Data Type Summary

Code	Data Type	Description
1	VARCHAR2(*size* [BYTE | CHAR])	Variable-length character string having maximum length *size* bytes or characters. You must specify *size* for VARCHAR2. Minimum *size* is 1 byte or 1 character. Maximum size is: • 32767 bytes or characters if MAX_STRING_SIZE = EXTENDED • 4000 bytes or characters if MAX_STRING_SIZE = STANDARD Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter. BYTE indicates that the column will have byte length semantics. CHAR indicates that the column will have character semantics.
1	NVARCHAR2(*size*)	Variable-length Unicode character string having maximum length *size* characters. You must specify *size* for NVARCHAR2. The number of bytes can be up to two times *size* for AL16UTF16 encoding and three times *size* for UTF8 encoding. Maximum *size* is determined by the national character set definition, with an upper limit of: • 32767 bytes if MAX_STRING_SIZE = EXTENDED • 4000 bytes if MAX_STRING_SIZE = STANDARD Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter.
2	NUMBER [(*p* [, *s*])]	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127. Both precision and scale are in decimal digits. A NUMBER value requires from 1 to 22 bytes.
2	FLOAT [(*p*)]	A subtype of the NUMBER data type having precision *p*. A FLOAT value is represented internally as NUMBER. The precision *p* can range from 1 to 126 binary digits. A FLOAT value requires from 1 to 22 bytes.
8	LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes. Provided for backward compatibility.
12	DATE	Valid date range from January 1, 4712 BC, to December 31, 9999 AD. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is fixed at 7 bytes. This data type contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It does not have fractional seconds or a time zone.
100	BINARY_FLOAT	32-bit floating point number. This data type requires 4 bytes.
101	BINARY_DOUBLE	64-bit floating point number. This data type requires 8 bytes.

Table 2-1 (Cont.) Built-In Data Type Summary

Code	Data Type	Description
180	TIMESTAMP [(*fractional_seconds_precision*)]	Year, month, and day values of date, as well as hour, minute, and second values of time, where *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values of *fractional_seconds_precision* are 0 to 9. The default is 6. The default format is determined explicitly by the NLS_TIMESTAMP_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is 7 or 11 bytes, depending on the precision. This data type contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It contains fractional seconds but does not have a time zone.
181	TIMESTAMP [(*fractional_seconds_precision*)] WITH TIME ZONE	All values of TIMESTAMP as well as time zone displacement value, where *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6. The default date format for the TIMESTAMP WITH TIME ZONE data type is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. The size is fixed at 13 bytes. This data type contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, and TIMEZONE_MINUTE. It has fractional seconds and an explicit time zone.
231	TIMESTAMP [(*fractional_seconds_precision*)] WITH LOCAL TIME ZONE	All values of TIMESTAMP WITH TIME ZONE, with the following exceptions: • Data is normalized to the database time zone when it is stored in the database. • When the data is retrieved, users see the data in the session time zone. The default format is determined explicitly by the NLS_TIMESTAMP_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is 7 or 11 bytes, depending on the precision.
182	INTERVAL YEAR [(*year_precision*)] TO MONTH	Stores a period of time in years and months, where *year_precision* is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 2. The size is fixed at 5 bytes.
183	INTERVAL DAY [(*day_precision*)] TO SECOND [(*fractional_seconds_precision*)]	Stores a period of time in days, hours, minutes, and seconds, where • *day_precision* is the maximum number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2. • *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6. The size is fixed at 11 bytes.
23	RAW(*size*)	Raw binary data of length *size* bytes. You must specify *size* for a RAW value. Maximum *size* is: • 32767 bytes if MAX_STRING_SIZE = EXTENDED • 2000 bytes if MAX_STRING_SIZE = STANDARD Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter.

Table 2-1 (Cont.) Built-In Data Type Summary

Code	Data Type	Description
24	LONG RAW	Raw binary data of variable length up to 2 gigabytes.
69	ROWID	Base 64 string representing the unique address of a row in its table. This data type is primarily for values returned by the ROWID pseudocolumn.
208	UROWID [(*size*)]	Base 64 string representing the logical address of a row of an index-organized table. The optional *size* is the size of a column of type UROWID. The maximum size and default is 4000 bytes.
96	CHAR [(*size* [BYTE | CHAR])]	Fixed-length character data of length *size* bytes or characters. Maximum *size* is 2000 bytes or characters. Default and minimum *size* is 1 byte. BYTE and CHAR have the same semantics as for VARCHAR2.
96	NCHAR[(*size*)]	Fixed-length character data of length *size* characters. The number of bytes can be up to two times *size* for AL16UTF16 encoding and three times *size* for UTF8 encoding. Maximum *size* is determined by the national character set definition, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character.
112	CLOB	A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) * (database block size).
112	NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) * (database block size). Stores national character set data.
113	BLOB	A binary large object. Maximum size is (4 gigabytes - 1) * (database block size).
114	BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.

The sections that follow describe the Oracle data types as they are stored in Oracle Database. For information on specifying these data types as literals, refer to Literals .

Character Data Types

Character data types store character (alphanumeric) data, which are words and free-form text, in the database character set or national character set. They are less restrictive than other data types and consequently have fewer properties. For example, character columns can store all alphanumeric values, but NUMBER columns can store only numeric values.

Character data is stored in strings with byte values corresponding to one of the character sets, such as 7-bit ASCII or EBCDIC, specified when the database was created. Oracle Database supports both single-byte and multibyte character sets.

These data types are used for character data:

- CHAR Data Type

- NCHAR Data Type
- VARCHAR2 Data Type
- NVARCHAR2 Data Type

For information on specifying character data types as literals, refer to Text Literals .

CHAR Data Type

The CHAR data type specifies a fixed-length character string in the database character set. You specify the database character set when you create your database.

When you create a table with a CHAR column, you specify the column length as *size* optionally followed by a length qualifier. The qualifier BYTE denotes byte length semantics while the qualifier CHAR denotes character length semantics. In the byte length semantics, *size* is the number of bytes to store in the column. In the character length semantics, *size* is the number of code points in the database character set to store in the column. A code point may have from 1 to 4 bytes depending on the database character set and the particular character encoded by the code point. Oracle recommends that you specify one of the length qualifiers to explicitly document the desired length semantics of the column. If you do not specify a qualifier, the value of the NLS_LENGTH_SEMANTICS parameter of the session creating the column defines the length semantics, unless the table belongs to the schema SYS, in which case the default semantics is BYTE.

Oracle ensures that all values stored in a CHAR column have the length specified by *size* in the selected length semantics. If you insert a value that is shorter than the column length, then Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, then Oracle returns an error. Note that if the column length is expressed in characters (code points), blank-padding does not guarantee that all column values have the same byte length.

You can omit *size* from the column definition. The default value is 1.

The maximum value of *size* is 2000, which means 2000 bytes or characters (code points), depending on the selected length semantics. However, independently, the absolute maximum length of any character value that can be stored into a CHAR column is 2000 bytes. For example, even if you define the column length to be 2000 characters, Oracle returns an error if you try to insert a 2000-character value in which one or more code points are wider than 1 byte. The value of *size* in characters is a length constraint, not guaranteed capacity. If you want a CHAR column to be always able to store *size* characters in any database character set, use a value of *size* that is less than or equal to 500.

To ensure proper data conversion between databases and clients with different character sets, you must ensure that CHAR data consists of well-formed strings.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on character set support and Data Type Comparison Rules for information on comparison semantics

NCHAR Data Type

The NCHAR data type specifies a fixed-length character string in the national character set. You specify the national character set as either AL16UTF16 or UTF8 when you create your database. AL16UTF16 and UTF8 are two encoding forms of the Unicode character set (UTF-16 and CESU-8, correspondingly) and hence NCHAR is a Unicode-only data type.

When you create a table with an NCHAR column, you specify the column length as *size* characters, or more precisely, code points in the national character set. One code point has always 2 bytes in AL16UTF16 and from 1 to 3 bytes in UTF8, depending on the particular character encoded by the code point.

Oracle ensures that all values stored in an NCHAR column have the length of *size* characters. If you insert a value that is shorter than the column length, then Oracle blank-pads the value to the column length. If you try to insert a value that is too long for the column, then Oracle returns an error. Note that if the national character set is UTF8, blank-padding does not guarantee that all column values have the same byte length.

You can omit *size* from the column definition. The default value is 1.

The maximum value of *size* is 1000 characters when the national character set is AL16UTF16, and 2000 characters when the national character set is UTF8. However, independently, the absolute maximum length of any character value that can be stored into an NCHAR column is 2000 bytes. For example, even if you define the column length to be 1000 characters, Oracle returns an error if you try to insert a 1000-character value but the national character set is UTF8 and all code points are 3 bytes wide. The value of *size* is a length constraint, not guaranteed capacity. If you want an NCHAR column to be always able to store *size* characters in both national character sets, use a value of *size* that is less than or equal to 666.

To ensure proper data conversion between databases and clients with different character sets, you must ensure that NCHAR data consists of well-formed strings.

If you assign a CHAR value to an NCHAR column, the value is implicitly converted from the database character set to the national character set. If you assign an NCHAR value to a CHAR column, the value is implicitly converted from the national character set to the database character set. If some of the characters from the NCHAR value cannot be represented in the database character set, then if the value of the session parameter NLS_NCHAR_CONV_EXCP is TRUE, then Oracle reports an error. If the value of the parameter is FALSE, non-representable characters are replaced with the default replacement character of the database character set, which is usually the question mark '?' or the inverted question mark '¿'.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support

VARCHAR2 Data Type

The VARCHAR2 data type specifies a variable-length character string in the database character set. You specify the database character set when you create your database.

When you create a table with a VARCHAR2 column, you must specify the column length as *size* optionally followed by a length qualifier. The qualifier BYTE denotes byte length semantics while the qualifier CHAR denotes character length semantics. In the byte length semantics, *size* is the maximum number of bytes that can be stored in the column. In the character length semantics, *size* is the maximum number of code points in the database character set that can be stored in the column. A code point may have from 1 to 4 bytes depending on the database character set and the particular character encoded by the code point. Oracle recommends that you specify one of the length qualifiers to explicitly document the desired length semantics of the column. If you do not specify a qualifier, the value of the NLS_LENGTH_SEMANTICS parameter of the session creating the column defines the length semantics, unless the table belongs to the schema SYS, in which case the default semantics is BYTE.

Oracle stores a character value in a VARCHAR2 column exactly as you specify it, without any blank-padding, provided the value does not exceed the length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

The minimum value of *size* is 1. The maximum value is:

- 32767 bytes if MAX_STRING_SIZE = EXTENDED
- 4000 bytes if MAX_STRING_SIZE = STANDARD

Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter and the internal storage mechanisms for extended data types.

While *size* may be expressed in bytes or characters (code points) the independent absolute maximum length of any character value that can be stored into a VARCHAR2 column is 32767 or 4000 bytes, depending on MAX_STRING_SIZE. For example, even if you define the column length to be 32767 characters, Oracle returns an error if you try to insert a 32767-character value in which one or more code points are wider than 1 byte. The value of *size* in characters is a length constraint, not guaranteed capacity. If you want a VARCHAR2 column to be always able to store *size* characters in any database character set, use a value of *size* that is less than or equal to 8191, if MAX_STRING_SIZE = EXTENDED, or 1000, if MAX_STRING_SIZE = STANDARD.

Oracle compares VARCHAR2 values using non-padded comparison semantics.

To ensure proper data conversion between databases with different character sets, you must ensure that VARCHAR2 data consists of well-formed strings. See *Oracle Database Globalization Support Guide* for more information on character set support.

> **See Also:**
>
> Data Type Comparison Rules for information on comparison semantics

VARCHAR Data Type

Do not use the VARCHAR data type. Use the VARCHAR2 data type instead. Although the VARCHAR data type is currently synonymous with VARCHAR2, the VARCHAR data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

NVARCHAR2 Data Type

The NVARCHAR2 data type specifies a variable-length character string in the national character set. You specify the national character set as either AL16UTF16 or UTF8 when you create your database. AL16UTF16 and UTF8 are two encoding forms of the Unicode character set (UTF-16 and CESU-8, correspondingly) and hence NVARCHAR2 is a Unicode-only data type.

When you create a table with an NVARCHAR2 column, you must specify the column length as *size* characters, or more precisely, code points in the national character set. One code point has always 2 bytes in AL16UTF16 and from 1 to 3 bytes in UTF8, depending on the particular character encoded by the code point.

Oracle stores a character value in an NVARCHAR2 column exactly as you specify it, without any blank-padding, provided the value does not exceed the length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

The minimum value of *size* is 1. The maximum value is:

- 16383 if MAX_STRING_SIZE = EXTENDED and the national character set is AL16UTF16
- 32767 if MAX_STRING_SIZE = EXTENDED and the national character set is UTF8
- 2000 if MAX_STRING_SIZE = STANDARD and the national character set is AL16UTF16
- 4000 if MAX_STRING_SIZE = STANDARD and the national character set is UTF8

Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter and the internal storage mechanisms for extended data types.

Independently of the maximum column length in characters, the absolute maximum length of any value that can be stored into an NVARCHAR2 column is 32767 or 4000 bytes, depending on MAX_STRING_SIZE. For example, even if you define the column length to be 16383 characters, Oracle returns an error if you try to insert a 16383-character value but the national character set is UTF8 and all code points are 3 bytes wide. The value of *size* is a length constraint, not guaranteed capacity. If you want an NVARCHAR2 column to be always able to store *size* characters in both national character sets, use a value of *size* that is less than or equal to 10922, if MAX_STRING_SIZE = EXTENDED, or 1333, if MAX_STRING_SIZE = STANDARD.

Oracle compares NVARCHAR2 values using non-padded comparison semantics.

To ensure proper data conversion between databases and clients with different character sets, you must ensure that NVARCHAR2 data consists of well-formed strings.

If you assign a VARCHAR2 value to an NVARCHAR2 column, the value is implicitly converted from the database character set to the national character set. If you assign an NVARCHAR2 value to a VARCHAR2 column, the value is implicitly converted from the national character set to the database character set. If some of the characters from the NVARCHAR2 value cannot be represented in the database character set, then if the value of the session parameter NLS_NCHAR_CONV_EXCP is TRUE, then Oracle reports an error. If the value of the parameter is FALSE, non-representable characters are replaced with the default replacement character of the database character set, which is usually the question mark '?' or the inverted question mark '¿'.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support.

Numeric Data Types

The Oracle Database numeric data types store positive and negative fixed and floating-point numbers, zero, infinity, and values that are the undefined result of an operation—"not a number" or `NAN`. For information on specifying numeric data types as literals, refer to Numeric Literals .

NUMBER Data Type

The `NUMBER` data type stores zero as well as positive and negative fixed numbers with absolute values from 1.0 x 10^{-130} to but not including 1.0 x 10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0 x 10^{126}, then Oracle returns an error. Each `NUMBER` value requires from 1 to 22 bytes.

Specify a fixed-point number using the following form:

```
NUMBER(p,s)
```

where:

- *p* is the **precision**, or the maximum number of significant decimal digits, where the most significant digit is the left-most nonzero digit, and the least significant digit is the right-most known digit. Oracle guarantees the portability of numbers with precision of up to 20 base-100 digits, which is equivalent to 39 or 40 decimal digits depending on the position of the decimal point.
- *s* is the **scale**, or the number of digits from the decimal point to the least significant digit. The scale can range from -84 to 127.
  - Positive scale is the number of significant digits to the right of the decimal point to and including the least significant digit.
  - Negative scale is the number of significant digits to the left of the decimal point, to but not including the least significant digit. For negative scale the least significant digit is on the left side of the decimal point, because the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

Scale can be greater than precision, most commonly when `e` notation is used. When scale is greater than precision, the precision specifies the maximum number of significant digits to the right of the decimal point. For example, a column defined as `NUMBER(4,5)` requires a zero for the first digit after the decimal point and rounds all values past the fifth digit after the decimal point.

It is good practice to specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed length. If a value exceeds the precision, then Oracle returns an error. If a value exceeds the scale, then Oracle rounds it.

Specify an integer using the following form:

```
NUMBER(p)
```

This represents a fixed-point number with precision *p* and scale 0 and is equivalent to `NUMBER(p,0)`.

Specify a floating-point number using the following form:

```
NUMBER
```

The absence of precision and scale designators specifies the maximum range and precision for an Oracle number.

> **See Also:**
>
> Floating-Point Numbers

Table 2-2 show how Oracle stores data using different precisions and scales.

Table 2-2 Storage of Scale and Precision

Actual Data	Specified As	Stored As
123.89	`NUMBER`	123.89
123.89	`NUMBER(3)`	124
123.89	`NUMBER(3,2)`	exceeds precision
123.89	`NUMBER(4,2)`	exceeds precision
123.89	`NUMBER(5,2)`	123.89
123.89	`NUMBER(6,1)`	123.9
123.89	`NUMBER(6,-2)`	100
.01234	`NUMBER(4,5)`	.01234
.00012	`NUMBER(4,5)`	.00012
.000127	`NUMBER(4,5)`	.00013
.0000012	`NUMBER(2,7)`	.0000012
.00000123	`NUMBER(2,7)`	.0000012
1.2e-4	`NUMBER(2,5)`	0.00012
1.2e-5	`NUMBER(2,5)`	0.00001

FLOAT Data Type

The `FLOAT` data type is a subtype of `NUMBER`. It can be specified with or without precision, which has the same definition it has for `NUMBER` and can range from 1 to 126. Scale cannot be specified, but is interpreted from the data. Each `FLOAT` value requires from 1 to 22 bytes.

To convert from binary to decimal precision, multiply *n* by 0.30103. To convert from decimal to binary precision, multiply the decimal precision by 3.32193. The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

The difference between NUMBER and FLOAT is best illustrated by example. In the following example the same values are inserted into NUMBER and FLOAT columns:

```
CREATE TABLE test (col1 NUMBER(5,2), col2 FLOAT(5));

INSERT INTO test VALUES (1.23, 1.23);
INSERT INTO test VALUES (7.89, 7.89);
INSERT INTO test VALUES (12.79, 12.79);
INSERT INTO test VALUES (123.45, 123.45);

SELECT * FROM test;

      COL1       COL2
---------- ----------
      1.23        1.2
      7.89        7.9
     12.79         13
    123.45        120
```

In this example, the FLOAT value returned cannot exceed 5 binary digits. The largest decimal number that can be represented by 5 binary digits is 31. The last row contains decimal values that exceed 31. Therefore, the FLOAT value must be truncated so that its significant digits do not require more than 5 binary digits. Thus 123.45 is rounded to 120, which has only two significant decimal digits, requiring only 4 binary digits.

Oracle Database uses the Oracle FLOAT data type internally when converting ANSI FLOAT data. Oracle FLOAT is available for you to use, but Oracle recommends that you use the BINARY_FLOAT and BINARY_DOUBLE data types instead, as they are more robust. Refer to Floating-Point Numbers for more information.

Floating-Point Numbers

Floating-point numbers can have a decimal point anywhere from the first to the last digit or can have no decimal point at all. An exponent may optionally be used following the number to increase the range, for example, 1.777 e^{-20}. A scale value is not applicable to floating-point numbers, because the number of digits that can appear after the decimal point is not restricted.

Binary floating-point numbers differ from NUMBER in the way the values are stored internally by Oracle Database. Values are stored using decimal precision for NUMBER. All literals that are within the range and precision supported by NUMBER are stored exactly as NUMBER. Literals are stored exactly because literals are expressed using decimal precision (the digits 0 through 9). Binary floating-point numbers are stored using binary precision (the digits 0 and 1). Such a storage scheme cannot represent all values using decimal precision exactly. Frequently, the error that occurs when converting a value from decimal to binary precision is undone when the value is converted back from binary to decimal precision. The literal 0.1 is such an example.

Oracle Database provides two numeric data types exclusively for floating-point numbers:

BINARY_FLOAT

BINARY_FLOAT is a 32-bit, single-precision floating-point number data type. Each BINARY_FLOAT value requires 4 bytes.

BINARY_DOUBLE

BINARY_DOUBLE is a 64-bit, double-precision floating-point number data type. Each BINARY_DOUBLE value requires 8 bytes.

In a NUMBER column, floating point numbers have decimal precision. In a BINARY_FLOAT or BINARY_DOUBLE column, floating-point numbers have binary precision. The binary floating-point numbers support the special values infinity and NaN (not a number).

You can specify floating-point numbers within the limits listed in Table 2-3. The format for specifying floating-point numbers is defined in Numeric Literals .

Table 2-3 Floating Point Number Limits

Value	BINARY_FLOAT	BINARY_DOUBLE
Maximum positive finite value	3.40282E+38F	1.79769313486231E+308
Minimum positive finite value	1.17549E-38F	2.22507485850720E-308

IEEE754 Conformance

The Oracle implementation of floating-point data types conforms substantially with the Institute of Electrical and Electronics Engineers (IEEE) Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985 (IEEE754). The floating-point data types conform to IEEE754 in the following areas:

- The SQL function SQRT implements square root. See SQRT .
- The SQL function REMAINDER implements remainder. See REMAINDER .
- Arithmetic operators conform. See Arithmetic Operators .
- Comparison operators conform, except for comparisons with NaN. Oracle orders NaN greatest with respect to all other values, and evaluates NaN equal to NaN. See Floating-Point Conditions .
- Conversion operators conform. See Conversion Functions .
- The default rounding mode is supported.
- The default exception handling mode is supported.
- The special values INF, -INF, and NaN are supported. See Floating-Point Conditions .
- Rounding of BINARY_FLOAT and BINARY_DOUBLE values to integer-valued BINARY_FLOAT and BINARY_DOUBLE values is provided by the SQL functions ROUND, TRUNC, CEIL, and FLOOR.
- Rounding of BINARY_FLOAT/BINARY_DOUBLE to decimal and decimal to BINARY_FLOAT/BINARY_DOUBLE is provided by the SQL functions TO_CHAR, TO_NUMBER, TO_NCHAR, TO_BINARY_FLOAT, TO_BINARY_DOUBLE, and CAST.

The floating-point data types do not conform to IEEE754 in the following areas:

- -0 is coerced to +0.
- Comparison with NaN is not supported.
- All NaN values are coerced to either BINARY_FLOAT_NAN or BINARY_DOUBLE_NAN.

- Non-default rounding modes are not supported.
- Non-default exception handling mode are not supported.

Numeric Precedence

Numeric precedence determines, for operations that support numeric data types, the data type Oracle uses if the arguments to the operation have different data types. `BINARY_DOUBLE` has the highest numeric precedence, followed by `BINARY_FLOAT`, and finally by `NUMBER`. Therefore, in any operation on multiple numeric values:

- If any of the operands is `BINARY_DOUBLE`, then Oracle attempts to convert all the operands implicitly to `BINARY_DOUBLE` before performing the operation.
- If none of the operands is `BINARY_DOUBLE` but any of the operands is `BINARY_FLOAT`, then Oracle attempts to convert all the operands implicitly to `BINARY_FLOAT` before performing the operation.
- Otherwise, Oracle attempts to convert all the operands to `NUMBER` before performing the operation.

If any implicit conversion is needed and fails, then the operation fails. Refer to Table 2-8 for more information on implicit conversion.

In the context of other data types, numeric data types have lower precedence than the datetime/interval data types and higher precedence than character and all other data types.

LONG Data Type

Do not create tables with `LONG` columns. Use LOB columns (`CLOB`, `NCLOB`, `BLOB`) instead. `LONG` columns are supported only for backward compatibility.

`LONG` columns store variable-length character strings containing up to 2 gigabytes -1, or 2^{31}-1 bytes. `LONG` columns have many of the characteristics of `VARCHAR2` columns. You can use `LONG` columns to store long text strings. The length of `LONG` values may be limited by the memory available on your computer. `LONG` literals are formed as described for Text Literals .

Oracle also recommends that you convert existing `LONG` columns to LOB columns. LOB columns are subject to far fewer restrictions than `LONG` columns. Further, LOB functionality is enhanced in every release, whereas `LONG` functionality has been static for several releases. See the *`modify_col_properties`* clause of ALTER TABLE and TO_LOB for more information on converting `LONG` columns to LOB.

You can reference `LONG` columns in SQL statements in these places:

- `SELECT` lists
- `SET` clauses of `UPDATE` statements
- `VALUES` clauses of `INSERT` statements

The use of `LONG` values is subject to these restrictions:

- A table can contain only one `LONG` column.
- You cannot create an object type with a `LONG` attribute.

- LONG columns cannot appear in WHERE clauses or in integrity constraints (except that they can appear in NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- LONG data cannot be specified in regular expressions.
- A stored function cannot return a LONG value.
- You can declare a variable or argument of a PL/SQL program unit using the LONG data type. However, you cannot then call the program unit from SQL.
- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.
- LONG and LONG RAW columns cannot be used in distributed SQL statements and cannot be replicated.
- If a table has both LONG and LOB columns, then you cannot bind more than 4000 bytes of data to both the LONG and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the LONG or the LOB column.

In addition, LONG columns cannot appear in these parts of SQL statements:

- GROUP BY clauses, ORDER BY clauses, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- The UNIQUE operator of a SELECT statement
- The column list of a CREATE CLUSTER statement
- The CLUSTER clause of a CREATE MATERIALIZED VIEW statement
- SQL built-in functions, expressions, or conditions
- SELECT lists of queries containing GROUP BY clauses
- SELECT lists of subqueries or queries combined by the UNION, INTERSECT, or MINUS set operators
- SELECT lists of CREATE TABLE ... AS SELECT statements
- ALTER TABLE ... MOVE statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG data type in the following manner:

- A SQL statement within a trigger can insert data into a LONG column.
- If data from a LONG column can be converted to a constrained data type (such as CHAR and VARCHAR2), then a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG data type.
- :NEW and :OLD cannot be used with LONG columns.

You can use Oracle Call Interface functions to retrieve a portion of a LONG value from the database.

> **See Also:**
>
> *Oracle Call Interface Programmer's Guide*

Datetime and Interval Data Types

The datetime data types are `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`. Values of datetime data types are sometimes called **datetimes**. The interval data types are `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`. Values of interval data types are sometimes called **intervals**. For information on expressing datetime and interval values as literals, refer to Datetime Literals and Interval Literals.

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the data type. Table 2-4 lists the datetime fields and their possible values for datetimes and intervals.

To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions `DBTIMEZONE` and `SESSIONTIMEZONE`. If the time zones have not been set manually, then Oracle Database uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, then Oracle uses UTC as the default value.

Table 2-4 Datetime Fields and Values

Datetime Field	Valid Values for Datetime	Valid Values for INTERVAL
`YEAR`	-4712 to 9999 (excluding year 0)	Any positive or negative integer
`MONTH`	01 to 12	0 to 11
`DAY`	01 to 31 (limited by the values of `MONTH` and `YEAR`, according to the rules of the current NLS calendar parameter)	Any positive or negative integer
`HOUR`	00 to 23	0 to 23
`MINUTE`	00 to 59	0 to 59
`SECOND`	00 to 59.9(n), where 9(n) is the precision of time fractional seconds. The 9(n) portion is not applicable for `DATE`.	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
`TIMEZONE_HOUR`	-12 to 14 (This range accommodates daylight saving time changes.) Not applicable for `DATE` or `TIMESTAMP`.	Not applicable
`TIMEZONE_MINUTE` (See note at end of table)	00 to 59. Not applicable for `DATE` or `TIMESTAMP`.	Not applicable
`TIMEZONE_REGION`	Query the `TZNAME` column of the `V$TIMEZONE_NAMES` data dictionary view. Not applicable for `DATE` or `TIMESTAMP`. For a complete listing of all time zone region names, refer to *Oracle Database Globalization Support Guide*.	Not applicable

Table 2-4 (Cont.) Datetime Fields and Values

Datetime Field	Valid Values for Datetime	Valid Values for INTERVAL
TIMEZONE_ABBR	Query the TZABBREV column of the V$TIMEZONE_NAMES data dictionary view. Not applicable for DATE or TIMESTAMP.	Not applicable

> **Note:**
>
> TIMEZONE_HOUR and TIMEZONE_MINUTE are specified together and interpreted as an entity in the format +\|- *hh*:*mi*, with values ranging from -12:59 to +14:00. Refer to *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for information on specifying time zone values for that API.

DATE Data Type

The DATE data type stores date and time information. Although date and time information can be represented in both character and number data types, the DATE data type has special associated properties. For each DATE value, Oracle stores the following information: year, month, day, hour, minute, and second.

You can specify a DATE value as a literal, or you can convert a character or numeric value to a date value with the TO_DATE function. For examples of expressing DATE values in both these ways, refer to Datetime Literals .

Using Julian Days

A Julian day number is the number of days since January 1, 4712 BC. Julian days allow continuous dating from a common reference. You can use the date format model "J" with date functions TO_DATE and TO_CHAR to convert between Oracle DATE values and their Julian equivalents.

> **Note:**
>
> Oracle Database uses the astronomical system of calculating Julian days, in which the year 4713 BC is specified as -4712. The historical system of calculating Julian days, in contrast, specifies 4713 BC as -4713. If you are comparing Oracle Julian days with values calculated using the historical system, then take care to allow for the 365-day difference in BC dates. For more information, see http://aa.usno.navy.mil/faq/docs/millennium.php.

The default date values are determined as follows:

- The year is the current year, as returned by SYSDATE.
- The month is the current month, as returned by SYSDATE.
- The day is 01 (the first day of the month).

- The hour, minute, and second are all 0.

These default values are used in a query that requests date values where the date itself is not specified, as in the following example, which is issued in the month of May:

```
SELECT TO_DATE('2009', 'YYYY')
  FROM DUAL;

TO_DATE('
---------
01-MAY-09
```

Example

This statement returns the Julian equivalent of January 1, 2009:

```
SELECT TO_CHAR(TO_DATE('01-01-2009', 'MM-DD-YYYY'),'J')
    FROM DUAL;

TO_CHAR
-------
2454833
```

> **See Also:**
>
> Selecting from the DUAL Table for a description of the `DUAL` table

TIMESTAMP Data Type

The `TIMESTAMP` data type is an extension of the `DATE` data type. It stores the year, month, and day of the `DATE` data type, plus hour, minute, and second values. This data type is useful for storing precise time values and for collecting and evaluating date information across geographic regions. Specify the `TIMESTAMP` data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)]
```

where *`fractional_seconds_precision`* optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default is 6.

> **See Also:**
>
> TO_TIMESTAMP for information on converting character data to `TIMESTAMP` data

TIMESTAMP WITH TIME ZONE Data Type

`TIMESTAMP WITH TIME ZONE` is a variant of `TIMESTAMP` that includes a **time zone region name** or a **time zone offset** in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—

formerly Greenwich Mean Time). This data type is useful for preserving local time zone information.

Specify the `TIMESTAMP WITH TIME ZONE` data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE
```

where *`fractional_seconds_precision`* optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default is 6.

Oracle time zone data is derived from the public domain information available at `http://www.iana.org/time-zones/`. Oracle time zone data may not reflect the most recent data available at this site.

See Also:

- *Oracle Database Globalization Support Guide* for more information on Oracle time zone data
- Support for Daylight Saving Times and Table 2-19 for information on daylight saving support
- TO_TIMESTAMP_TZ for information on converting character data to `TIMESTAMP WITH TIME ZONE` data
- ALTER SESSION for information on the `ERROR_ON_OVERLAP_TIME` session parameter

TIMESTAMP WITH LOCAL TIME ZONE Data Type

`TIMESTAMP WITH LOCAL TIME ZONE` is another variant of `TIMESTAMP` that is sensitive to time zone information. It differs from `TIMESTAMP WITH TIME ZONE` in that data stored in the database is normalized to the database time zone, and the time zone information is not stored as part of the column data. When a user retrieves the data, Oracle returns it in the user's local session time zone. This data type is useful for date information that is always to be displayed in the time zone of the client system in a two-tier application.

Specify the `TIMESTAMP WITH LOCAL TIME ZONE` data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE
```

where *`fractional_seconds_precision`* optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default is 6.

Oracle time zone data is derived from the public domain information available at `http://www.iana.org/time-zones/`. Oracle time zone data may not reflect the most recent data available at this site.

See Also:

- *Oracle Database Globalization Support Guide* for more information on Oracle time zone data
- *Oracle Database Development Guide* for examples of using this data type and CAST for information on converting character data to TIMESTAMP WITH LOCAL TIME ZONE

INTERVAL YEAR TO MONTH Data Type

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. This data type is useful for representing the difference between two datetime values when only the year and month values are significant.

Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where *year_precision* is the number of digits in the YEAR datetime field. The default value of *year_precision* is 2.

You have a great deal of flexibility when specifying interval values as literals. Refer to Interval Literals for detailed information on specifying interval values as literals. Also see Datetime and Interval Examples for an example using intervals.

INTERVAL DAY TO SECOND Data Type

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. This data type is useful for representing the precise difference between two datetime values.

Specify this data type as follows:

```
INTERVAL DAY [(day_precision)]
   TO SECOND [(fractional_seconds_precision)]
```

where

- *day_precision* is the number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2.
- *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

You have a great deal of flexibility when specifying interval values as literals. Refer to Interval Literals for detailed information on specify interval values as literals. Also see Datetime and Interval Examples for an example using intervals.

Datetime/Interval Arithmetic

You can perform a number of arithmetic operations on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE) and interval (INTERVAL DAY TO SECOND and INTERVAL YEAR TO MONTH) data. Oracle calculates the results based on the following rules:

- You can use NUMBER constants in arithmetic operations on date and timestamp values, but not interval values. Oracle internally converts timestamp values to date values and interprets NUMBER constants in arithmetic datetime and interval expressions as numbers of days. For example, SYSDATE + 1 is tomorrow. SYSDATE - 7 is one week ago. SYSDATE + (10/1440) is ten minutes from now. Subtracting the hire_date column of the sample table employees from SYSDATE returns the number of days since each employee was hired. You cannot multiply or divide date or timestamp values.
- Oracle implicitly converts BINARY_FLOAT and BINARY_DOUBLE operands to NUMBER.
- Each DATE value contains a time component, and the result of many date operations include a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours. These fractions are also returned by Oracle built-in functions for common operations on DATE data. For example, the MONTHS_BETWEEN function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31-day month.
- If one operand is a DATE value or a numeric value, neither of which contains time zone or fractional seconds components, then:
  - Oracle implicitly converts the other operand to DATE data. The exception is multiplication of a numeric value times an interval, which returns an interval.
  - If the other operand has a time zone value, then Oracle uses the session time zone in the returned value.
  - If the other operand has a fractional seconds value, then the fractional seconds value is lost.
- When you pass a timestamp, interval, or numeric value to a built-in function that was designed only for the DATE data type, Oracle implicitly converts the non-DATE value to a DATE value. Refer to Datetime Functions for information on which functions cause implicit conversion to DATE.
- When interval calculations return a datetime value, the result must be an actual datetime value or the database returns an error. For example, the next two statements return errors:

  ```
  SELECT TO_DATE('31-AUG-2004','DD-MON-YYYY') + TO_YMINTERVAL('0-1')
    FROM DUAL;

  SELECT TO_DATE('29-FEB-2004','DD-MON-YYYY') + TO_YMINTERVAL('1-0')
    FROM DUAL;
  ```

  The first fails because adding one month to a 31-day month would result in September 31, which is not a valid date. The second fails because adding one year to a date that exists only every four years is not valid. However, the next statement succeeds, because adding four years to a February 29 date is valid:

  ```
  SELECT TO_DATE('29-FEB-2004', 'DD-MON-YYYY') + TO_YMINTERVAL('4-0')
    FROM DUAL;

  TO_DATE('
  ---------
  29-FEB-08
  ```
- Oracle performs all timestamp arithmetic in UTC time. For TIMESTAMP WITH LOCAL TIME ZONE, Oracle converts the datetime value from the database time zone to UTC and converts back to the database time zone after performing the arithmetic. For TIMESTAMP WITH TIME ZONE, the datetime value is always in UTC, so no conversion is necessary.

Table 2-5 is a matrix of datetime arithmetic operations. Dashes represent operations that are not supported.

Table 2-5 Matrix of Datetime Arithmetic

Operand & Operator	DATE	TIMESTAMP	INTERVAL	Numeric
DATE				
+	—	—	DATE	DATE
-	NUMBER	INTERVAL	DATE	DATE
*	—	—	—	—
/	—	—	—	—
TIMESTAMP				
+	—	—	TIMESTAMP	DATE
-	INTERVAL	INTERVAL	TIMESTAMP	DATE
*	—	—	—	—
/	—	—	—	—
INTERVAL				
+	DATE	TIMESTAMP	INTERVAL	—
-	—	—	INTERVAL	—
*	—	—	—	INTERVAL
/	—	—	—	INTERVAL
Numeric				
+	DATE	DATE	—	NA
-	—	—	—	NA
*	—	—	INTERVAL	NA
/	—	—	—	NA

Examples

You can add an interval value expression to a start time. Consider the sample table `oe.orders` with a column `order_date`. The following statement adds 30 days to the value of the `order_date` column:

```
SELECT order_id, order_date + INTERVAL '30' DAY AS "Due Date"
  FROM orders
  ORDER BY order_id, "Due Date";
```

Support for Daylight Saving Times

Oracle Database automatically determines, for any given time zone region, whether daylight saving is in effect and returns local time values accordingly. The datetime value is sufficient for Oracle to determine whether daylight saving time is in effect for a given region in all cases except **boundary cases**. A boundary case occurs during the period when daylight saving goes into or comes out of effect. For example, in the US-Pacific region, when daylight saving goes into effect, the time changes from 2:00 a.m. to 3:00 a.m. The one hour interval between 2 and 3 a.m. does not exist. When daylight saving goes out of effect, the time changes from 2:00 a.m. back to 1:00 a.m., and the one-hour interval between 1 and 2 a.m. is repeated.

To resolve these boundary cases, Oracle uses the `TZR` and `TZD` format elements, as described in Table 2-19. `TZR` represents the time zone region name in datetime input strings. Examples are '`Australia/North`', '`UTC`', and '`Singapore`'. `TZD` represents an abbreviated form of the time zone region name with daylight saving information. Examples are '`PST`' for US/Pacific standard time and '`PDT`' for US/Pacific daylight time. To see a listing of valid values for the `TZR` and `TZD` format elements, query the `TZNAME` and `TZABBREV` columns of the `V$TIMEZONE_NAMES` dynamic performance view.

Note:

Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

For a complete listing of the time zone region names in both files, refer to *Oracle Database Globalization Support Guide*.

Oracle time zone data is derived from the public domain information available at `http://www.iana.org/time-zones/`. Oracle time zone data may not reflect the most recent data available at this site.

See Also:

- Datetime Format Models for information on the format elements and the session parameter ERROR_ON_OVERLAP_TIME .
- *Oracle Database Globalization Support Guide* for more information on Oracle time zone data
- *Oracle Database Reference* for information on the dynamic performance views

Datetime and Interval Examples

The following example shows how to specify some datetime and interval data types.

```
CREATE TABLE time_table
  (start_time    TIMESTAMP,
   duration_1    INTERVAL DAY (6) TO SECOND (5),
   duration_2    INTERVAL YEAR TO MONTH);
```

The `start_time` column is of type `TIMESTAMP`. The implicit fractional seconds precision of `TIMESTAMP` is 6.

The `duration_1` column is of type `INTERVAL DAY TO SECOND`. The maximum number of digits in field `DAY` is 6 and the maximum number of digits in the fractional second is 5. The maximum number of digits in all other datetime fields is 2.

The `duration_2` column is of type `INTERVAL YEAR TO MONTH`. The maximum number of digits of the value in each field (`YEAR` and `MONTH`) is 2.

Interval data types do not have format models. Therefore, to adjust their presentation, you must combine character functions such as `EXTRACT` and concatenate the components. For example, the following examples query the `hr.employees` and `oe.orders` tables, respectively, and change interval output from the form "*yy-mm*" to "*yy* years *mm* months" and from "*dd-hh*" to "*dddd* days *hh* hours":

```
SELECT last_name, EXTRACT(YEAR FROM (SYSDATE - hire_date) YEAR TO MONTH)
       || ' years '
       || EXTRACT(MONTH FROM (SYSDATE - hire_date) YEAR TO MONTH)
       || ' months'  "Interval"
  FROM employees;

LAST_NAME                 Interval
------------------------- --------------------
OConnell                  2 years 3 months
Grant                     1 years 9 months
Whalen                    6 years 1 months
Hartstein                 5 years 8 months
Fay                       4 years 2 months
Mavris                    7 years 4 months
Baer                      7 years 4 months
Higgins                   7 years 4 months
Gietz                     7 years 4 months
. . .

SELECT order_id, EXTRACT(DAY FROM (SYSDATE - order_date) DAY TO SECOND)
       || ' days '
       || EXTRACT(HOUR FROM (SYSDATE - order_date) DAY TO SECOND)
       || ' hours' "Interval"
  FROM orders;

  ORDER_ID Interval
---------- --------------------
      2458 780 days 23 hours
      2397 685 days 22 hours
      2454 733 days 21 hours
      2354 447 days 20 hours
      2358 635 days 20 hours
      2381 508 days 18 hours
      2440 765 days 17 hours
      2357 1365 days 16 hours
      2394 602 days 15 hours
      2435 763 days 15 hours
. . .
```

RAW and LONG RAW Data Types

The `RAW` and `LONG RAW` data types store data that is not to be explicitly converted by Oracle Database when moving data between different systems. These data types are intended for binary data or byte strings. For example, you can use `LONG RAW` to store graphics, sound, documents, or arrays of binary data, for which the interpretation is dependent on the use.

Oracle strongly recommends that you convert `LONG RAW` columns to binary LOB (`BLOB`) columns. LOB columns are subject to far fewer restrictions than `LONG` columns. See TO_LOB for more information.

RAW is a variable-length data type like VARCHAR2, except that Oracle Net (which connects client software to a database or one database to another) and the Oracle import and export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Oracle Net and the Oracle import and export utilities automatically convert CHAR, VARCHAR2, and LONG data between different database character sets, if data is transported between databases, or between the database character set and the client character set, if data is transported between a database and a client. The client character set is determined by the type of the client interface, such as OCI or JDBC, and the client configuration (for example, the NLS_LANG environment variable).

When Oracle implicitly converts RAW or LONG RAW data to character data, the resulting character value contains a hexadecimal representation of the binary input, where each character is a hexadecimal digit (0-9, A-F) representing four consecutive bits of RAW data. For example, one byte of RAW data with bits 11001011 becomes the value CB.

When Oracle implicitly converts character data to RAW or LONG RAW, it interprets each consecutive input character as a hexadecimal representation of four consecutive bits of binary data and builds the resulting RAW or LONG RAW value by concatenating those bits. If any of the input characters is not a hexadecimal digit (0-9, A-F, a-f), then an error is reported. If the number of characters is odd, then the result is undefined.

The SQL functions RAWTOHEX and HEXTORAW perform explicit conversions that are equivalent to the above implicit conversions. Other types of conversions between RAW and character data are possible with functions in the Oracle-supplied PL/SQL packages UTL_RAW and UTL_I18N.

Large Object (LOB) Data Types

The built-in LOB data types BLOB, CLOB, and NCLOB (stored internally) and BFILE (stored externally) can store large and unstructured data such as text, image, video, and spatial data. The size of BLOB, CLOB, and NCLOB data can be up to (2^{32}-1 bytes) * (the value of the CHUNK parameter of LOB storage). If the tablespaces in your database are of standard block size, and if you have used the default value of the CHUNK parameter of LOB storage when creating a LOB column, then this is equivalent to (2^{32}-1 bytes) * (database block size). BFILE data can be up to 2^{64}-1 bytes, although your operating system may impose restrictions on this maximum.

When creating a table, you can optionally specify different tablespace and storage characteristics for LOB columns or LOB object attributes from those specified for the table.

CLOB, NCLOB, and BLOB values up to approximately 4000 bytes are stored inline if you enable storage in row at the time the LOB column is created. LOBs greater than 4000 bytes are always stored externally. Refer to ENABLE STORAGE IN ROW for more information.

LOB columns contain LOB locators that can refer to internal (in the database) or external (outside the database) LOB values. Selecting a LOB from a table actually returns the LOB locator and not the entire LOB value. The DBMS_LOB package and Oracle Call Interface (OCI) operations on LOBs are performed through these locators.

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

- LOBs can be attributes of an object type (user-defined data type).
- The LOB locator is stored in the table column, either with or without the actual LOB value. BLOB, NCLOB, and CLOB values can be stored in separate tablespaces. BFILE data is stored in an external file on the server.
- When you access a LOB column, the locator is returned.

- A LOB can be up to (2^{32}-1 bytes)*(database block size) in size. BFILE data can be up to 2^{64}-1 bytes, although your operating system may impose restrictions on this maximum.
- LOBs permit efficient, random, piece-wise access to and manipulation of data.
- You can define more than one LOB column in a table.
- With the exception of NCLOB, you can define one or more LOB attributes in an object.
- You can declare LOB bind variables.
- You can select LOB columns and LOB attributes.
- You can insert a new row or update an existing row that contains one or more LOB columns or an object with one or more LOB attributes. In update operations, you can set the internal LOB value to NULL, empty, or replace the entire LOB with data. You can set the BFILE to NULL or make it point to a different file.
- You can update a LOB row-column intersection or a LOB attribute with another LOB row-column intersection or LOB attribute.
- You can delete a row containing a LOB column or LOB attribute and thereby also delete the LOB value. For BFILEs, the actual operating system file is not deleted.

You can access and populate rows of an inline LOB column (a LOB column stored in the database) or a LOB attribute (an attribute of an object type column stored in the database) simply by issuing an INSERT or UPDATE statement.

Restrictions on LOB Columns

LOB columns are subject to a number of rules and restrictions. See *Oracle Database SecureFiles and Large Objects Developer's Guide* for a complete listing.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* and *Oracle Call Interface Programmer's Guide* for more information about these interfaces and LOBs
- the *modify_col_properties* clause of ALTER TABLE and TO_LOB for more information on converting LONG columns to LOB columns

BFILE Data Type

The BFILE data type enables access to binary file LOBs that are stored in file systems outside Oracle Database. A BFILE column or attribute stores a BFILE locator, which serves as a pointer to a binary file on the server file system. The locator maintains the directory name and the filename.

You can change the filename and path of a BFILE without affecting the base table by using the BFILENAME function. Refer to BFILENAME for more information on this built-in SQL function.

Binary file LOBs do not participate in transactions and are not recoverable. Rather, the underlying operating system provides file integrity and durability. BFILE data can be up to 2^{64}-1 bytes, although your operating system may impose restrictions on this maximum.

The database administrator must ensure that the external file exists and that Oracle processes have operating system read permissions on the file.

The `BFILE` data type enables read-only support of large binary files. You cannot modify or replicate such a file. Oracle provides APIs to access file data. The primary interfaces that you use to access file data are the `DBMS_LOB` package and Oracle Call Interface (OCI).

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* and *Oracle Call Interface Programmer's Guide* for more information about LOBs and CREATE DIRECTORY

BLOB Data Type

The `BLOB` data type stores unstructured binary large objects. `BLOB` objects can be thought of as bitstreams with no character set semantics. `BLOB` objects can store binary data up to (4 gigabytes -1) * (the value of the `CHUNK` parameter of LOB storage). If the tablespaces in your database are of standard block size, and if you have used the default value of the `CHUNK` parameter of LOB storage when creating a LOB column, then this is equivalent to (4 gigabytes - 1) * (database block size).

`BLOB` objects have full transactional support. Changes made through SQL, the `DBMS_LOB` package, or Oracle Call Interface (OCI) participate fully in the transaction. `BLOB` value manipulations can be committed and rolled back. However, you cannot save a `BLOB` locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

CLOB Data Type

The `CLOB` data type stores single-byte and multibyte character data. Both fixed-width and variable-width character sets are supported, and both use the database character set. `CLOB` objects can store up to (4 gigabytes -1) * (the value of the `CHUNK` parameter of LOB storage) of character data. If the tablespaces in your database are of standard block size, and if you have used the default value of the `CHUNK` parameter of LOB storage when creating a LOB column, then this is equivalent to (4 gigabytes - 1) * (database block size).

`CLOB` objects have full transactional support. Changes made through SQL, the `DBMS_LOB` package, or Oracle Call Interface (OCI) participate fully in the transaction. `CLOB` value manipulations can be committed and rolled back. However, you cannot save a `CLOB` locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB Data Type

The `NCLOB` data type stores Unicode data. Both fixed-width and variable-width character sets are supported, and both use the national character set. `NCLOB` objects can store up to (4 gigabytes -1) * (the value of the `CHUNK` parameter of LOB storage) of character text data. If the tablespaces in your database are of standard block size, and if you have used the default value of the `CHUNK` parameter of LOB storage when creating a LOB column, then this is equivalent to (4 gigabytes - 1) * (database block size).

NCLOB objects have full transactional support. Changes made through SQL, the DBMS_LOB package, or OCI participate fully in the transaction. NCLOB value manipulations can be committed and rolled back. However, you cannot save an NCLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support

Extended Data Types

Beginning with Oracle Database 12*c*, you can specify a maximum size of 32767 bytes for the VARCHAR2, NVARCHAR2, and RAW data types. You can control whether your database supports this new maximum size by setting the initialization parameter MAX_STRING_SIZE as follows:

- If MAX_STRING_SIZE = STANDARD, then the size limits for releases prior to Oracle Database 12*c* apply: 4000 bytes for the VARCHAR2 and NVARCHAR2 data types, and 2000 bytes for the RAW data type. This is the default.
- If MAX_STRING_SIZE = EXTENDED, then the size limit is 32767 bytes for the VARCHAR2, NVARCHAR2, and RAW data types.

> **See Also:**
>
> Setting MAX_STRING_SIZE = EXTENDED may update database objects and possibly invalidate them. Refer to *Oracle Database Reference* for complete information on the implications of this parameter and how to set and enable this new functionality.

A VARCHAR2 or NVARCHAR2 data type with a declared size of greater than 4000 bytes, or a RAW data type with a declared size of greater than 2000 bytes, is an **extended data type**. Extended data type columns are stored out-of-line, leveraging Oracle's LOB technology. The LOB storage is always aligned with the table. In tablespaces managed with Automatic Segment Space Management (ASSM), extended data type columns are stored as SecureFiles LOBs. Otherwise, they are stored as BasicFiles LOBs. The use of LOBs as a storage mechanism is internal only. Therefore, you cannot manipulate these LOBs using the DBMS_LOB package.

> **Note:**
>
> - Oracle strongly discourages the use of BasicFiles LOBs as a storage mechanism. BasicFiles LOBs not only impose restrictions on the capabilities of extended data type columns, but the BasicFiles data type is planned to be deprecated in a future release.
> - Extended data types are subject to the same rules and restrictions as LOBs. Refer to *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

Note that, although you must set `MAX_STRING_SIZE = EXTENDED` in order to set the size of a `RAW` data type to greater than 2000 bytes, a `RAW` data type is stored as an out-of-line LOB only if it has a size of greater than 4000 bytes. For example, you must set `MAX_STRING_SIZE = EXTENDED` in order to declare a `RAW(3000)` data type. However, the column is stored inline.

You can use extended data types just as you would standard data types, with the following considerations:

- For special considerations when creating an index on an extended data type column, or when requiring an index to enforce a primary key or unique constraint, see Creating an Index on an Extended Data Type Column.
- If the partitioning key column for a list partition is an extended data type column, then the list of values that you want to specify for a partition may exceed the 4K byte limit for the partition bounds. See the *list_partitions* clause of `CREATE TABLE` for information on how to work around this issue.
- The value of the initialization parameter `MAX_STRING_SIZE` affects the following:
  - The maximum length of a text literal. See Text Literals for more information.
  - The size limit for concatenating two character strings. See Concatenation Operator for more information.
  - The length of the collation key returned by the `NLSSORT` function. See NLSSORT .
  - The size of some of the attributes of the `XMLFormat` object. See XML Format Model for more information.
  - The size of some expressions in the following XML functions: XMLCOLATTVAL , XMLELEMENT , XMLFOREST , XMLPI , and XMLTABLE .

Rowid Data Types

Each row in the database has an address. The sections that follow describe the two forms of row address in an Oracle Database.

ROWID Data Type

The rows in heap-organized tables that are native to Oracle Database have row addresses called **rowids**. You can examine a rowid row address by querying the pseudocolumn `ROWID`. Values of this pseudocolumn are strings representing the address of each row. These strings have the data type `ROWID`. You can also create tables and clusters that contain actual columns having the `ROWID` data type. Oracle Database does not guarantee that the values of such columns are valid rowids. Refer to Pseudocolumns for more information on the ROWID pseudocolumn.

Rowids contain the following information:

- The **data block** of the data file containing the row. The length of this string depends on your operating system.
- The **row** in the data block.
- The **database file** containing the row. The first data file has the number 1. The length of this string depends on your operating system.
- The **data object number**, which is an identification number assigned to every database segment. You can retrieve the data object number from the data dictionary views USER_OBJECTS, DBA_OBJECTS, and ALL_OBJECTS. Objects that share the same segment (clustered tables in the same cluster, for example) have the same object number.

Rowids are stored as base 64 values that can contain the characters A-Z, a-z, 0-9, and the plus sign (+) and forward slash (/). Rowids are not available directly. You can use the supplied package DBMS_ROWID to interpret rowid contents. The package functions extract and provide information on the four rowid elements listed above.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the functions available with the DBMS_ROWID package and how to use them

UROWID Data Type

The rows of some tables have addresses that are not physical or permanent or were not generated by Oracle Database. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Rowids of foreign tables (such as DB2 tables accessed through a gateway) are not standard Oracle rowids.

Oracle uses universal rowids (**urowids**) to store the addresses of index-organized and foreign tables. Index-organized tables have logical urowids and foreign tables have foreign urowids. Both types of urowid are stored in the ROWID pseudocolumn (as are the physical rowids of heap-organized tables).

Oracle creates logical rowids based on the primary key of the table. The logical rowids do not change as long as the primary key does not change. The ROWID pseudocolumn of an index-organized table has a data type of UROWID. You can access this pseudocolumn as you would the ROWID pseudocolumn of a heap-organized table (using a SELECT ... ROWID statement). If you want to store the rowids of an index-organized table, then you can define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

ANSI, DB2, and SQL/DS Data Types

SQL statements that create tables and clusters can also use ANSI data types and data types from the IBM products SQL/DS and DB2. Oracle recognizes the ANSI or IBM data type name that differs from the Oracle Database data type name. It converts the data type to the equivalent Oracle data type, records the Oracle data type as the name of the column data type, and stores the column data in the Oracle data type based on the conversions shown in the tables that follow.

Table 2-6 ANSI Data Types Converted to Oracle Data Types

ANSI SQL Data Type	Oracle Data Type
`CHARACTER(n)` `CHAR(n)`	`CHAR(n)`
`CHARACTER VARYING(n)` `CHAR VARYING(n)`	`VARCHAR2(n)`

`NATIONAL CHARACTER(n)`
`NATIONAL CHAR(n)`
`NCHAR(n)`
	`NCHAR(n)`

`NATIONAL CHARACTER VARYING(n)`
`NATIONAL CHAR VARYING(n)`
`NCHAR VARYING(n)`
	`NVARCHAR2(n)`

`NUMERIC[(p,s)]` `DECIMAL[(p,s)]` (**Note 1**)	`NUMBER(p,s)`

`INTEGER`
`INT`
`SMALLINT`
	`NUMBER(38)`

`FLOAT` (**Note 2**)
`DOUBLE PRECISION` (**Note 3**)
`REAL` (**Note 4**)
	`FLOAT(126)`
	`FLOAT(126)`
	`FLOAT(63)`

Notes:

1. The `NUMERIC` and `DECIMAL` data types can specify only fixed-point numbers. For those data types, the scale (`s`) defaults to 0.
2. The `FLOAT` data type is a floating-point number with a binary precision b. The default precision for this data type is 126 binary, or 38 decimal.
3. The `DOUBLE PRECISION` data type is a floating-point number with binary precision 126.
4. The `REAL` data type is a floating-point number with a binary precision of 63, or 18 decimal.

Do not define columns with the following SQL/DS and DB2 data types, because they have no corresponding Oracle data type:

- `GRAPHIC`
- `LONG VARGRAPHIC`
- `VARGRAPHIC`
- `TIME`

Note that data of type `TIME` can also be expressed as Oracle datetime data.

See Also:

Datetime and Interval Data Types

Table 2-7 SQL/DS and DB2 Data Types Converted to Oracle Data Types

SQL/DS or DB2 Data Type	Oracle Data Type
`CHARACTER(n)`	`CHAR(n)`
`VARCHAR(n)`	`VARCHAR(n)`
`LONG VARCHAR`	`LONG`
`DECIMAL(p,s)` (**Note 1**)	`NUMBER(p,s)`
`INTEGER` `SMALLINT`	`NUMBER(p,0)`
`FLOAT` (**Note 2**)	`NUMBER`

Notes:

1. The `DECIMAL` data type can specify only fixed-point numbers. For this data type, *s* defaults to 0.
2. The `FLOAT` data type is a floating-point number with a binary precision *b*. The default precision for this data type is 126 binary or 38 decimal.

User-Defined Types

User-defined data types use Oracle built-in data types and other user-defined data types as the building blocks of object types that model the structure and behavior of data in applications. The sections that follow describe the various categories of user-defined types.

See Also:

- *Oracle Database Concepts* for information about Oracle built-in data types
- CREATE TYPE and the CREATE TYPE BODY for information about creating user-defined types
- *Oracle Database Object-Relational Developer's Guide* for information about using user-defined types

Object Types

Object types are abstractions of the real-world entities, such as purchase orders, that application programs deal with. An object type is a schema object with three kinds of components:

- A **name**, which identifies the object type uniquely within that schema.
- **Attributes**, which are built-in types or other user-defined types. Attributes model the structure of the real-world entity.
- **Methods**, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C or Java and stored externally. Methods implement operations the application can perform on the real-world entity.

REF Data Types

An **object identifier** (represented by the keyword `OID`) uniquely identifies an object and enables you to reference the object from other objects or from relational tables. A data type category called `REF` represents such references. A `REF` data type is a container for an object identifier. `REF` values are pointers to objects.

When a `REF` value points to a nonexistent object, the `REF` is said to be "dangling". A dangling `REF` is different from a null `REF`. To determine whether a `REF` is dangling or not, use the condition `IS [NOT] DANGLING`. For example, given object view `oc_orders` in the sample schema `oe`, the column `customer_ref` is of type `REF` to type `customer_typ`, which has an attribute `cust_email`:

```
SELECT o.customer_ref.cust_email
  FROM oc_orders o
  WHERE o.customer_ref IS NOT DANGLING;
```

Varrays

An array is an ordered set of data elements. All elements of a given array are of the same data type. Each element has an **index**, which is a number corresponding to the position of the element in the array.

The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called **varrays**. You must specify a maximum size when you declare the varray.

When you declare a varray, it does not allocate space. It defines a type, which you can use as:

- The data type of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

Oracle normally stores an array object either in line (as part of the row data) or out of line (in a LOB), depending on its size. However, if you specify separate storage characteristics for a varray, then Oracle stores it out of line, regardless of its size. Refer to the *varray_col_properties* of CREATE TABLE for more information about varray storage.

Nested Tables

A nested table type models an unordered set of elements. The elements may be built-in types or user-defined types. You can view a nested table as a single-column table or, if the nested table is an object type, as a multicolumn table, with a column for each attribute of the object type.

A nested table definition does not allocate space. It defines a type, which you can use to declare:

- The data type of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

When a nested table appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

Oracle-Supplied Types

Oracle provides SQL-based interfaces for defining new types when the built-in or ANSI-supported types are not sufficient. The behavior for these types can be implemented in C/C++, Java, or PL/ SQL. Oracle Database automatically provides the low-level infrastructure services needed for input-output, heterogeneous client-side access for new data types, and optimizations for data transfers between the application and the database.

These interfaces can be used to build user-defined (or object) types and are also used by Oracle to create some commonly useful data types. Several such data types are supplied with the server, and they serve both broad horizontal application areas (for example, the `Any` types) and specific vertical ones (for example, the spatial types).

The Oracle-supplied types, along with cross-references to the documentation of their implementation and use, are described in the following sections:

- Any Types
- XML Types
- Spatial Types
- Media Types

Any Types

The `Any` types provide highly flexible modeling of procedure parameters and table columns where the actual type is not known. These data types let you dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type. These types have OCI and PL/SQL interfaces for construction and access.

ANYTYPE

This type can contain a type description of any named SQL type or unnamed transient type.

ANYDATA

This type contains an instance of a given type, with data, plus a description of the type. `ANYDATA` can be used as a table column data type and lets you store heterogeneous values in a single column. The values can be of SQL built-in types as well as user-defined types.

ANYDATASET

This type contains a description of a given type plus a set of data instances of that type. `ANYDATASET` can be used as a procedure parameter data type where such flexibility is needed. The values of the data instances can be of SQL built-in types as well as user-defined types.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types

XML Types

Extensible Markup Language (XML) is a standard format developed by the World Wide Web Consortium (W3C) for representing structured and unstructured data on the World Wide Web. Universal resource identifiers (URIs) identify resources such as Web pages anywhere on the Web. Oracle provides types to handle XML and URI data, as well as a class of URIs called `DBURIRef` types to access data stored within the database itself. It also provides a set of types to store and access both external and internal URIs from within the database.

XMLType

This Oracle-supplied type can be used to store and query XML data in the database. `XMLType` has member functions you can use to access, extract, and query the XML data using XPath expressions. XPath is another standard developed by the W3C committee to traverse XML documents. Oracle `XMLType` functions support many W3C XPath expressions. Oracle also provides a set of SQL functions and PL/SQL packages to create `XMLType` values from existing relational or object-relational data.

`XMLType` is a system-defined type, so you can use it as an argument of a function or as the data type of a table or view column. You can also create tables and views of `XMLType`. When you create an `XMLType` column in a table, you can choose to store the XML data in a `CLOB` column, as binary XML (stored internally as a `CLOB`), or object relationally.

You can also register the schema (using the `DBMS_XMLSCHEMA` package) and create a table or column conforming to the registered schema. In this case Oracle stores the XML data in underlying object-relational columns by default, but you can specify storage in a `CLOB` or binary XML column even for schema-based data.

Queries and DML on `XMLType` columns operate the same regardless of the storage mechanism.

> **See Also:**
>
> *Oracle XML DB Developer's Guide*for information about using XMLType columns

URI Data Types

Oracle supplies a family of URI types—`URIType`, `DBURIType`, `XDBURIType`, and `HTTPURIType`—which are related by an inheritance hierarchy. `URIType` is an object type and the others are subtypes of `URIType`. Since `URIType` is the supertype, you can create columns of this type and store `DBURIType` or `HTTPURIType` type instances in this column.

HTTPURIType

You can use `HTTPURIType` to store URLs to external Web pages or to files. Oracle accesses these files using HTTP (Hypertext Transfer Protocol).

XDBURIType

You can use `XDBURIType` to expose documents in the XML database hierarchy as URIs that can be embedded in any `URIType` column in a table. The `XDBURIType` consists of a URL, which comprises the hierarchical name of the XML document to which it refers and an optional fragment representing the XPath syntax. The fragment is separated from the URL part by a pound sign (#). The following lines are examples of `XDBURIType`:

```
/home/oe/doc1.xml
/home/oe/doc1.xml#/orders/order_item
```

DBURIType

`DBURIType` can be used to store `DBURIRef` values, which reference data inside the database. Storing `DBURIRef` values lets you reference data stored inside or outside the database and access the data consistently.

`DBURIRef` values use an XPath-like representation to reference data inside the database. If you imagine the database as an XML tree, then you would see the tables, rows, and columns as elements in the XML document. For example, the sample human resources user `hr` would see the following XML tree:

```
<HR>
  <EMPLOYEES>
    <ROW>
      <EMPLOYEE_ID>205</EMPLOYEE_ID>
      <LAST_NAME>Higgins</LAST_NAME>
      <SALARY>12008</SALARY>
      .. <!-- other columns -->
    </ROW>
    ... <!-- other rows -->
  </EMPLOYEES>
  <!-- other tables..-->
</HR>
<!-- other user schemas on which you have some privilege on..-->
```

The `DBURIRef` is an XPath expression over this virtual XML document. So to reference the `SALARY` value in the `EMPLOYEES` table for the employee with employee number 205, you can write a `DBURIRef` as,

```
/HR/EMPLOYEES/ROW[EMPLOYEE_ID=205]/SALARY
```

Using this model, you can reference data stored in CLOB columns or other columns and expose them as URLs to the external world.

URIFactory Package

Oracle also provides the URIFactory package, which can create and return instances of the various subtypes of the URITypes. The package analyzes the URL string, identifies the type of URL (HTTP, DBURI, and so on), and creates an instance of the subtype. To create a DBURI instance, the URL must begin with the prefix /oradb. For example, URIFactory.getURI('/oradb/HR/EMPLOYEES') would create a DBURIType instance and URIFactory.getUri('/sys/schema') would create an XDBURIType instance.

See Also:

- *Oracle Database Object-Relational Developer's Guide* for general information on object types and type inheritance
- *Oracle XML DB Developer's Guide* for more information about these supplied types and their implementation
- *Oracle Database Advanced Queuing User's Guide* for information about using XMLType with Oracle Advanced Queuing

Spatial Types

Oracle Spatial and Graph is designed to make spatial data management easier and more natural to users of location-enabled applications, geographic information system (GIS) applications, and geoimaging applications. After the spatial data is stored in an Oracle Database, you can easily manipulate, retrieve, and relate it to all the other data stored in the database. The following data types are available only if you have installed Oracle Spatial and Graph.

SDO_GEOMETRY

The geometric description of a spatial object is stored in a single row, in a single column of object type SDO_GEOMETRY in a user-defined table. Any table that has a column of type SDO_GEOMETRY must have another column, or set of columns, that defines a unique primary key for that table. Tables of this sort are sometimes called geometry tables.

The SDO_GEOMETRY object type has the following definition:

```
CREATE TYPE SDO_GEOMETRY AS OBJECT
  (sgo_gtype        NUMBER,
   sdo_srid         NUMBER,
   sdo_point        SDO_POINT_TYPE,
   sdo_elem_info    SDO_ELEM_INFO_ARRAY,
   sdo_ordinates    SDO_ORDINATE_ARRAY);
/
```

SDO_TOPO_GEOMETRY

This type describes a topology geometry, which is stored in a single row, in a single column of object type `SDO_TOPO_GEOMETRY` in a user-defined table.

The `SDO_TOPO_GEOMETRY` object type has the following definition:

```
CREATE TYPE SDO_TOPO_GEOMETRY AS OBJECT
  (tg_type        NUMBER,
   tg_id          NUMBER,
   tg_layer_id    NUMBER,
   topology_id    NUMBER);
/
```

SDO_GEORASTER

In the GeoRaster object-relational model, a raster grid or image object is stored in a single row, in a single column of object type `SDO_GEORASTER` in a user-defined table. Tables of this sort are called GeoRaster tables.

The `SDO_GEORASTER` object type has the following definition:

```
CREATE TYPE SDO_GEORASTER AS OBJECT
  (rasterType         NUMBER,
   spatialExtent      SDO_GEOMETRY,
   rasterDataTable    VARCHAR2(32),
   rasterID           NUMBER,
   metadata           XMLType);
/
```

> **See Also:**
>
> *Oracle Spatial and Graph Developer's Guide*, *Oracle Spatial and Graph Topology Data Model and Network Data Model Graph Developer's Guide*, and *Oracle Spatial and Graph GeoRaster Developer's Guide* for information on the full implementation of the spatial data types and guidelines for using them

Media Types

Oracle Multimedia can use object types, similar to Java or C++ classes, to describe multimedia data. An instance of these object types consists of attributes, including metadata and the media data, and methods. The Multimedia data types are created in the `ORDSYS` schema. Public synonyms exist for all the data types, so you can access them without specifying the schema name.

Oracle Multimedia provides the following object types:

- `ORDAudio`

  Supports the storage and management of audio data.

- `ORDDicom`

Supports the storage and management of Digital Imaging and Communications in Medicine (DICOM), the format universally recognized as the standard for medical imaging.

- `ORDDoc`

  Supports storage and management of any type of media data, including audio, image and video data. Use this type when you want all media to be stored in a single column.

- `ORDImage`

  Supports the storage and management of image data.

- `ORDVideo`

  Supports the storage and management of video data.

The following data types provide compliance with the ISO-IEC 13249-5 Still Image standard, commonly referred to as SQL/MM StillImage:

- `SI_AverageColor`

  Represents a feature that characterizes an image by its average color.

- `SI_Color`

  Encapsulates color values.

- `SI_ColorHistogram`

  Represents a feature that characterizes an image by the relative frequencies of the colors exhibited by samples of the raw image.

- `SI_FeatureList`

  A list containing up to four of the image features represented by the preceding object types (`SI_AverageColor`, `SI_ColorHistogram`, `SI_PositionalColor`, and `SI_Texture`), where each feature is associated with a feature weight.

- `SI_PositionalColor`

  Given an image divided into `n` by `m` rectangles, the `SI_PositionalColor` object type represents the feature that characterizes an image by the `n` by `m` most significant colors of the rectangles.

- `SI_StillImage`

  Represents digital images with inherent image characteristics such as height, width, and format.

- `SI_Texture`

  Represents a feature that characterizes an image by the size of repeating items (coarseness), brightness variations (contrast), and predominant direction (directionality).

> **Note:**
>
> The Oracle Multimedia support for object types that comply with the first edition of the ISO/IEC 13249-5:2001 SQLMM Part5:StillImage standard (commonly referred to as the SQL/MM Still Image standard) is deprecated in Oracle Database 12*c* Release 2 (12.2), and may be desupported in a future release. See *Oracle Multimedia Reference* for more information.

See Also:

- *Oracle Multimedia DICOM Developer's Guide* for information on the `ORDDicom` object type
- *Oracle Multimedia Reference* for information on all other object types listed in this section

Data Type Comparison Rules

This section describes how Oracle Database compares values of each data type.

Numeric Values

A larger value is considered greater than a smaller one. All negative numbers are less than zero and all positive numbers. Thus, -1 is less than 100; -100 is less than -1.

The floating-point value `NaN` (not a number) is greater than any other numeric value and is equal to itself.

See Also:

Numeric Precedence and Floating-Point Numbers for more information on comparison semantics

Datetime Values

A later date or timestamp is considered greater than an earlier one. For example, the date equivalent of '29-MAR-2005' is less than that of '05-JAN-2006' and the timestamp equivalent of '05-JAN-2006 1:35pm' is greater than that of '05-JAN-2005 10:09am'.

When two timestamps with time zone are compared, they are first normalized to UTC, that is, to the timezone offset '+00:00'. For example, the timestamp with time zone equivalent of '16-OCT-2016 05:59am Europe/Warsaw' is equal to that of '15-OCT-2016 08:59pm US/Pacific'. Both represent the same absolute point in time, which represented in UTC is October 16th, 2016, 03:59am.

Binary Values

A binary value of the data type `RAW` or `BLOB` is a sequence of bytes. When two binary values are compared, the corresponding, consecutive bytes of the two byte sequences are compared in turn. If the first bytes of both compared values are different, the binary value that contains the byte with the lower numeric value is considered smaller. If the first bytes are equal, second bytes are compared analogously, and so on, until either the compared bytes differ or the comparison process reaches the end of one of the values. In the latter case, the value that is shorter is considered smaller.

Binary values of the data type `BLOB` cannot be compared directly in comparison conditions. However, they can be compared with the PL/SQL function `DBMS_LOB.COMPARE`.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_LOB.COMPARE` function

Character Values

Character values are compared on the basis of two measures:

- Binary or linguistic collation
- Blank-padded or nonpadded comparison semantics

The following subsections describe the two measures.

Binary and Linguistic Collation

In binary collation, which is the default, Oracle compares character values like binary values. Two sequences of bytes that form the encodings of two character values in their storage character set are treated as binary values and compared as described in Binary Values . The result of this comparison is returned as the result of the binary comparison of the source character values.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on character sets

For many languages, the binary collation can yield a linguistically incorrect ordering of character values. For example, in most common character sets, all the uppercase Latin letters have character codes with lower values than all the lowercase Latin letters. Hence, the binary collation yields the following order:

```
MacDonald
MacIntosh
Macdonald
Macintosh
```

However, most users expect these four values to be presented in the order:

```
MacDonald
Macdonald
MacIntosh
Macintosh
```

This shows that binary collation may not be suitable even for English character values.

Oracle Database supports linguistic collations that order strings according to rules of various spoken languages. It also supports collation variants that match character values case- and accent-insensitively. Linguistic collations are more expensive but they provide superior user experience.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about linguistic sorting

Restrictions for Linguistic Collations

Comparison conditions, `ORDER BY`, `GROUP BY` and `MATCH_RECOGNIZE` query clauses, `COUNT(DISTINCT)` and statistical aggregate functions, `LIKE` conditions, and `ORDER BY` and `PARTITION BY` analytic clauses generate collation keys when using linguistic collations. The collation keys are the same values that are returned by the function `NLSSORT` and are subject to the same restrictions that are described in NLSSORT .

Blank-Padded and Nonpadded Comparison Semantics

With blank-padded semantics, if the two values have different lengths, then Oracle first adds blanks to the end of the shorter one so their lengths are equal. Oracle then compares the values character by character up to the first character that differs. The value with the greater character in the first differing position is considered greater. If two values have no differing characters, then they are considered equal. This rule means that two values are equal if they differ only in the number of trailing blanks. Oracle uses blank-padded comparison semantics only when both values in the comparison are either expressions of data type `CHAR`, `NCHAR`, text literals, or values returned by the `USER` function.

With nonpadded semantics, Oracle compares two values character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values of different length are identical up to the end of the shorter one, then the longer value is considered greater. If two values of equal length have no differing characters, then the values are considered equal. Oracle uses nonpadded comparison semantics whenever one or both values in the comparison have the data type `VARCHAR2` or `NVARCHAR2`.

The results of comparing two character values using different comparison semantics may vary. The table that follows shows the results of comparing five pairs of character values using each comparison semantic. Usually, the results of blank-padded and nonpadded comparisons are the same. The last comparison in the table illustrates the differences between the blank-padded and nonpadded comparison semantics.

Blank-Padded	Nonpadded
`'ac' > 'ab'`	`'ac' > 'ab'`
`'ab' > 'a '`	`'ab' > 'a '`
`'ab' > 'a'`	`'ab' > 'a'`
`'ab' = 'ab'`	`'ab' = 'ab'`
`'a ' = 'a'`	**`'a ' > 'a'`**

Data-Bound Collation

Starting with Oracle Database 12*c* Release 2 (12.2), the collation to use when comparing or matching a given character value is associated with the value itself. It is called the **data-bound collation**. The data-bound collation can be viewed as an attribute of the data type of the value.

In previous Oracle Database releases, the session parameters `NLS_COMP` and `NLS_SORT` coarsely determined the collation for all collation-sensitive SQL operations in a database session. The data-bound collation architecture enables applications to consistently apply language-specific comparison rules to exactly the data that needs these rules.

Oracle Database 12*c* Release 2 (12.2) allows you to declare a collation for a table column. When a column is passed as an argument to a collation-sensitive SQL operation, the SQL operation uses the column's declared collation to process the column's values. If the SQL operation has multiple character arguments that are compared to each other, the **collation determination rules** determine the collation to use.

There are two types of data-bound collations:

- Named Collation: This collation is a particular set of collating rules specified by a collation name. Named collations are the same collations that are specified as values for the `NLS_SORT` parameter. A named collation can be either a binary collation or a linguistic collation.
- Pseudo-collation: This collation does not directly specify the collating rules for a SQL operation. Instead, it instructs the operation to check the values of the session parameters `NLS_SORT` and `NLS_COMP` for the actual named collation to use. Pseudo-collations are the bridge between the new declarative method of specifying collations and the old method that uses session parameters. In particular, the pseudo-collation `USING_NLS_COMP` directs a SQL operation to behave exactly as it used to behave before Oracle Database 12*c* Release 2.

When you declare a named collation for a column, you statically determine how the column values are compared. When you declare a pseudo-collation, you can dynamically control comparison behavior with the session parameter `NLS_COMP` and `NLS_SORT`. However, static objects, such as indexes and constraints, defined on a column declared with a pseudo-collation, fall back to using a binary collation. Dynamically settable collating rules cannot be used to compare values for a static object.

The collation for a character literal or bind variable that is used in an expression is derived from the default collation of the database object containing the expression, such as a view or materialized view query, a PL/SQL stored unit code, a user-defined type method code, or a standalone DML or query statement. In Oracle Database 12*c* Release 2, the default collation of PL/SQL stored units, user-defined type methods, and standalone SQL statements is always the pseudo-collation `USING_NLS_COMP`. The default collation of views and materialized views can be specified in the `DEFAULT COLLATION` clause of the `CREATE VIEW` and `CREATE MATERIALIZED VIEW` statements.

If a SQL operation returns character values, the **collation derivation rules** determine the **derived collation** for the result, so that its collation is known, when the result is passed as an argument to another collation-sensitive SQL operation in the expression tree or to a top-level consumer, such as an SQL statement clause in a `SELECT` statement. If a SQL operation operates on character argument values, then the derived collation of its character result is based on the collations of the arguments. Otherwise, the derivation rules are the same as for a character literal.

You can override the derived collation of an expression node, such as a simple expression or an operator result, by using the `COLLATE` operator.

Oracle Database allows you to declare a case-insensitive collation for a column, table or schema, so that the column or all character columns in a table or a schema can be always compared in a case-insensitive way.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for more information on data-bound collation architecture, including the detailed collation derivation and determination rules
> - COLLATE Operator

Object Values

Object values are compared using one of two comparison functions: `MAP` and `ORDER`. Both functions compare object type instances, but they are quite different from one another. These functions must be specified as part of any object type that will be compared with other object types.

> **See Also:**
>
> CREATE TYPE for a description of `MAP` and `ORDER` methods and the values they return

Varrays and Nested Tables

Comparison of nested tables is described in Comparison Conditions .

Data Type Precedence

Oracle uses data type precedence to determine implicit data type conversion, which is discussed in the section that follows. Oracle data types take the following precedence:

- Datetime and interval data types
- `BINARY_DOUBLE`
- `BINARY_FLOAT`
- `NUMBER`
- Character data types
- All other built-in data types

Data Conversion

Generally an expression cannot contain values of different data types. For example, an expression cannot multiply 5 by 10 and then add 'JAMES'. However, Oracle supports both implicit and explicit conversion of values from one data type to another.

Implicit and Explicit Data Conversion

Oracle recommends that you specify explicit conversions, rather than rely on implicit or automatic conversions, for these reasons:

- SQL statements are easier to understand when you use explicit data type conversion functions.
- Implicit data type conversion can have a negative impact on performance, especially if the data type of a column value is converted to that of a constant rather than the other way around.
- Implicit conversion depends on the context in which it occurs and may not work the same way in every case. For example, implicit conversion from a datetime value to a `VARCHAR2` value may return an unexpected year depending on the value of the `NLS_DATE_FORMAT` parameter.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.
- If implicit data type conversion occurs in an index expression, then Oracle Database might not use the index because it is defined for the pre-conversion data type. This can have a negative impact on performance.

Implicit Data Conversion

Oracle Database automatically converts a value from one data type to another when such a conversion makes sense.

Table 2-8 is a matrix of Oracle implicit conversions. The table shows all possible conversions, without regard to the direction of the conversion or the context in which it is made.

An 'X' in a cell indicates implicit conversion of the data types named in the first column and header row.

Table 2-8 Implicit Type Conversion Matrix

Data Type	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	DATETIME/INTERVAL	NUMBER	BINARY_FLOAT	BINARY_DOUBLE	LONG	RAW	ROWID	CLOB	BLOB	NCLOB
CHAR	--	X	X	X	X	X	X	X	X	X	X	X	X	X	X
VARCHAR2	X	--	X	X	X	X	X	X	X	X	X	X	X	--	X
NCHAR	X	X	--	X	X	X	X	X	X	X	X	X	X	--	X

Table 2-8 (Cont.) Implicit Type Conversion Matrix

Data Type	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	DATETIME/INTERVAL	NUMBER	BINARY_FLOAT	BINARY_DOUBLE	LONG	RAW	ROWID	CLOB	BLOB	NCLOB
NVARCHAR2	X	X	X	--	X	X	X	X	X	X	X	X	X	--	X
DATE	X	X	X	X	--	--	--	--	--	--	--	--	--	--	--
DATETIME/INTERVAL	X	X	X	X	--	--	--	--	--	X	--	--	--	--	--
NUMBER	X	X	X	X	--	--	--	X	X	--	--	--	--	--	--
BINARY_FLOAT	X	X	X	X	--	--	X	--	X	--	--	--	--	--	--
BINARY_DOUBLE	X	X	X	X	--	--	X	X	--	--	--	--	--	--	--
LONG	X	X	X	X	--	X[1]	--	--	--	--	X	--	X	--	X
RAW	X	X	X	X	--	--	--	--	--	X	--	--	--	X	--
ROWID	X	X	X	X	--	--	--	--	--	--	--	--	--	--	--
CLOB	X	X	X	X	--	--	--	--	--	X	--	--	--	--	X
BLOB	--	--	--	--	--	--	--	--	--	--	X	--	--	--	--
NCLOB	X	X	X	X	--	--	--	--	--	X	--	--	X	--	--

1 You cannot convert `LONG` to `INTERVAL` directly, but you can convert `LONG` to `VARCHAR2` using `TO_CHAR(`*`interval`*`)`, and then convert the resulting `VARCHAR2` value to `INTERVAL`.

Implicit Data Type Conversion Rules

- During `INSERT` and `UPDATE` operations, Oracle converts the value to the data type of the affected column.
- During `SELECT FROM` operations, Oracle converts the data from the column to the type of the target variable.
- When manipulating numeric values, Oracle usually adjusts precision and scale to allow for maximum capacity. In such cases, the numeric data type resulting from such operations can differ from the numeric data type found in the underlying tables.
- When comparing a character value with a numeric value, Oracle converts the character data to a numeric value.
- Conversions between character values or `NUMBER` values and floating-point number values can be inexact, because the character types and `NUMBER` use decimal precision to represent the numeric value, and the floating-point numbers use binary precision.

- When converting a CLOB value into a character data type such as VARCHAR2, or converting BLOB to RAW data, if the data to be converted is larger than the target data type, then the database returns an error.
- During conversion from a timestamp value to a DATE value, the fractional seconds portion of the timestamp value is truncated. This behavior differs from earlier releases of Oracle Database, when the fractional seconds portion of the timestamp value was rounded.
- Conversions from BINARY_FLOAT to BINARY_DOUBLE are exact.
- Conversions from BINARY_DOUBLE to BINARY_FLOAT are inexact if the BINARY_DOUBLE value uses more bits of precision that supported by the BINARY_FLOAT.
- When comparing a character value with a DATE value, Oracle converts the character data to DATE.
- When you use a SQL function or operator with an argument of a data type other than the one it accepts, Oracle converts the argument to the accepted data type.
- When making assignments, Oracle converts the value on the right side of the equal sign (=) to the data type of the target of the assignment on the left side.
- During concatenation operations, Oracle converts from noncharacter data types to CHAR or NCHAR.
- During arithmetic operations on and comparisons between character and noncharacter data types, Oracle converts from any character data type to a numeric, date, or rowid, as appropriate. In arithmetic operations between CHAR/VARCHAR2 and NCHAR/NVARCHAR2, Oracle converts to a NUMBER.
- Most SQL character functions are enabled to accept CLOBs as parameters, and Oracle performs implicit conversions between CLOB and character types. Therefore, functions that are not yet enabled for CLOBs can accept CLOBs through implicit conversion. In such cases, Oracle converts the CLOBs to CHAR or VARCHAR2 before the function is invoked. If the CLOB is larger than 4000 bytes, then Oracle converts only the first 4000 bytes to CHAR.
- When converting RAW or LONG RAW data to or from character data, the binary data is represented in hexadecimal form, with one hexadecimal character representing every four bits of RAW data. Refer to "RAW and LONG RAW Data Types " for more information.
- Comparisons between CHAR and VARCHAR2 and between NCHAR and NVARCHAR2 types may entail different character sets. The default direction of conversion in such cases is from the database character set to the national character set. Table 2-9 shows the direction of implicit conversions between different character types.

Table 2-9 Conversion Direction of Different Character Types

Source Data Type	to CHAR	to VARCHAR2	to NCHAR	to NVARCHAR2
from CHAR	--	VARCHAR2	NCHAR	NVARCHAR2
from VARCHAR2	VARCHAR2	--	NVARCHAR2	NVARCHAR2
from NCHAR	NCHAR	NCHAR	--	NVARCHAR2
from NVARCHAR2	NVARCHAR2	NVARCHAR2	NVARCHAR2	--

User-defined types such as collections cannot be implicitly converted, but must be explicitly converted using CAST ... MULTISET.

Implicit Data Conversion Examples

Text Literal Example

The text literal '10' has data type CHAR. Oracle implicitly converts it to the NUMBER data type if it appears in a numeric expression as in the following statement:

```
SELECT salary + '10'
  FROM employees;
```

Character and Number Values Example

When a condition compares a character value and a NUMBER value, Oracle implicitly converts the character value to a NUMBER value, rather than converting the NUMBER value to a character value. In the following statement, Oracle implicitly converts '200' to 200:

```
SELECT last_name
  FROM employees
  WHERE employee_id = '200';
```

Date Example

In the following statement, Oracle implicitly converts '24-JUN-06' to a DATE value using the default date format 'DD-MON-YY':

```
SELECT last_name
  FROM employees
  WHERE hire_date = '24-JUN-06';
```

Explicit Data Conversion

You can explicitly specify data type conversions using SQL conversion functions. Table 2-10 shows SQL functions that explicitly convert a value from one data type to another.

You cannot specify LONG and LONG RAW values in cases in which Oracle can perform implicit data type conversion. For example, LONG and LONG RAW values cannot appear in expressions with functions or operators. Refer to LONG Data Type for information on the limitations on LONG and LONG RAW data types.

Table 2-10 Explicit Type Conversions

Source Data Type	to CHAR, VARCHAR2, NCHAR, NVARCHAR2	to NUMBER	to Datetime/ Interval	to RAW	to ROWID	to LONG, LONG RAW	to CLOB, NCLOB, BLOB	to BINARY_FLOAT	to BINARY_DOUBLE
from CHAR, VARCHAR2, NCHAR, NVARCHAR2	TO_CHAR (char.) TO_NCHAR (char.)	TO_NUMBER	TO_DATE TO_TIMESTAMP TO_TIMESTAMP_TZ TO_YMINTERVAL TO_DSINTERVAL	HEXTORAW	CHARTO=ROWID	--	TO_CLOB TO_NCLOB	TO_BINARY_FLOAT	TO_BINARY_DOUBLE
from NUMBER	TO_CHAR (number) TO_NCHAR (number)	--	TO_DATE NUMTOYM-INTERVAL NUMTODS-INTERVAL	--	--	--	--	TO_BINARY_FLOAT	TO_BINARY_DOUBLE
from Datetime / Interval	TO_CHAR (date) TO_NCHAR (datetime)	--	--	--	--	--	--	--	--
from RAW	RAWTOHEX RAWTONHEX	--	--	--	--	--	TO_BLOB	--	--
from ROWID	ROWIDTOCHAR	--	--	--	--	--	--	--	--
from LONG / LONG RAW	--	--	--	--	--	--	TO_LOB	--	--
from CLOB, NCLOB, BLOB	TO_CHAR TO_NCHAR	--	--	--	--	--	TO_CLOB TO_NCLOB	--	--
from CLOB, NCLOB, BLOB	TO_CHAR TO_NCHAR	--	--	--	--	--	TO_CLOB TO_NCLOB	--	--
from BINARY_FLOAT	TO_CHAR (char.) TO_NCHAR (char.)	TO_NUMBER	--	--	--	--	--	TO_BINARY_FLOAT	TO_BINARY_DOUBLE

Table 2-10 (Cont.) Explicit Type Conversions

Source Data Type	to CHAR, VARCHAR2, NCHAR, NVARCHAR 2	to NUMB ER	to Datetime/ Interval	to RAW	to ROWID	to LO NG, LO NG RA W	to CLOB, NCLOB, BLOB	to BINARY _FLOAT	to BINARY _DOUB LE
from BINARY_D OUBLE	TO_CHAR (char.) TO_NCHAR (char.)	TO_NUM BER	--	--	--	--	--	TO_BINA RY_FLOA T	TO_BINA RY_DOUB LE

See Also:

Conversion Functions for details on all of the explicit conversion functions

Security Considerations for Data Conversion

When a datetime value is converted to text, either by implicit conversion or by explicit conversion that does not specify a format model, the format model is defined by one of the globalization session parameters. Depending on the source data type, the parameter name is `NLS_DATE_FORMAT`, `NLS_TIMESTAMP_FORMAT`, or `NLS_TIMESTAMP_TZ_FORMAT`. The values of these parameters can be specified in the client environment or in an `ALTER SESSION` statement.

The dependency of format models on session parameters can have a negative impact on database security when conversion without an explicit format model is applied to a datetime value that is being concatenated to text of a dynamic SQL statement. Dynamic SQL statements are those statements whose text is concatenated from fragments before being passed to a database for execution. Dynamic SQL is frequently associated with the built-in PL/SQL package `DBMS_SQL` or with the PL/SQL statement `EXECUTE IMMEDIATE`, but these are not the only places where dynamically constructed SQL text may be passed as argument. For example:

```
EXECUTE IMMEDIATE
'SELECT last_name FROM employees WHERE hire_date > ''' || start_date || '''';
```

where *`start_date`* has the data type `DATE`.

In the above example, the value of *`start_date`* is converted to text using a format model specified in the session parameter `NLS_DATE_FORMAT`. The result is concatenated into SQL text. A datetime format model can consist simply of literal text enclosed in double quotation marks. Therefore, any user who can explicitly set globalization parameters for a session can decide what text is produced by the above conversion. If the SQL statement is executed by a PL/SQL procedure, the procedure becomes vulnerable to SQL injection through the session parameter. If the procedure runs with definer's rights, with higher privileges than the session itself, the user can gain unauthorized access to sensitive data.

See Also:

Oracle Database PL/SQL Language Reference for further examples and for recommendations on avoiding this security risk

Note:

This security risk also applies to middle-tier applications that construct SQL text from datetime values converted to text by the database or by OCI datetime functions. Those applications are vulnerable if session globalization parameters are obtained from a user preference.

Implicit and explicit conversion for numeric values may also suffer from the analogous problem, as the conversion result may depend on the session parameter `NLS_NUMERIC_CHARACTERS`. This parameter defines the decimal and group separator characters. If the decimal separator is defined to be the quotation mark or the double quotation mark, some potential for SQL injection emerges.

See Also:

- *Oracle Database Globalization Support Guide* for detailed descriptions of the session globalization parameters
- Format Models for information on the format models

Literals

The terms **literal** and **constant value** are synonymous and refer to a fixed data value. For example, 'JACK', 'BLUE ISLAND', and '101' are all character literals; 5001 is a numeric literal. Character literals are enclosed in single quotation marks so that Oracle can distinguish them from schema object names.

This section contains these topics:

- Text Literals
- Numeric Literals
- Datetime Literals
- Interval Literals

Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions. You can specify character literals with the '*text*' notation, national character literals with the *N'text'* notation, and numeric literals with the *integer*, or *number* notation, depending on the context of the literal. The syntactic forms of these notations appear in the sections that follow.

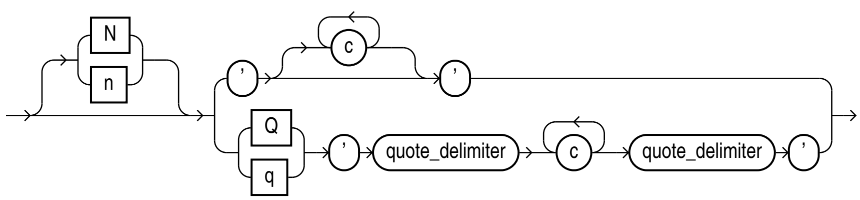
To specify a datetime or interval data type as a literal, you must take into account any optional precisions included in the data types. Examples of specifying datetime and interval data types as literals are provided in the relevant sections of Data Types .

Text Literals

Use the text literal notation to specify values whenever *string* appears in the syntax of expressions, conditions, SQL functions, and SQL statements in other parts of this reference. This reference uses the terms **text literal**, **character literal**, and **string** interchangeably. Text, character, and string literals are always surrounded by single quotation marks. If the syntax uses the term *char*, then you can specify either a text literal or another expression that resolves to character data — for example, the `last_name` column of the `hr.employees` table. When *char* appears in the syntax, the single quotation marks are not used.

The syntax of text literals or strings follows:

***string*::=**



where `N` or `n` specifies the literal using the national character set (`NCHAR` or `NVARCHAR2` data). By default, text entered using this notation is translated into the national character set by way of the database character set when used by the server. To avoid potential loss of data during the text literal conversion to the database character set, set the environment variable `ORA_NCHAR_LITERAL_REPLACE` to `TRUE`. Doing so transparently replaces the `n'` internally and preserves the text literal for SQL processing.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about N-quoted literals

In the top branch of the syntax:

- *c* is any member of the user's character set. A single quotation mark (') within the literal must be preceded by an escape character. To represent one single quotation mark within a literal, enter two single quotation marks.
- ' ' are two single quotation marks that begin and end text literals.

In the bottom branch of the syntax:

- `Q` or `q` indicates that the alternative quoting mechanism will be used. This mechanism allows a wide range of delimiters for the text string.

- The outermost `' '` are two single quotation marks that precede and follow, respectively, the opening and closing *`quote_delimiter`*.
- *`c`* is any member of the user's character set. You can include quotation marks (") in the text literal made up of `c` characters. You can also include the *`quote_delimiter`*, as long as it is not immediately followed by a single quotation mark.
- *`quote_delimiter`* is any single- or multibyte character except space, tab, and return. The *`quote_delimiter`* can be a single quotation mark. However, if the *`quote_delimiter`* appears in the text literal itself, ensure that it is not immediately followed by a single quotation mark.

  If the opening *`quote_delimiter`* is one of `[`, `{`, `<`, or `(`, then the closing *`quote_delimiter`* must be the corresponding `]`, `}`, `>`, or `)`. In all other cases, the opening and closing *`quote_delimiter`* must be the same character.

Text literals have properties of both the `CHAR` and `VARCHAR2` data types:

- Within expressions and conditions, Oracle treats text literals as though they have the data type `CHAR` by comparing them using blank-padded comparison semantics.
- A text literal can have a maximum length of 4000 bytes if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 bytes if `MAX_STRING_SIZE = EXTENDED`. See Extended Data Types for more information.

Here are some valid text literals:

```
'Hello'
'ORACLE.dbs'
'Jackie''s raincoat'
'09-MAR-98'
N'nchar literal'
```

Here are some valid text literals using the alternative quoting mechanism:

```
q'!name LIKE '%DBMS_%%'!'
q'<'So,' she said, 'It's finished.'>'
q'{SELECT * FROM employees WHERE last_name = 'Smith';}'
nq'ï Ÿ1234 ï'
q'"name like '['"'
```

See Also:

Blank-Padded and Nonpadded Comparison Semantics

Numeric Literals

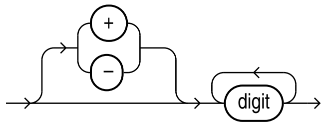
Use numeric literal notation to specify fixed and floating-point numbers.

Integer Literals

You must use the integer notation to specify an integer whenever *`integer`* appears in expressions, conditions, SQL functions, and SQL statements described in other parts of this reference.

The syntax of *`integer`* follows:

***integer*::=**



where *`digit`* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

An integer can store a maximum of 38 digits of precision.

Here are some valid integers:

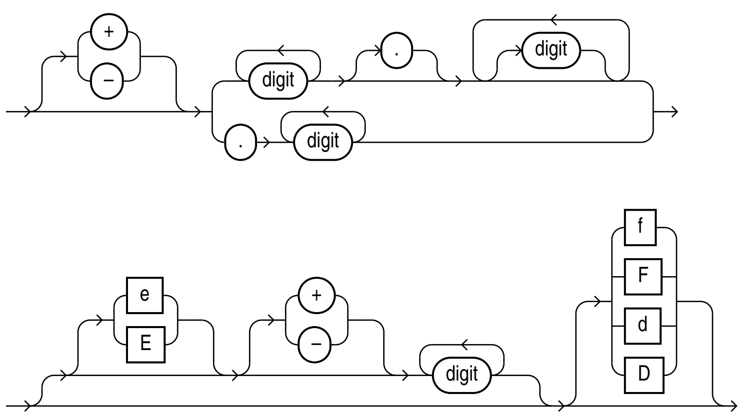
```
7
+255
```

NUMBER and Floating-Point Literals

You must use the number or floating-point notation to specify values whenever *`number`* or *`n`* appears in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of *`number`* follows:

***number*::=**



where

- + or - indicates a positive or negative value. If you omit the sign, then a positive value is the default.
- *`digit`* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
- e or E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from -130 to 125.
- f or F indicates that the number is a 32-bit binary floating point number of type `BINARY_FLOAT`.

- d or D indicates that the number is a 64-bit binary floating point number of type `BINARY_DOUBLE`.

  If you omit f or F and d or D, then the number is of type `NUMBER`.

  The suffixes f (F) and d (D) are supported only in floating-point number literals, not in character strings that are to be converted to `NUMBER`. For example, if Oracle is expecting a `NUMBER` and it encounters the string `'9'`, then it converts the string to the number 9. However, if Oracle encounters the string `'9f'`, then conversion fails and an error is returned.

A number of type `NUMBER` can store a maximum of 38 digits of precision. If the literal requires more precision than provided by `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`, then Oracle truncates the value. If the range of the literal exceeds the range supported by `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`, then Oracle raises an error.

Numeric literals are SQL syntax elements, which are not sensitive to NLS settings. The decimal separator character in numeric literals is always the period (.). However, if a text literal is specified where a numeric value is expected, then the text literal is implicitly converted to a number in an NLS-sensitive way. The decimal separator contained in the text literal must be the one established with the initialization parameter `NLS_NUMERIC_CHARACTERS`. Oracle recommends that you use numeric literals in SQL scripts to make them work independently of the NLS environment.

The following examples illustrate the behavior of decimal separators in numeric literals and text literals. These examples assume that you have established the comma (,) as the NLS decimal separator for the current session with the following statement:

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS=',.';
```

The previous statement also establishes the period (.) as the NLS group separator, but that is irrelevant for these examples.

This example uses the required decimal separator (.) in the numeric literal `1.23` and the established NLS decimal separator (,) in the text literal '`2,34`'. The text literal is converted to the numeric value 2.34, and the output is displayed using commas for the decimal separators.

```
SELECT 2 * 1.23, 3 * '2,34' FROM DUAL;

    2*1.23   3*'2,34'
---------- ----------
      2,46       7,02
```

The next example shows that a comma is not treated as part of a numeric literal. Rather, the comma is treated as the delimiter in a list of two numeric expressions: `2*1` and `23`.

```
SELECT 2 * 1,23 FROM DUAL;

       2*1         23
---------- ----------
         2         23
```

The next example shows that the decimal separator in a text literal must match the NLS decimal separator in order for implicit text-to-number conversion to succeed. The following statement fails because the decimal separator (.) does not match the established NLS decimal separator (,):

```
SELECT 3 * '2.34' FROM DUAL;
           *
```

```
ERROR at line 1:
ORA-01722: invalid number
```

> **See Also:**
>
> ALTER SESSION and *Oracle Database Reference*

Here are some valid `NUMBER` literals:

```
25
+6.34
0.5
25e-03
-1
```

Here are some valid floating-point number literals:

```
25f
+6.34F
0.5d
-1D
```

You can also use the following supplied floating-point literals in situations where a value cannot be expressed as a numeric literal:

Table 2-11 Floating-Point Literals

Literal	Meaning	Example
`binary_float_nan`	A value of type `BINARY_FLOAT` for which the condition `IS NAN` is true	`SELECT COUNT(*) FROM employees WHERE TO_BINARY_FLOAT(commission_pct) != BINARY_FLOAT_NAN;`
`binary_float_infinity`	Single-precision positive infinity	`SELECT COUNT(*) FROM employees WHERE salary < BINARY_FLOAT_INFINITY;`
`binary_double_nan`	A value of type `BINARY_DOUBLE` for which the condition `IS NAN` is true	`SELECT COUNT(*) FROM employees WHERE TO_BINARY_FLOAT(commission_pct) != BINARY_FLOAT_NAN;`
`binary_double_infinity`	Double-precision positive infinity	`SELECT COUNT(*) FROM employees WHERE salary < BINARY_DOUBLE_INFINITY;`

Datetime Literals

Oracle Database supports four datetime data types: `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`.

Date Literals

You can specify a `DATE` value as a string literal, or you can convert a character or numeric value to a date value with the `TO_DATE` function. `DATE` literals are the only case in which Oracle Database accepts a `TO_DATE` expression in place of a string literal.

To specify a `DATE` value as a literal, you must use the Gregorian calendar. You can specify an ANSI literal, as shown in this example:

```
DATE '1998-12-25'
```

The ANSI date literal contains no time portion, and must be specified in the format '`YYYY-MM-DD`'. Alternatively you can specify an Oracle date value, as in the following example:

```
TO_DATE('98-DEC-25 17:30','YY-MON-DD HH24:MI')
```

The default date format for an Oracle `DATE` value is specified by the initialization parameter `NLS_DATE_FORMAT`. This example date format includes a two-digit number for the day of the month, an abbreviation of the month name, the last two digits of the year, and a 24-hour time designation.

Oracle automatically converts character values that are in the default date format into date values when they are used in date expressions.

If you specify a date value without a time component, then the default time is midnight (00:00:00 or 12:00:00 for 24-hour and 12-hour clock time, respectively). If you specify a date value without a date, then the default date is the first day of the current month.

Oracle `DATE` columns always contain both the date and time fields. Therefore, if you query a `DATE` column, then you must either specify the time field in your query or ensure that the time fields in the `DATE` column are set to midnight. Otherwise, Oracle may not return the query results you expect. You can use the `TRUNC` date function to set the time field to midnight, or you can include a greater-than or less-than condition in the query instead of an equality or inequality condition.

Here are some examples that assume a table `my_table` with a number column `row_num` and a `DATE` column `datecol`:

```
INSERT INTO my_table VALUES (1, SYSDATE);
INSERT INTO my_table VALUES (2, TRUNC(SYSDATE));

SELECT *
  FROM my_table;

   ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02

SELECT *
  FROM my_table
  WHERE datecol > TO_DATE('02-OCT-02', 'DD-MON-YY');

   ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02

SELECT *
```

```
  FROM my_table
  WHERE datecol = TO_DATE('03-OCT-02','DD-MON-YY');

   ROW_NUM DATECOL
---------- ---------
         2 03-OCT-02
```

If you know that the time fields of your `DATE` column are set to midnight, then you can query your `DATE` column as shown in the immediately preceding example, or by using the `DATE` literal:

```
SELECT *
  FROM my_table
  WHERE datecol = DATE '2002-10-03';


   ROW_NUM DATECOL
---------- ---------
         2 03-OCT-02
```

However, if the `DATE` column contains values other than midnight, then you must filter out the time fields in the query to get the correct result. For example:

```
SELECT *
  FROM my_table
  WHERE TRUNC(datecol) = DATE '2002-10-03';


   ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02
```

Oracle applies the `TRUNC` function to each row in the query, so performance is better if you ensure the midnight value of the time fields in your data. To ensure that the time fields are set to midnight, use one of the following methods during inserts and updates:

- Use the `TO_DATE` function to mask out the time fields:

  ```
  INSERT INTO my_table
    VALUES (3, TO_DATE('3-OCT-2002','DD-MON-YYYY'));
  ```

- Use the `DATE` literal:

  ```
  INSERT INTO my_table
    VALUES (4, '03-OCT-02');
  ```

- Use the `TRUNC` function:

  ```
  INSERT INTO my_table
    VALUES (5, TRUNC(SYSDATE));
  ```

The date function `SYSDATE` returns the current system date and time. The function `CURRENT_DATE` returns the current session date. For information on `SYSDATE`, the `TO_*` datetime functions, and the default date format, see Datetime Functions .

TIMESTAMP Literals

The `TIMESTAMP` data type stores year, month, day, hour, minute, and second, and fractional second values. When you specify `TIMESTAMP` as a literal, the *`fractional_seconds_precision`* value can be any number of digits up to 9, as follows:

```
TIMESTAMP '1997-01-31 09:26:50.124'
```

TIMESTAMP WITH TIME ZONE Literals

The `TIMESTAMP WITH TIME ZONE` data type is a variant of `TIMESTAMP` that includes a time zone region name or time zone offset. When you specify `TIMESTAMP WITH TIME ZONE` as a literal, the *`fractional_seconds_precision`* value can be any number of digits up to 9. For example:

```
TIMESTAMP '1997-01-31 09:26:56.66 +02:00'
```

Two `TIMESTAMP WITH TIME ZONE` values are considered identical if they represent the same instant in UTC, regardless of the `TIME ZONE` offsets stored in the data. For example,

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '1999-04-15 11:00:00 -5:00'
```

8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

You can replace the UTC offset with the `TZR` (time zone region name) format element. For example, the following example has the same value as the preceding example:

```
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
```

To eliminate the ambiguity of boundary cases when the daylight saving time switches, use both the `TZR` and a corresponding `TZD` format element. The following example ensures that the preceding example will return a daylight saving time value:

```
TIMESTAMP '1999-10-29 01:30:00 US/Pacific PDT'
```

You can also express the time zone offset using a datetime expression:

```
SELECT TIMESTAMP '2009-10-29 01:30:00' AT TIME ZONE 'US/Pacific'
  FROM DUAL;
```

> **See Also:**
>
> Datetime Expressions for more information

If you do not add the `TZD` format element, and the datetime value is ambiguous, then Oracle returns an error if you have the `ERROR_ON_OVERLAP_TIME` session parameter set to `TRUE`. If that parameter is set to `FALSE`, then Oracle interprets the ambiguous datetime as standard time in the specified region.

TIMESTAMP WITH LOCAL TIME ZONE Literals

The `TIMESTAMP WITH LOCAL TIME ZONE` data type differs from `TIMESTAMP WITH TIME ZONE` in that data stored in the database is normalized to the database time zone. The time zone offset is not stored as part of the column data. There is no literal for `TIMESTAMP WITH LOCAL TIME ZONE`. Rather, you represent values of this data type using any of the other valid datetime literals. The table that follows shows some of the formats you can use to insert a value into a `TIMESTAMP WITH LOCAL TIME ZONE` column, along with the corresponding value returned by a query.

Table 2-12 TIMESTAMP WITH LOCAL TIME ZONE Literals

Value Specified in INSERT Statement	Value Returned by Query
`'19-FEB-2004'`	19-FEB-2004.00.00.000000 AM
`SYSTIMESTAMP`	19-FEB-04 02.54.36.497659 PM
`TO_TIMESTAMP('19-FEB-2004', 'DD-MON-YYYY')`	19-FEB-04 12.00.00.000000 AM
`SYSDATE`	19-FEB-04 02.55.29.000000 PM
`TO_DATE('19-FEB-2004', 'DD-MON-YYYY')`	19-FEB-04 12.00.00.000000 AM
`TIMESTAMP'2004-02-19 8:00:00 US/Pacific'`	19-FEB-04 08.00.00.000000 AM

Notice that if the value specified does not include a time component (either explicitly or implicitly), then the value returned defaults to midnight.

Interval Literals

An interval literal specifies a period of time. You can specify these differences in terms of years and months, or in terms of days, hours, minutes, and seconds. Oracle Database supports two types of interval literals, `YEAR TO MONTH` and `DAY TO SECOND`. Each type contains a leading field and may contain a trailing field. The leading field defines the basic unit of date or time being measured. The trailing field defines the smallest increment of the basic unit being considered. For example, a `YEAR TO MONTH` interval considers an interval of years to the nearest month. A `DAY TO MINUTE` interval considers an interval of days to the nearest minute.

If you have date data in numeric form, then you can use the `NUMTOYMINTERVAL` or `NUMTODSINTERVAL` conversion function to convert the numeric data into interval values.

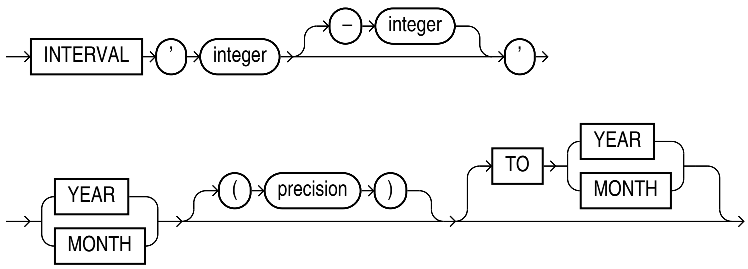
Interval literals are used primarily with analytic functions.

> **See Also:**
>
> Analytic Functions , NUMTODSINTERVAL , and NUMTOYMINTERVAL

INTERVAL YEAR TO MONTH

Specify `YEAR TO MONTH` interval literals using the following syntax:

***interval_year_to_month*::=**



where

- *'integer [-integer]'* specifies integer values for the leading and optional trailing field of the literal. If the leading field is `YEAR` and the trailing field is `MONTH`, then the range of integer values for the month field is 0 to 11.
- *precision* is the maximum number of digits in the leading field. The valid range of the leading field precision is 0 to 9 and its default value is 2.

Restriction on the Leading Field

If you specify a trailing field, then it must be less significant than the leading field. For example, `INTERVAL '0-1' MONTH TO YEAR` is not valid.

The following `INTERVAL YEAR TO MONTH` literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

Examples of the other forms of the literal follow, including some abbreviated versions:

Table 2-13 Forms of INTERVAL YEAR TO MONTH Literals

Form of Interval Literal	Interpretation
`INTERVAL '123-2' YEAR(3) TO MONTH`	An interval of 123 years, 2 months. You must specify the leading field precision if it is greater than the default of 2 digits.
`INTERVAL '123' YEAR(3)`	An interval of 123 years 0 months.
`INTERVAL '300' MONTH(3)`	An interval of 300 months.
`INTERVAL '4' YEAR`	Maps to `INTERVAL '4-0' YEAR TO MONTH` and indicates 4 years.
`INTERVAL '50' MONTH`	Maps to `INTERVAL '4-2' YEAR TO MONTH` and indicates 50 months or 4 years 2 months.
`INTERVAL '123' YEAR`	Returns an error, because the default precision is 2, and '123' has 3 digits.

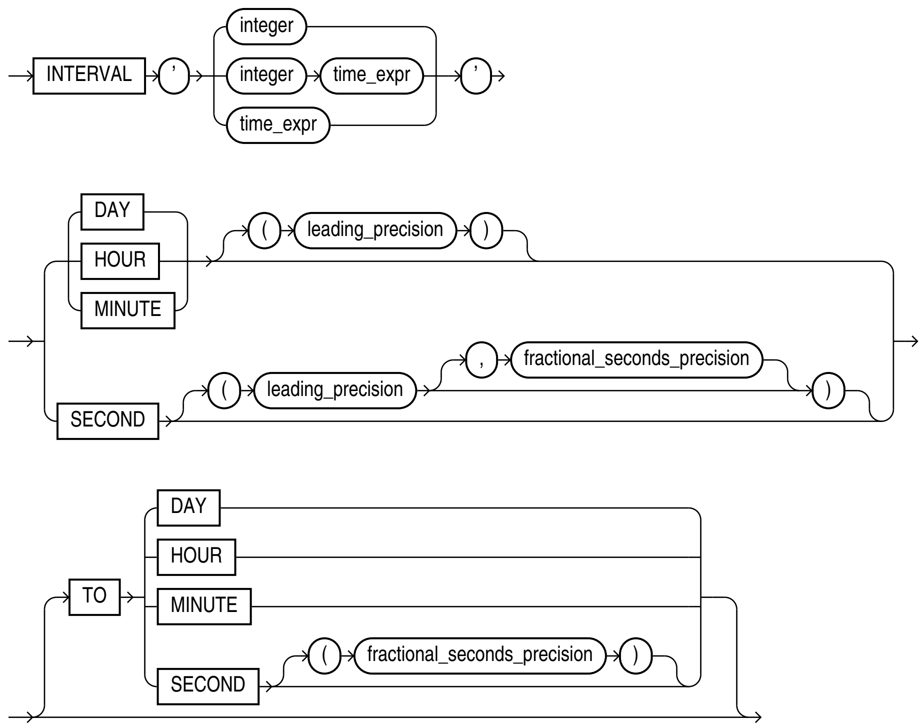
You can add or subtract one `INTERVAL YEAR TO MONTH` literal to or from another to yield another `INTERVAL YEAR TO MONTH` literal. For example:

```
INTERVAL '5-3' YEAR TO MONTH + INTERVAL'20' MONTH =
INTERVAL '6-11' YEAR TO MONTH
```

INTERVAL DAY TO SECOND

Specify `DAY TO SECOND` interval literals using the following syntax:

***interval_day_to_second*::=**



where

- *`integer`* specifies the number of days. If this value contains more digits than the number specified by the leading precision, then Oracle returns an error.
- *`time_expr`* specifies a time in the format *`HH`*`[:`*`MI`*`[:`*`SS`*`[.`*`n`*`]]]` or *`MI`*`[:`*`SS`*`[.`*`n`*`]]` or *`SS`*`[.`*`n`*`]`, where *`n`* specifies the fractional part of a second. If *`n`* contains more digits than the number specified by *`fractional_seconds_precision`*, then *`n`* is rounded to the number of digits specified by the *`fractional_seconds_precision`* value. You can specify *`time_expr`* following an integer and a space only if the leading field is `DAY`.
- *`leading_precision`* is the number of digits in the leading field. Accepted values are 0 to 9. The default is 2.
- *`fractional_seconds_precision`* is the number of digits in the fractional part of the `SECOND` datetime field. Accepted values are 1 to 9. The default is 6.

Restriction on the Leading Field:

If you specify a trailing field, then it must be less significant than the leading field. For example, `INTERVAL MINUTE TO DAY` is not valid. As a result of this restriction, if `SECOND` is the leading field, the interval literal cannot have any trailing field.

The valid range of values for the trailing field are as follows:

- `HOUR`: 0 to 23
- `MINUTE`: 0 to 59
- `SECOND`: 0 to 59.999999999

Examples of the various forms of `INTERVAL DAY TO SECOND` literals follow, including some abbreviated versions:

Table 2-14 Forms of INTERVAL DAY TO SECOND Literals

Form of Interval Literal	Interpretation
`INTERVAL '4 5:12:10.222' DAY TO SECOND(3)`	4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.
`INTERVAL '4 5:12' DAY TO MINUTE`	4 days, 5 hours and 12 minutes.
`INTERVAL '400 5' DAY(3) TO HOUR`	400 days 5 hours.
`INTERVAL '400' DAY(3)`	400 days.
`INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)`	11 hours, 12 minutes, and 10.2222222 seconds.
`INTERVAL '11:20' HOUR TO MINUTE`	11 hours and 20 minutes.
`INTERVAL '10' HOUR`	10 hours.
`INTERVAL '10:22' MINUTE TO SECOND`	10 minutes 22 seconds.
`INTERVAL '10' MINUTE`	10 minutes.
`INTERVAL '4' DAY`	4 days.
`INTERVAL '25' HOUR`	25 hours.
`INTERVAL '40' MINUTE`	40 minutes.
`INTERVAL '120' HOUR(3)`	120 hours.
`INTERVAL '30.12345' SECOND(2,4)`	30.1235 seconds. The fractional second '12345' is rounded to '1235' because the precision is 4.

You can add or subtract one `DAY TO SECOND` interval literal from another `DAY TO SECOND` literal. For example.

```
INTERVAL'20' DAY - INTERVAL'240' HOUR = INTERVAL'10-0' DAY TO SECOND
```

Format Models

A **format model** is a character literal that describes the format of datetime or numeric data stored in a character string. A format model does not change the internal representation of the value in the database. When you convert a character string into a date or number, a format model determines how Oracle Database interprets the string. In SQL statements, you can use a format model as an argument of the `TO_CHAR` and `TO_DATE` functions to specify:

- The format for Oracle to use to return a value from the database
- The format for a value you have specified for Oracle to store in the database

For example:

- The datetime format model for the string '`17:45:29`' is '`HH24:MI:SS`'.
- The datetime format model for the string '`11-Nov-1999`' is '`DD-Mon-YYYY`'.

- The number format model for the string '$2,304.25' is '$9,999.99'.

For lists of number and datetime format model elements, see Table 2-15 and Table 2-17.

The values of some formats are determined by the value of initialization parameters. For such formats, you can specify the characters returned by these format elements implicitly using the initialization parameter NLS_TERRITORY. You can change the default date format for your session with the ALTER SESSION statement.

> **See Also:**
>
> - ALTER SESSION for information on changing the values of these parameters and Format Model Examples for examples of using format models
> - TO_CHAR (datetime) , TO_CHAR (number) , and TO_DATE
> - *Oracle Database Reference* and *Oracle Database Globalization Support Guide* for information on these parameters

This remainder of this section describes how to use the following format models:

- Number Format Models
- Datetime Format Models
- Format Model Modifiers

Number Format Models

You can use number format models in the following functions:

- In the TO_CHAR function to translate a value of NUMBER, BINARY_FLOAT, or BINARY_DOUBLE data type to VARCHAR2 data type
- In the TO_NUMBER function to translate a value of CHAR or VARCHAR2 data type to NUMBER data type
- In the TO_BINARY_FLOAT and TO_BINARY_DOUBLE functions to translate CHAR and VARCHAR2 expressions to BINARY_FLOAT or BINARY_DOUBLE values

All number format models cause the number to be rounded to the specified number of significant digits. If a value has more significant digits to the left of the decimal place than are specified in the format, then pound signs (#) replace the value. This event typically occurs when you are using TO_CHAR with a restrictive number format string, causing a rounding operation.

- If a positive NUMBER value is extremely large and cannot be represented in the specified format, then the infinity sign (~) replaces the value. Likewise, if a negative NUMBER value is extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~).
- If a BINARY_FLOAT or BINARY_DOUBLE value is converted to CHAR or NCHAR, and the input is either infinity or NaN (not a number), then Oracle always returns the pound

signs to replace the value. However, if you omit the format model, then Oracle returns either `Inf` or `Nan` as a string.

Number Format Elements

A number format model is composed of one or more number format elements. The tables that follow list the elements of a number format model and provide some examples.

Negative return values automatically contain a leading negative sign and positive values automatically contain a leading space unless the format model contains the `MI`, `S`, or `PR` format element.

Table 2-15 Number Format Elements

Element	Example	Description
, (comma)	`9,999`	Returns a comma in the specified position. You can specify multiple commas in a number format model. **Restrictions:** • A comma element cannot begin a number format model. • A comma cannot appear to the right of a decimal character or period in a number format model.
. (period)	`99.99`	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** You can specify only one period in a number format model.
$	`$9999`	Returns value with a leading dollar sign.
0	`0999` `9990`	Returns leading zeros. Returns trailing zeros.
9	`9999`	Returns value with the specified number of digits with a leading space if positive or with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
B	`B9999`	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of zeros in the format model).
C	`C999`	Returns in the specified position the ISO currency symbol (the current value of the `NLS_ISO_CURRENCY` parameter).
D	`99D99`	Returns in the specified position the decimal character, which is the current value of the `NLS_NUMERIC_CHARACTER` parameter. The default is a period (.). **Restriction:** You can specify only one decimal character in a number format model.
EEEE	`9.9EEEE`	Returns a value using in scientific notation.
G	`9G999`	Returns in the specified position the group separator (the current value of the `NLS_NUMERIC_CHARACTER` parameter). You can specify multiple group separators in a number format model. **Restriction:** A group separator cannot appear to the right of a decimal character or period in a number format model.
L	`L999`	Returns in the specified position the local currency symbol (the current value of the `NLS_CURRENCY` parameter).
MI	`9999MI`	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing blank. **Restriction:** The MI format element can appear only in the last position of a number format model.

Table 2-15 (Cont.) Number Format Elements

Element	Example	Description
PR	`9999PR`	Returns negative value in <angle brackets>. Returns positive value with a leading and trailing blank. **Restriction:** The PR format element can appear only in the last position of a number format model.
RN rn	`RN` `rn`	Returns a value as Roman numerals in uppercase. Returns a value as Roman numerals in lowercase. Value can be an integer between 1 and 3999.
S	`S9999` `9999S`	Returns negative value with a leading minus sign (-). Returns positive value with a leading plus sign (+). Returns negative value with a trailing minus sign (-). Returns positive value with a trailing plus sign (+). **Restriction:** The S format element can appear only in the first or last position of a number format model.
TM	`TM`	The text minimum number format model returns (in decimal output) the smallest number of characters possible. This element is case insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If the output exceeds 64 characters, then Oracle Database automatically returns the number in scientific notation. **Restrictions:** • You cannot precede this element with any other element. • You can follow this element only with one 9 or one E (or e), but not with any combination of these. The following statement returns an error: `SELECT TO_CHAR(1234, 'TM9e') FROM DUAL;`
U	`U9999`	Returns in the specified position the Euro (or other) dual currency symbol, determined by the current value of the `NLS_DUAL_CURRENCY` parameter.
V	`999V99`	Returns a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the `V`.
X	`XXXX` `xxxx`	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, then Oracle Database rounds it to an integer. **Restrictions:** • This element accepts only positive values or 0. Negative values return an error. • You can precede this element only with 0 (which returns leading zeroes) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, then the return always has one leading blank. Refer to the format model modifier FM for more information.

Table 2-16 shows the results of the following query for different values of *number* and '*fmt*':

```
SELECT TO_CHAR(number, 'fmt')
  FROM DUAL;
```

Table 2-16 Results of Number Conversions

number	'fmt'	Result
`-1234567890`	`9999999999S`	`'1234567890-'`

Table 2-16 (Cont.) Results of Number Conversions

number	'fmt'	Result
0	99.99	' .00'
+0.1	99.99	' .10'
-0.2	99.99	' -.20'
0	90.99	' 0.00'
+0.1	90.99	' 0.10'
-0.2	90.99	' -0.20'
0	9999	' 0'
1	9999	' 1'
0	B9999	' '
1	B9999	' 1'
0	B90.99	' '
+123.456	999.999	' 123.456'
-123.456	999.999	'-123.456'
+123.456	FM999.009	'123.456'
+123.456	9.9EEEE	' 1.2E+02'
+1E+123	9.9EEEE	' 1.0E+123'
+123.456	FM9.9EEEE	'1.2E+02'
+123.45	FM999.009	'123.45'
+123.0	FM999.009	'123.00'
+123.45	L999.99	' $123.45'
+123.45	FML999.99	'$123.45'
+1234567890	9999999999S	'1234567890+'

Datetime Format Models

You can use datetime format models in the following functions:

- In the TO_* datetime functions to translate a character value that is in a format other than the default format into a datetime value. (The TO_* datetime functions are TO_DATE, TO_TIMESTAMP, and TO_TIMESTAMP_TZ.)
- In the TO_CHAR function to translate a datetime value into a character value that is in a format other than the default format (for example, to print the date from an application)

The total length of a datetime format model cannot exceed 22 characters.

The default datetime formats are specified either explicitly with the NLS session parameters NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT, and NLS_TIMESTAMP_TZ_FORMAT, or implicitly with the NLS session parameter NLS_TERRITORY. You can change the default datetime formats for your session with the ALTER SESSION statement.

See Also:

ALTER SESSION and *Oracle Database Globalization Support Guide* for information on the NLS parameters

Datetime Format Elements

A datetime format model is composed of one or more datetime format elements as listed in Table 2-17.

- For input format models, format items cannot appear twice, and format items that represent similar information cannot be combined. For example, you cannot use 'SYYYY' and 'BC' in the same format string.
- The second column indicates whether the format element can be used in the `TO_*` datetime functions. All format elements can be used in the `TO_CHAR` function.
- The following datetime format elements can be used in timestamp and interval format models, but not in the original `DATE` format model: `FF`, `TZD`, `TZH`, `TZM`, and `TZR`.
- Many datetime format elements are padded with blanks or leading zeroes to a specific length. Refer to the format model modifier FM for more information.

Note:

Oracle recommends that you use the 4-digit year element (`YYYY`) instead of the shorter year elements for these reasons:

- The 4-digit year element eliminates ambiguity.
- The shorter year elements may affect query optimization because the year is not known at query compile time and can only be determined at run time.

Uppercase Letters in Date Format Elements

Capitalization in a spelled-out word, abbreviation, or Roman numeral follows capitalization in the corresponding format element. For example, the date format model 'DAY' produces capitalized words like 'MONDAY'; 'Day' produces 'Monday'; and 'day' produces 'monday'.

Punctuation and Character Literals in Datetime Format Models

You can include these characters in a date format model:

- Punctuation such as hyphens, slashes, commas, periods, and colons
- Character literals, enclosed in double quotation marks

These characters appear in the return value in the same location as they appear in the format model.

Table 2-17 Datetime Format Elements

Element	TO_* datetime functions?	Description

`-`
`/`
`,`
`.`
`;`
`:`
`"text"`
	Yes	Punctuation and quoted text is reproduced in the result.

`AD` `A.D.`	Yes	AD indicator with or without periods.
`AM` `A.M.`	Yes	Meridian indicator with or without periods.
`BC` `B.C.`	Yes	BC indicator with or without periods.
`CC` `SCC`		Century. • If the last 2 digits of a 4-digit year are between 01 and 99 (inclusive), then the century is one greater than the first 2 digits of that year. • If the last 2 digits of a 4-digit year are 00, then the century is the same as the first 2 digits of that year. For example, 2002 returns 21; 2000 returns 20.
`D`	Yes	Day of week (1-7). This element depends on the NLS territory of the session.
`DAY`	Yes	Name of day.
`DD`	Yes	Day of month (1-31).
`DDD`	Yes	Day of year (1-366).
`DL`	Yes	Returns a value in the long date format, which is an extension of the Oracle Database `DATE` format, determined by the current value of the `NLS_DATE_FORMAT` parameter. Makes the appearance of the date components (day name, month number, and so forth) depend on the `NLS_TERRITORY` and `NLS_LANGUAGE` parameters. For example, in the `AMERICAN_AMERICA` locale, this is equivalent to specifying the format `'fmDay, Month dd, yyyy'`. In the `GERMAN_GERMANY` locale, it is equivalent to specifying the format '`fmDay, dd. Month yyyy`'. **Restriction:** You can specify this format only with the `TS` element, separated by white space.

Table 2-17 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
DS	Yes	Returns a value in the short date format. Makes the appearance of the date components (day name, month number, and so forth) depend on the NLS_TERRITORY and NLS_LANGUAGE parameters. For example, in the AMERICAN_AMERICA locale, this is equivalent to specifying the format 'MM/DD/RRRR'. In the ENGLISH_UNITED_KINGDOM locale, it is equivalent to specifying the format 'DD/MM/RRRR'. **Restriction:** You can specify this format only with the TS element, separated by white space.
DY	Yes	Abbreviated name of day.
E	Yes	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
EE	Yes	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
FF [1..9]	Yes	Fractional seconds; no radix character is printed. Use the X format element to add the radix character. Use the numbers 1 to 9 after FF to specify the number of digits in the fractional second portion of the datetime value returned. If you do not specify a digit, then Oracle Database uses the precision specified for the datetime data type or the data type's default precision. Valid in timestamp and interval formats, but not in DATE formats. **Examples:** 'HH:MI:SS.FF' SELECT TO_CHAR(SYSTIMESTAMP, 'SS.FF3') from DUAL;
FM	Yes	Returns a value with no leading or trailing blanks. **See Also**: FM
FX	Yes	Requires exact matching between the character data and the format model. **See Also**: FX
HH HH12	Yes	Hour of day (1-12).
HH24	Yes	Hour of day (0-23).
IW		Calendar week of year (1-52 or 1-53), as defined by the ISO 8601 standard. • A calendar week starts on Monday. • The first calendar week of the year includes January 4. • The first calendar week of the year may include December 29, 30 and 31. • The last calendar week of the year may include January 1, 2, and 3.
IYYY		4-digit year of the year containing the calendar week, as defined by the ISO 8601 standard.
IYY IY I		Last 3, 2, or 1 digit(s) of the year containing the calendar week, as defined by the ISO 8601 standard.

Table 2-17 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
J	Yes	Julian day; the number of days since January 1, 4712 BC. Number specified with J must be integers.
MI	Yes	Minute (0-59).
MM	Yes	Month (01-12; January = 01).
MON	Yes	Abbreviated name of month.
MONTH	Yes	Name of month.
PM P.M.	Yes	Meridian indicator with or without periods.
Q		Quarter of year (1, 2, 3, 4; January - March = 1).
RM	Yes	Roman numeral month (I-XII; January = I).
RR	Yes	Lets you store 20th century dates in the 21st century using only two digits. **See Also:** The RR Datetime Format Element
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you do not want this functionality, then enter the 4-digit year.
SS	Yes	Second (0-59).
SSSSS	Yes	Seconds past midnight (0-86399).
TS	Yes	Returns a value in the short time format. Makes the appearance of the time components (hour, minutes, and so forth) depend on the NLS_TERRITORY and NLS_LANGUAGE initialization parameters. **Restriction:** You can specify this format only with the DL or DS element, separated by white space.
TZD	Yes	Daylight saving information. The TZD value is an abbreviated time zone string with daylight saving information. It must correspond with the region specified in TZR. Valid in timestamp and interval formats, but not in DATE formats. **Example:** PST (for US/Pacific standard time); PDT (for US/Pacific daylight time).
TZH	Yes	Time zone hour. (See TZM format element.) Valid in timestamp and interval formats, but not in DATE formats. **Example:** 'HH:MI:SS.FFTZH:TZM'.
TZM	Yes	Time zone minute. (See TZH format element.) Valid in timestamp and interval formats, but not in DATE formats. **Example:** 'HH:MI:SS.FFTZH:TZM'.

Table 2-17 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
`TZR`	Yes	Time zone region information. The value must be one of the time zone region names supported in the database. Valid in timestamp and interval formats, but not in `DATE` formats. **Example:** US/Pacific
`WW`		Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
`W`		Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
`X`	Yes	Local radix character. **Example:** `'HH:MI:SSXFF'`.
`Y,YYY`	Yes	Year with comma in this position.
`YEAR` `SYEAR`		Year, spelled out; `S` prefixes BC dates with a minus sign (-).
`YYYY` `SYYYY`	Yes	4-digit year; `S` prefixes BC dates with a minus sign.

`YYY`
`YY`
`Y`
	Yes	Last 3, 2, or 1 digit(s) of year.

Oracle Database converts strings to dates with some flexibility. For example, when the `TO_DATE` function is used, a format model containing punctuation characters matches an input string lacking some or all of these characters, provided each numerical element in the input string contains the maximum allowed number of digits—for example, two digits '05' for 'MM' or four digits '2007' for 'YYYY'. The following statement does not return an error:

```
SELECT TO_CHAR(TO_DATE('0207','MM/YY'), 'MM/YY') FROM DUAL;

TO_CH
-----
02/07
```

However, the following format string does return an error, because the FX (format exact) format modifier requires an exact match of the expression and the format string:

```
SELECT TO_CHAR(TO_DATE('0207', 'fxmm/yy'), 'mm/yy') FROM DUAL;
SELECT TO_CHAR(TO_DATE('0207', 'fxmm/yy'), 'mm/yy') FROM DUAL;
                        *
ERROR at line 1:
ORA-01861: literal does not match format string
```

Any non-alphanumeric character is allowed to match the punctuation characters in the format model. For example, the following statement does not return an error:

```
SELECT TO_CHAR (TO_DATE('02#07','MM/YY'), 'MM/YY') FROM DUAL;

TO_CH
-----
02/07
```

> **See Also:**
>
> Format Model Modifiers and String-to-Date Conversion Rules for more information

Datetime Format Elements and Globalization Support

The functionality of some datetime format elements depends on the country and language in which you are using Oracle Database. For example, these datetime format elements return spelled values:

- MONTH
- MON
- DAY
- DY
- BC or AD or B.C. or A.D.
- AM or PM or A.M or P.M.

The language in which these values are returned is specified either explicitly with the initialization parameter `NLS_DATE_LANGUAGE` or implicitly with the initialization parameter `NLS_LANGUAGE`. The values returned by the `YEAR` and `SYEAR` datetime format elements are always in English.

The datetime format element `D` returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter `NLS_TERRITORY`.

> **See Also:**
>
> *Oracle Database Reference* and *Oracle Database Globalization Support Guide* for information on globalization support initialization parameters

ISO Standard Date Format Elements

Oracle calculates the values returned by the datetime format elements IYYY, IYY, IY, I, and IW according to the ISO standard. For information on the differences between these values and those returned by the datetime format elements YYYY, YYY, YY, Y, and WW, see the discussion of globalization support in *Oracle Database Globalization Support Guide*.

The RR Datetime Format Element

The `RR` datetime format element is similar to the `YY` datetime format element, but it provides additional flexibility for storing date values in other centuries. The `RR` datetime format element lets you store 20th century dates in the 21st century by specifying only the last two digits of the year.

If you use the `TO_DATE` function with the `YY` datetime format element, then the year returned always has the same first 2 digits as the current year. If you use the `RR` datetime format element instead, then the century of the return value varies according to the specified two-digit year and the last two digits of the current year.

That is:

- If the specified two-digit year is 00 to 49, then
  - If the last two digits of the current year are 00 to 49, then the returned year has the same first two digits as the current year.
  - If the last two digits of the current year are 50 to 99, then the first 2 digits of the returned year are 1 greater than the first 2 digits of the current year.
- If the specified two-digit year is 50 to 99, then
  - If the last two digits of the current year are 00 to 49, then the first 2 digits of the returned year are 1 less than the first 2 digits of the current year.
  - If the last two digits of the current year are 50 to 99, then the returned year has the same first two digits as the current year.

The following examples demonstrate the behavior of the RR datetime format element.

RR Datetime Format Examples

Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY') "Year" FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY') "Year" FROM DUAL;

Year
----
2017
```

Now assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY') "Year" FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY') "Year" FROM DUAL;

Year
```

```
----
2017
```

Note that the queries return the same values regardless of whether they are issued before or after the year 2000. The RR datetime format element lets you write SQL statements that will return the same values from years whose first two digits are different.

Datetime Format Element Suffixes

Table 2-18 lists suffixes that can be added to datetime format elements:

Table 2-18 Date Format Element Suffixes

Suffix	Meaning	Example Element	Example Value
TH	Ordinal Number	DDTH	4TH
SP	Spelled Number	DDSP	FOUR
SPTH or THSP	Spelled, ordinal number	DDSPTH	FOURTH

Notes on date format element suffixes:

- When you add one of these suffixes to a datetime format element, the return value is always in English.
- Datetime suffixes are valid only to format output. You cannot use them to insert a date into the database.

Format Model Modifiers

The FM and FX modifiers, used in format models in the TO_CHAR function, control blank padding and exact format checking.

A modifier can appear in a format model more than once. In such a case, each subsequent occurrence toggles the effects of the modifier. Its effects are enabled for the portion of the model following its first occurrence, and then disabled for the portion following its second, and then reenabled for the portion following its third, and so on.

FM

Fill mode. Oracle uses trailing blank characters and leading zeroes to fill format elements to a constant width. The width is equal to the display width of the largest element for the relevant format model:

- Numeric elements are padded with leading zeros to the width of the maximum value allowed for the element. For example, the YYYY element is padded to four digits (the length of '9999'), HH24 to two digits (the length of '23'), and DDD to three digits (the length of '366').
- The character elements MONTH, MON, DAY, and DY are padded with trailing blanks to the width of the longest full month name, the longest abbreviated month name, the longest full date name, or the longest abbreviated day name, respectively, among valid names determined by the values of NLS_DATE_LANGUAGE and NLS_CALENDAR parameters. For example, when NLS_DATE_LANGUAGE is AMERICAN and NLS_CALENDAR is GREGORIAN (the default), the largest element for MONTH is SEPTEMBER, so all values of the MONTH format element are padded to nine display characters. The values of the NLS_DATE_LANGUAGE

and NLS_CALENDAR parameters are specified in the third argument to TO_CHAR and TO_* datetime functions or they are retrieved from the NLS environment of the current session.

- The character element RM is padded with trailing blanks to the length of 4, which is the length of 'viii'.
- Other character elements and spelled-out numbers (SP, SPTH, and THSP suffixes) are not padded.

The FM modifier suppresses the above padding in the return value of the TO_CHAR function.

FX

Format exact. This modifier specifies exact matching for the character argument and datetime format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without FX, Oracle ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without FX, numbers in the character argument can omit leading zeros.

  When FX is enabled, you can disable this check for leading zeros by using the FM modifier as well.

If any portion of the character argument violates any of these conditions, then Oracle returns an error message.

Format Model Examples

The following statement uses a date format model to return a character expression:

```
SELECT TO_CHAR(SYSDATE, 'fmDDTH') || ' of ' ||
       TO_CHAR(SYSDATE, 'fmMonth') || ', ' ||
       TO_CHAR(SYSDATE, 'YYYY') "Ides"
  FROM DUAL;

Ides
------------------
3RD of April, 2008
```

The preceding statement also uses the FM modifier. If FM is omitted, then the month is blank-padded to nine characters:

```
SELECT TO_CHAR(SYSDATE, 'DDTH') || ' of ' ||
   TO_CHAR(SYSDATE, 'Month') || ', ' ||
   TO_CHAR(SYSDATE, 'YYYY') "Ides"
  FROM DUAL;

Ides
-----------------------
03RD of April    , 2008
```

The following statement places a single quotation mark in the return value by using a date format model that includes two consecutive single quotation marks:

```
SELECT TO_CHAR(SYSDATE, 'fmDay') || '''s Special' "Menu"
  FROM DUAL;

Menu
-----------------
Tuesday's Special
```

Two consecutive single quotation marks can be used for the same purpose within a character literal in a format model.

Table 2-19 shows whether the following statement meets the matching conditions for different values of *char* and '*fmt*' using FX (the table named table has a column date_column of data type DATE):

```
UPDATE table
  SET date_column = TO_DATE(char, 'fmt');
```

Table 2-19 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1998'	'DD-MON-YYYY'	Match
' 15! JAN % /1998'	'DD-MON-YYYY'	Error
'15/JAN/1998'	'FXDD-MON-YYYY'	Error
'15-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXDD-MON-YYYY'	Error
'01-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXFMDD-MON-YYYY'	Match

Format of Return Values: Examples

You can use a format model to specify the format for Oracle to use to return values from the database to you.

The following statement selects the salaries of the employees in Department 80 and uses the TO_CHAR function to convert these salaries into character values with the format specified by the number format model '$99,990.99':

```
SELECT last_name employee, TO_CHAR(salary, '$99,990.99')
  FROM employees
  WHERE department_id = 80;
```

Because of this format model, Oracle returns salaries with leading dollar signs, commas every three digits, and two decimal places.

The following statement selects the date on which each employee from Department 20 was hired and uses the TO_CHAR function to convert these dates to character strings with the format specified by the date format model 'fmMonth DD, YYYY':

```
SELECT last_name employee, TO_CHAR(hire_date,'fmMonth DD, YYYY') hiredate
  FROM employees
  WHERE department_id = 20;
```

With this format model, Oracle returns the hire dates without blank padding (as specified by fm), two digits for the day, and the century included in the year.

> **See Also:**
>
> Format Model Modifiers for a description of the fm format element

Supplying the Correct Format Model: Examples

When you insert or update a column value, the data type of the value that you specify must correspond to the column data type of the column. You can use format models to specify the format of a value that you are converting from one data type to another data type required for a column.

For example, a value that you insert into a DATE column must be a value of the DATE data type or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE data type). If the value is in another format, then you must use the TO_DATE function to convert the value to the DATE data type. You must also use a format model to specify the format of the character string.

The following statement updates Hunold's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '2008 05 20' to a DATE value:

```
UPDATE employees
  SET hire_date = TO_DATE('2008 05 20','YYYY MM DD')
  WHERE last_name = 'Hunold';
```

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values (unless you have used the FX or FXFM modifiers in the format model to control exact format checking):

- You can omit punctuation included in the format string from the date string if all the digits of the numerical format elements, including leading zeros, are specified. For example, specify 02 and not 2 for two-digit format elements such as MM, DD, and YY.
- You can omit time fields found at the end of a format string from the date string.
- You can use any non-alphanumeric character in the date string to match the punctuation symbol in the format string.
- If a match fails between a datetime format element and the corresponding characters in the date string, then Oracle attempts alternative format elements, as shown in Table 2-20.

Table 2-20 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'MM'	'MON' and 'MONTH'
'MON	'MONTH'
'MONTH'	'MON'

Table 2-20 (Cont.) Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'YY'	'YYYY'
'RR'	'RRRR'

XML Format Model

The SYS_XMLAgg and SYS_XMLGen (deprecated) functions return an instance of type XMLType containing an XML document. Oracle provides the XMLFormat object, which lets you format the output of these functions.

Table 2-21 lists and describes the attributes of the XMLFormat object. The function that implements this type follows the table.

> **See Also:**
>
> - SYS_XMLAGG for information on the SYS_XMLAgg function
> - SYS_XMLGEN for information on the SYS_XMLGen function
> - *Oracle XML DB Developer's Guide* for more information on the implementation of the XMLFormat object and its use

Table 2-21 Attributes of the XMLFormat Object

Attribute	Data Type	Purpose
enclTag	VARCHAR2(4000) or VARCHAR2(32767)[1]	The name of the enclosing tag for the result of the SYS_XMLAgg or SYS_XMLGen (deprecated) function. SYS_XMLAgg: The default is ROWSET. SYS_XMLGen: If the input to the function is a column name, then the default is the column name. Otherwise the default is ROW. When schemaType is set to USE_GIVEN_SCHEMA, this attribute also gives the name of the XMLSchema element.
schemaType	VARCHAR2(100)	The type of schema generation for the output document. Valid values are 'NO_SCHEMA' and 'USE_GIVEN_SCHEMA'. The default is 'NO_SCHEMA'.
schemaName	VARCHAR2(4000) or VARCHAR2(32767)[1]	The name of the target schema Oracle uses if the value of the schemaType is 'USE_GIVEN_SCHEMA'. If you specify schemaName, then Oracle uses the enclosing tag as the element name.
targetNameSpace	VARCHAR2(4000) or VARCHAR2(32767)[1]	The target namespace if the schema is specified (that is, schemaType is GEN_SCHEMA_*, or USE_GIVEN_SCHEMA)
dburlPrefix	VARCHAR2(4000) or VARCHAR2(32767)[1]	The URL to the database to use if WITH_SCHEMA is specified. If this attribute is not specified, then Oracle declares the URL to the types as a relative URL reference.
processingIns	VARCHAR2(4000) or VARCHAR2(32767)[1]	User-provided processing instructions, which are appended to the top of the function output before the element.

1 The data type for this attribute is VARCHAR2(4000) if the initialization parameter MAX_STRING_SIZE = STANDARD, and VARCHAR2(32767) if MAX_STRING_SIZE = EXTENDED. See Extended Data Types for more information.

The function that implements the XMLFormat object follows:

```
STATIC FUNCTION createFormat(
     enclTag IN varchar2 := 'ROWSET',
     schemaType IN varchar2 := 'NO_SCHEMA',
     schemaName IN varchar2 := null,
     targetNameSpace IN varchar2 := null,
     dburlPrefix IN varchar2 := null,
     processingIns IN varchar2 := null) RETURN XMLGenFormatType
       deterministic parallel_enable,
  MEMBER PROCEDURE genSchema (spec IN varchar2),
  MEMBER PROCEDURE setSchemaName(schemaName IN varchar2),
  MEMBER PROCEDURE setTargetNameSpace(targetNameSpace IN varchar2),
  MEMBER PROCEDURE setEnclosingElementName(enclTag IN varchar2),
  MEMBER PROCEDURE setDbUrlPrefix(prefix IN varchar2),
  MEMBER PROCEDURE setProcessingIns(pi IN varchar2),
  CONSTRUCTOR FUNCTION XMLGenFormatType (
     enclTag IN varchar2 := 'ROWSET',
     schemaType IN varchar2 := 'NO_SCHEMA',
     schemaName IN varchar2 := null,
     targetNameSpace IN varchar2 := null,
     dbUrlPrefix IN varchar2 := null,
     processingIns IN varchar2 := null) RETURN SELF AS RESULT
      deterministic parallel_enable,
  STATIC function createFormat2(
      enclTag in varchar2 := 'ROWSET',
      flags in raw) return sys.xmlgenformattype
      deterministic parallel_enable
);
```

Nulls

If a column in a row has no value, then the column is said to be **null**, or to contain null. Nulls can appear in columns of any data type that are not restricted by NOT NULL or PRIMARY KEY integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

Oracle Database treats a character value with a length of zero as null. However, do not use null to represent a numeric value of zero, because they are not equivalent.

> **Note:**
>
> Oracle Database currently treats a character value with a length of zero as null. However, this may not continue to be true in future releases, and Oracle recommends that you do not treat empty strings the same as nulls.

Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

Nulls in SQL Functions

For information on null handling in SQL functions, see Nulls in SQL Functions .

Nulls with Comparison Conditions

To test for nulls, use only the comparison conditions `IS NULL` and `IS NOT NULL`. If you use any other condition with nulls and the result depends on the value of the null, then the result is `UNKNOWN`. Because null represents a lack of data, a null cannot be equal or unequal to any value or to another null. However, Oracle considers two nulls to be equal when evaluating a `DECODE` function. Refer to DECODE for syntax and additional information.

Oracle also considers two nulls to be equal if they appear in compound keys. That is, Oracle considers identical two compound keys containing nulls if all the non-null components of the keys are equal.

Nulls in Conditions

A condition that evaluates to `UNKNOWN` acts almost like `FALSE`. For example, a `SELECT` statement with a condition in the `WHERE` clause that evaluates to `UNKNOWN` returns no rows. However, a condition evaluating to `UNKNOWN` differs from `FALSE` in that further operations on an `UNKNOWN` condition evaluation will evaluate to `UNKNOWN`. Thus, `NOT FALSE` evaluates to `TRUE`, but `NOT UNKNOWN` evaluates to `UNKNOWN`.

Table 2-22 shows examples of various evaluations involving nulls in conditions. If the conditions evaluating to `UNKNOWN` were used in a `WHERE` clause of a `SELECT` statement, then no rows would be returned for that query.

Table 2-22 Conditions Containing Nulls

Condition	Value of A	Evaluation
`a IS NULL`	`10`	`FALSE`
`a IS NOT NULL`	`10`	`TRUE`
`a IS NULL`	`NULL`	`TRUE`
`a IS NOT NULL`	`NULL`	`FALSE`
`a = NULL`	`10`	`UNKNOWN`
`a != NULL`	`10`	`UNKNOWN`
`a = NULL`	`NULL`	`UNKNOWN`
`a != NULL`	`NULL`	`UNKNOWN`
`a = 10`	`NULL`	`UNKNOWN`
`a != 10`	`NULL`	`UNKNOWN`

For the truth tables showing the results of logical conditions containing nulls, see Table 6-5, Table 6-6, and Table 6-7.

Comments

You can create two types of comments:

- Comments within SQL statements are stored as part of the application code that executes the SQL statements.
- Comments associated with individual schema or nonschema objects are stored in the data dictionary along with metadata on the objects themselves.

Comments Within SQL Statements

Comments can make your application easier for you to read and maintain. For example, you can include a comment in a statement that describes the purpose of the statement within your application. With the exception of hints, comments within SQL statements do not affect the statement execution. Refer to Hints on using this particular form of comment.

A comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement in two ways:

- Begin the comment with a slash and an asterisk (/*). Proceed with the text of the comment. This text can span multiple lines. End the comment with an asterisk and a slash (*/). The opening and terminating characters need not be separated from the text by a space or a line break.
- Begin the comment with -- (two hyphens). Proceed with the text of the comment. This text cannot extend to a new line. End the comment with a line break.

Some of the tools used to enter SQL have additional restrictions. For example, if you are using SQL*Plus, by default you cannot have a blank line inside a multiline comment. For more information, refer to the documentation for the tool you use as an interface to the database.

A SQL statement can contain multiple comments of both styles. The text of a comment can contain any printable characters in your database character set.

Example

These statements contain many comments:

```
SELECT last_name, employee_id, salary + NVL(commission_pct, 0),
       job_id, e.department_id
  /* Select all employees whose compensation is
  greater than that of Pataballa.*/
  FROM employees e, departments d
  /*The DEPARTMENTS table is used to get the department name.*/
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct,0) >   /* Subquery:       */
      (SELECT salary + NVL(commission_pct,0)
        /* total compensation is salary + commission_pct */
        FROM employees
        WHERE last_name = 'Pataballa')
  ORDER BY last_name, employee_id;

SELECT last_name,                                    -- select the name
       employee_id                                   -- employee id
       salary + NVL(commission_pct, 0),              -- total compensation
```

```
       job_id,                                  -- job
       e.department_id                          -- and department
  FROM employees e,                             -- of all employees
       departments d
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct, 0) >       -- whose compensation
                                                -- is greater than
        (SELECT salary + NVL(commission_pct,0)  -- the compensation
          FROM employees
          WHERE last_name = 'Pataballa')        -- of Pataballa
  ORDER BY last_name                            -- and order by last name
           employee_id                          -- and employee id.
;
```

Comments on Schema and Nonschema Objects

You can use the `COMMENT` command to associate a comment with a schema object (table, view, materialized view, operator, indextype, mining model) or a nonschema object (edition) using the `COMMENT` command. You can also create a comment on a column, which is part of a table schema object. Comments associated with schema and nonschema objects are stored in the data dictionary. Refer to COMMENT for a description of this form of comment.

Hints

Hints are comments in a SQL statement that pass instructions to the Oracle Database optimizer. The optimizer uses these hints to choose an execution plan for the statement, unless some condition exists that prevents the optimizer from doing so.

Hints were introduced in Oracle7, when users had little recourse if the optimizer generated suboptimal plans. Now Oracle provides a number of tools, including the SQL Tuning Advisor, SQL plan management, and SQL Performance Analyzer, to help you address performance problems that are not solved by the optimizer. Oracle strongly recommends that you use those tools rather than hints. The tools are far superior to hints, because when used on an ongoing basis, they provide fresh solutions as your data and database environment change.

Hints should be used sparingly, and only after you have collected statistics on the relevant tables and evaluated the optimizer plan without hints using the `EXPLAIN PLAN` statement. Changing database conditions as well as query performance enhancements in subsequent releases can have significant impact on how hints in your code affect performance.

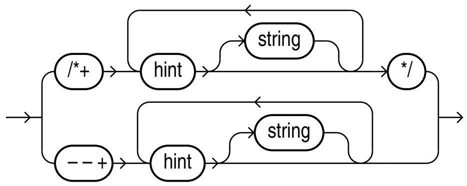
The remainder of this section provides information on some commonly used hints. If you decide to use hints rather than the more advanced tuning tools, be aware that any short-term benefit resulting from the use of hints may not continue to result in improved performance over the long term.

Using Hints

A statement block can have only one comment containing hints, and that comment must follow the `SELECT`, `UPDATE`, `INSERT`, `MERGE`, or `DELETE` keyword.

The following syntax diagram shows hints contained in both styles of comments that Oracle supports within a statement block. The hint syntax must follow immediately after an `INSERT`, `UPDATE`, `DELETE`, `SELECT`, or `MERGE` keyword that begins the statement block.

***hint*::=**



where:

- The plus sign (+) causes Oracle to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter. No space is permitted.
- *`hint`* is one of the hints discussed in this section. The space between the plus sign and the hint is optional. If the comment contains multiple hints, then separate the hints by at least one space.
- *`string`* is other commenting text that can be interspersed with the hints.

The `--+` syntax requires that the entire comment be on a single line.

Oracle Database ignores hints and does not return an error under the following circumstances:

- The hint contains misspellings or syntax errors. However, the database does consider other correctly specified hints in the same comment.
- The comment containing the hint does not follow a `DELETE`, `INSERT`, `MERGE`, `SELECT`, or `UPDATE` keyword.
- A combination of hints conflict with each other. However, the database does consider other hints in the same comment.
- The database environment uses PL/SQL version 1, such as Forms version 3 triggers, Oracle Forms 4.5, and Oracle Reports 2.5.
- A global hint refers to multiple query blocks. Refer to Specifying Multiple Query Blocks in a Global Hint for more information.

Specifying a Query Block in a Hint

You can specify an optional query block name in many hints to specify the query block to which the hint applies. This syntax lets you specify in the outer query a hint that applies to an inline view.

The syntax of the query block argument is of the form *`@queryblock`*, where *`queryblock`* is an identifier that specifies a query block in the query. The *`queryblock`* identifier can either be system-generated or user-specified. When you specify a hint in the query block itself to which the hint applies, you omit the *`@queryblock`* syntax.

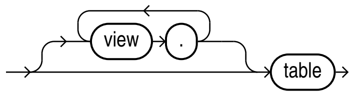
- The system-generated identifier can be obtained by using `EXPLAIN PLAN` for the query. Pretransformation query block names can be determined by running `EXPLAIN PLAN` for the query using the `NO_QUERY_TRANSFORMATION` hint. See NO_QUERY_TRANSFORMATION Hint .

- The user-specified name can be set with the `QB_NAME` hint. See QB_NAME Hint .

Specifying Global Hints

Many hints can apply both to specific tables or indexes and more globally to tables within a view or to columns that are part of indexes. The syntactic elements *`tablespec`* and *`indexspec`* define these **global hints**.

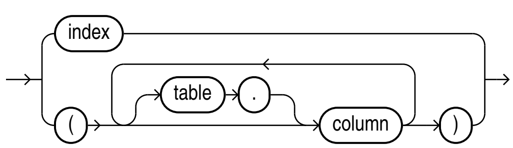
***tablespec*::=**



You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, then use the alias rather than the table name in the hint. However, do not include the schema name with the table name within the hint, even if the schema name appears in the statement.

> **Note:**
>
> Specifying a global hint using the *`tablespec`* clause does not work for queries that use ANSI joins, because the optimizer generates additional views during parsing. Instead, specify `@`*`queryblock`* to indicate the query block to which the hint applies.

***indexspec*::=**



When *`tablespec`* is followed by *`indexspec`* in the specification of a hint, a comma separating the table name and index name is permitted but not required. Commas are also permitted, but not required, to separate multiple occurrences of *`indexspec`*.

Specifying Multiple Query Blocks in a Global Hint

Oracle Database ignores global hints that refer to multiple query blocks. To avoid this issue, Oracle recommends that you specify the object alias in the hint instead of using *`tablespec`* and *`indexspec`*.

For example, consider the following view `v` and table `t`:

```
CREATE VIEW v AS
  SELECT e.last_name, e.department_id, d.location_id
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;

CREATE TABLE t AS
```

```
SELECT * from employees
WHERE employee_id < 200;
```

Note:

The following examples use the `EXPLAIN PLAN` statement, which enables you to display the execution plan and determine if a hint is honored or ignored. Refer to EXPLAIN PLAN for more information.

The `LEADING` hint is ignored in the following query because it refers to multiple query blocks, that is, the main query block containing table `t` and the view query block `v`:

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'Test 1'
  INTO plan_table FOR
    (SELECT /*+ LEADING(v.e v.d t) */ *
     FROM t, v
     WHERE t.department_id = v.department_id);
```

The following `SELECT` statement returns the execution plan, which shows that the `LEADING` hint was ignored:

```
SELECT id, LPAD(' ',2*(LEVEL-1))||operation operation, options, object_name,
object_alias
  FROM plan_table
  START WITH id = 0 AND statement_id = 'Test 1'
  CONNECT BY PRIOR id = parent_id AND statement_id = 'Test 1'
  ORDER BY id;

 ID OPERATION            OPTIONS    OBJECT_NAME   OBJECT_ALIAS
--- -------------------- ---------- ------------- --------------------
  0 SELECT STATEMENT
  1   HASH JOIN
  2     HASH JOIN
  3       TABLE ACCESS   FULL       DEPARTMENTS   D@SEL$2
  4       TABLE ACCESS   FULL       EMPLOYEES     E@SEL$2
  5     TABLE ACCESS     FULL       T             T@SEL$1
```

The `LEADING` hint is honored in the following query because it refers to object aliases, which can be found in the execution plan that was returned by the previous query:

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'Test 2'
  INTO plan_table FOR
    (SELECT /*+ LEADING(E@SEL$2 D@SEL$2 T@SEL$1) */ *
     FROM t, v
     WHERE t.department_id = v.department_id);
```

The following `SELECT` statement returns the execution plan, which shows that the `LEADING` hint was honored:

```
SELECT id, LPAD(' ',2*(LEVEL-1))||operation operation, options,
  object_name, object_alias
  FROM plan_table
  START WITH id = 0 AND statement_id = 'Test 2'
  CONNECT BY PRIOR id = parent_id AND statement_id = 'Test 2'
  ORDER BY id;
```

```
 ID OPERATION            OPTIONS    OBJECT_NAME   OBJECT_ALIAS
--- -------------------- ---------- ------------- --------------------
  0 SELECT STATEMENT
  1   HASH JOIN
  2     HASH JOIN
  3       TABLE ACCESS   FULL       EMPLOYEES     E@SEL$2
  4       TABLE ACCESS   FULL       DEPARTMENTS   D@SEL$2
  5     TABLE ACCESS     FULL       T             T@SEL$1
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information on using `EXPLAIN PLAN` to learn how the optimizer is executing a query

Hints by Functional Category

Table 2-23 lists the hints by functional category and contains cross-references to the syntax and semantics for each hint. An alphabetical reference of the hints follows the table.

Table 2-23 Hints by Functional Category

Hint	Link to Syntax and Semantics
Optimization Goals and Approaches	ALL_ROWS Hint FIRST_ROWS Hint
Access Path Hints	CLUSTER Hint
--	CLUSTERING Hint NO_CLUSTERING Hint
--	FULL Hint
--	HASH Hint
--	INDEX Hint NO_INDEX Hint
--	INDEX_ASC Hint INDEX_DESC Hint
--	INDEX_COMBINE Hint
--	INDEX_JOIN Hint
--	INDEX_FFS Hint
--	INDEX_SS Hint
--	INDEX_SS_ASC Hint
--	INDEX_SS_DESC Hint
--	NATIVE_FULL_OUTER_JOIN Hint NO_NATIVE_FULL_OUTER_JOIN Hint
--	NO_INDEX_FFS Hint
--	NO_INDEX_SS Hint
--	NO_ZONEMAP Hint

Table 2-23 (Cont.) Hints by Functional Category

Hint	Link to Syntax and Semantics
In-Memory Column Store Hints	INMEMORY Hint NO_INMEMORY Hint
--	INMEMORY_PRUNING Hint NO_INMEMORY_PRUNING Hint
Join Order Hints	ORDERED Hint
--	LEADING Hint
Join Operation Hints	USE_BAND Hint NO_USE_BAND Hint
--	USE_CUBE Hint NO_USE_CUBE Hint
--	USE_HASH Hint NO_USE_HASH Hint
--	USE_MERGE Hint NO_USE_MERGE Hint
--	USE_NL Hint USE_NL_WITH_INDEX Hint NO_USE_NL Hint
Parallel Execution Hints	ENABLE_PARALLEL_DML Hint DISABLE_PARALLEL_DML Hint
--	PARALLEL Hint NO_PARALLEL Hint
--	PARALLEL_INDEX Hint NO_PARALLEL_INDEX Hint
--	PQ_CONCURRENT_UNION Hint NO_PQ_CONCURRENT_UNION Hint
--	PQ_DISTRIBUTE Hint
--	PQ_FILTER Hint
--	PQ_SKEW Hint NO_PQ_SKEW Hint
Online Application Upgrade Hints	CHANGE_DUPKEY_ERROR_INDEX Hint
--	IGNORE_ROW_ON_DUPKEY_INDEX Hint
--	RETRY_ON_ROW_CHANGE Hint
Query Transformation Hints	FACT Hint NO_FACT Hint
--	MERGE Hint NO_MERGE Hint
--	NO_EXPAND Hint USE_CONCAT Hint

Table 2-23 (Cont.) Hints by Functional Category

Hint	Link to Syntax and Semantics
--	REWRITE Hint NO_REWRITE Hint
--	UNNEST Hint NO_UNNEST Hint
--	STAR_TRANSFORMATION Hint NO_STAR_TRANSFORMATION Hint
--	NO_QUERY_TRANSFORMATION Hint
XML Hints	NO_XMLINDEX_REWRITE Hint
--	NO_XML_QUERY_REWRITE Hint
Other Hints	APPEND Hint APPEND_VALUES Hint NOAPPEND Hint
--	CACHE Hint NOCACHE Hint
--	CONTAINERS Hint
--	CURSOR_SHARING_EXACT Hint
--	DRIVING_SITE Hint
--	DYNAMIC_SAMPLING Hint
	FRESH_MV Hint
--	GATHER_OPTIMIZER_STATISTICS Hint NO_GATHER_OPTIMIZER_STATISTICS Hint
	GROUPING Hint
--	MODEL_MIN_ANALYSIS Hint
--	MONITOR Hint
--	NO_MONITOR Hint
--	OPT_PARAM Hint
--	PUSH_PRED Hint NO_PUSH_PRED Hint
--	PUSH_SUBQ Hint NO_PUSH_SUBQ Hint
--	PX_JOIN_FILTER Hint NO_PX_JOIN_FILTER Hint
--	QB_NAME Hint
--	RESULT_CACHE Hint NO_RESULT_CACHE Hint

Alphabetical Listing of Hints

This section provides syntax and semantics for all hints in alphabetical order.

ALL_ROWS Hint

The ALL_ROWS hint instructs the optimizer to optimize a statement block with a goal of best throughput, which is minimum total resource consumption. For example, the optimizer uses the query optimization approach to optimize this statement for best throughput:

```
SELECT /*+ ALL_ROWS */ employee_id, last_name, salary, job_id
  FROM employees
  WHERE employee_id = 107;
```

If you specify either the ALL_ROWS or the FIRST_ROWS hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values, such as allocated storage for such tables, to estimate the missing statistics and to subsequently choose an execution plan. These estimates might not be as accurate as those gathered by the DBMS_STATS package, so you should use the DBMS_STATS package to gather statistics.

If you specify hints for access paths or join operations along with either the ALL_ROWS or FIRST_ROWS hint, then the optimizer gives precedence to the access paths and join operations specified by the hints.

APPEND Hint

The APPEND hint instructs the optimizer to use direct-path INSERT with the subquery syntax of the INSERT statement.

- Conventional INSERT is the default in serial mode. In serial mode, direct path can be used only if you include the APPEND hint.
- Direct-path INSERT is the default in parallel mode. In parallel mode, conventional insert can be used only if you specify the NOAPPEND hint.

The decision whether the INSERT will go parallel or not is independent of the APPEND hint.

In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

The APPEND hint is only supported with the subquery syntax of the INSERT statement, not the VALUES clause. If you specify the APPEND hint with the VALUES clause, it is ignored and conventional insert will be used. To use direct-path INSERT with the VALUES clause, refer to "APPEND_VALUES Hint ".

See Also:

NOAPPEND Hint for information on that hint and *Oracle Database Administrator's Guide* for information on direct-path inserts

APPEND_VALUES Hint

The APPEND_VALUES hint instructs the optimizer to use direct-path INSERT with the VALUES clause. If you do not specify this hint, then conventional INSERT is used.

In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

The APPEND_VALUES hint can be used to greatly enhance performance. Some examples of its uses are:

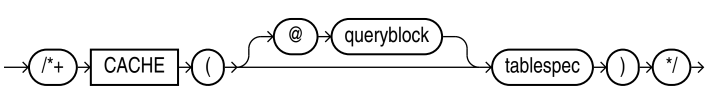
- In an Oracle Call Interface (OCI) program, when using large array binds or array binds with row callbacks
- In PL/SQL, when loading a large number of rows with a FORALL loop that has an INSERT statement with a VALUES clause

The APPEND_VALUES hint is only supported with the VALUES clause of the INSERT statement. If you specify the APPEND_VALUES hint with the subquery syntax of the INSERT statement, it is ignored and conventional insert will be used. To use direct-path INSERT with a subquery, refer to "APPEND Hint ".

See Also:

Oracle Database Administrator's Guide for information on direct-path inserts

CACHE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

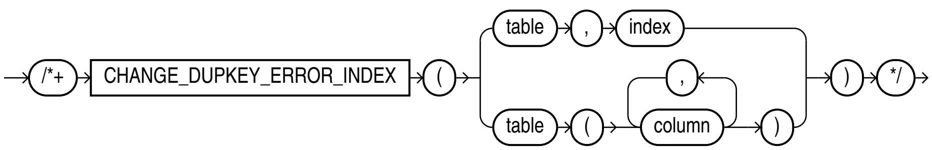
The CACHE hint instructs the optimizer to place the blocks retrieved for the table at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This hint is useful for small lookup tables.

In the following example, the `CACHE` hint overrides the default caching specification of the table:

```
SELECT /*+ FULL (hr_emp) CACHE(hr_emp) */ last_name
  FROM employees hr_emp;
```

The `CACHE` and `NOCACHE` hints affect system statistics `table scans (long tables)` and `table scans (short tables)`, as shown in the `V$SYSSTAT` data dictionary view.

CHANGE_DUPKEY_ERROR_INDEX Hint



> **Note:**
>
> The `CHANGE_DUPKEY_ERROR_INDEX`, `IGNORE_ROW_ON_DUPKEY_INDEX`, and `RETRY_ON_ROW_CHANGE` hints are unlike other hints in that they have a semantic effect. The general philosophy explained in Hints does not apply for these three hints.

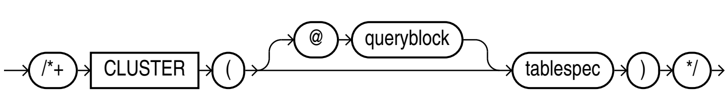
The `CHANGE_DUPKEY_ERROR_INDEX` hint provides a mechanism to unambiguously identify a unique key violation for a specified set of columns or for a specified index. When a unique key violation occurs for the specified index, an ORA-38911 error is reported instead of an ORA-001.

This hint applies to `INSERT`, `UPDATE` operations. If you specify an index, then the index must exist and be unique. If you specify a column list instead of an index, then a unique index whose columns match the specified columns in number and order must exist.

This use of this hint results in error messages if specific rules are violated. Refer to IGNORE_ROW_ON_DUPKEY_INDEX Hint for details.

> **Note:**
>
> This hint disables both `APPEND` mode and parallel DML.

CLUSTER Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `CLUSTER` hint instructs the optimizer to use a cluster scan to access the specified table. This hint applies only to tables in an indexed cluster.

CLUSTERING Hint

This hint is valid only for `INSERT` and `MERGE` operations on tables that are enabled for attribute clustering. The `CLUSTERING` hint enables attribute clustering for direct-path inserts (serial or parallel). This results in partially-clustered data, that is, data that is clustered per each insert or merge operation. This hint overrides a `NO ON LOAD` setting in the DDL that created or altered the table. This hint has no effect on tables that are not enabled for attribute clustering.

> **See Also:**
>
> - clustering_when clause of `CREATE TABLE` for more information on the `NO ON LOAD` setting
> - NO_CLUSTERING Hint

CONTAINERS Hint

The `CONTAINERS` hint is useful in a multitenant container database (CDB). You can specify this hint in a `SELECT` statement that contains the `CONTAINERS()` clause. Such a statement lets you query data in the specified table or view across all containers in a CDB or application container.

- To query data in a CDB, you must be a common user connected to the CDB root, and the table or view must exist in the root and all PDBs. The query returns all rows from the table or view in the CDB root and in all open PDBs.
- To query data in an application container, you must be a common user connected to the application root, and the table or view must exist in the application root and all PDBs in the application container. The query returns all rows from the table or view in the application root and in all open PDBs in the application container.

Statements that contain the `CONTAINERS()` clause generate and execute recursive SQL statements in each queried PDB. You can use the `CONTAINERS` hint to pass a default PDB hint to each recursive SQL statement. For *`hint`*, you can specify any SQL hint that is appropriate for the `SELECT` statement.

In the following example, the NO_PARALLEL hint is passed to each recursive SQL statement that is executed as part of the evaluation of the CONTAINERS() clause:

```
SELECT /*+ CONTAINERS(DEFAULT_PDB_HINT='NO_PARALLEL') */
  (CASE WHEN COUNT(*) < 10000
        THEN 'Less than 10,000'
        ELSE '10,000 or more' END) "Number of Tables"
  FROM CONTAINERS(DBA_TABLES);
```

> **See Also:**
>
> *containers_clause* for more information on the CONTAINERS() clause

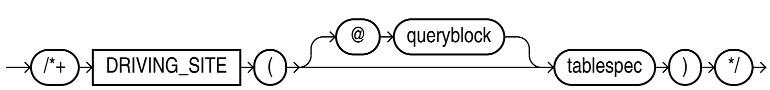
CURSOR_SHARING_EXACT Hint

Oracle can replace literals in SQL statements with bind variables, when it is safe to do so. This replacement is controlled with the CURSOR_SHARING initialization parameter. The CURSOR_SHARING_EXACT hint instructs the optimizer to switch this behavior off. When you specify this hint, Oracle executes the SQL statement without any attempt to replace literals with bind variables.

DISABLE_PARALLEL_DML Hint

The DISABLE_PARALLEL_DML hint disables parallel DML for DELETE, INSERT, MERGE, and UPDATE statements. You can use this hint to disable parallel DML for an individual statement when parallel DML is enabled for the session with the ALTER SESSION ENABLE PARALLEL DML statement.

DRIVING_SITE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

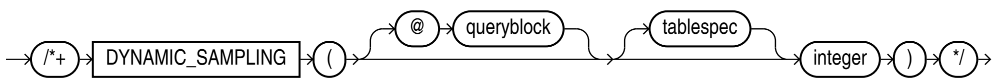
The DRIVING_SITE hint instructs the optimizer to execute the query at a different site than that selected by the database. This hint is useful if you are using distributed query optimization.

For example:

```
SELECT /*+ DRIVING_SITE(departments) */ *
  FROM employees, departments@rsite
  WHERE employees.department_id = departments.department_id;
```

If this query is executed without the hint, then rows from `departments` are sent to the local site, and the join is executed there. With the hint, the rows from `employees` are sent to the remote site, and the query is executed there and the result set is returned to the local site.

DYNAMIC_SAMPLING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `DYNAMIC_SAMPLING` hint instructs the optimizer how to control dynamic sampling to improve server performance by determining more accurate predicate selectivity and statistics for tables and indexes.

You can set the value of `DYNAMIC_SAMPLING` to a value from 0 to 10. The higher the level, the more effort the compiler puts into dynamic sampling and the more broadly it is applied. Sampling defaults to cursor level unless you specify *`tablespec`*.

The *`integer`* value is `0` to `10`, indicating the degree of sampling.

If a cardinality statistic already exists for the table, then the optimizer uses it. Otherwise, the optimizer enables dynamic sampling to estimate the cardinality statistic.

If you specify *`tablespec`* and the cardinality statistic already exists, then:

- If there is no single-table predicate (a `WHERE` clause that evaluates only one table), then the optimizer trusts the existing statistics and ignores this hint. For example, the following query will not result in any dynamic sampling if `employees` is analyzed:

  ```
  SELECT /*+ DYNAMIC_SAMPLING(e 1) */ count(*)
    FROM employees e;
  ```

- If there is a single-table predicate, then the optimizer uses the existing cardinality statistic and estimates the selectivity of the predicate using the existing statistics.

To apply dynamic sampling to a specific table, use the following form of the hint:

```
SELECT /*+ DYNAMIC_SAMPLING(employees 1) */ *
  FROM employees
  WHERE ...
```

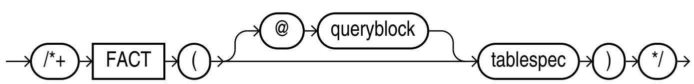
> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information about dynamic sampling and the sampling levels that you can set

ENABLE_PARALLEL_DML Hint

The `ENABLE_PARALLEL_DML` hint enables parallel DML for `DELETE`, `INSERT`, `MERGE`, and `UPDATE` statements. You can use this hint to enable parallel DML for an individual statement, rather than enabling parallel DML for the session with the `ALTER SESSION ENABLE PARALLEL DML` statement.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for information about enabling parallel DML

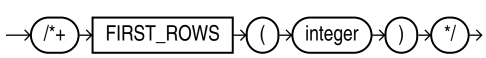
FACT Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `FACT` hint is used in the context of the star transformation. It instructs the optimizer that the table specified in *`tablespec`* should be considered as a fact table.

FIRST_ROWS Hint



The `FIRST_ROWS` hint instructs Oracle to optimize an individual SQL statement for fast response, choosing the plan that returns the first *`n`* rows most efficiently. For *`integer`*, specify the number of rows to return.

For example, the optimizer uses the query optimization approach to optimize the following statement for best response time:

```
SELECT /*+ FIRST_ROWS(10) */ employee_id, last_name, salary, job_id
  FROM employees
  WHERE department_id = 20;
```

In this example each department contains many employees. The user wants the first 10 employees of department 20 to be displayed as quickly as possible.

The optimizer ignores this hint in `DELETE` and `UPDATE` statement blocks and in `SELECT` statement blocks that include any blocking operations, such as sorts or groupings.

Such statements cannot be optimized for best response time, because Oracle Database must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any such statement, then the database optimizes for best throughput.

> **See Also:**
>
> ALL_ROWS Hint for additional information on the `FIRST_ROWS` hint and statistics

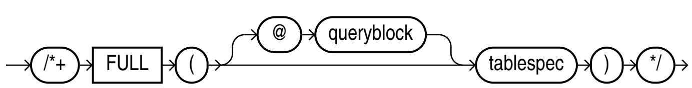
FRESH_MV Hint

The `FRESH_MV` hint applies when querying a real-time materialized view. This hint instructs the optimizer to use on-query computation to fetch up-to-date data from the materialized view, even if the materialized view is stale.

The optimizer ignores this hint in `SELECT` statement blocks that query an object that is not a real-time materialized view, and in all `UPDATE`, `INSERT`, `MERGE`, and `DELETE` statement blocks.

> **See Also:**
>
> The { ENABLE | DISABLE } ON QUERY COMPUTATION clause of `CREATE MATERIALIZED VIEW` for more information on real-time materialized views

FULL Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `FULL` hint instructs the optimizer to perform a full table scan for the specified table. For example:

```
SELECT /*+ FULL(e) */ employee_id, last_name
  FROM hr.employees e
  WHERE last_name LIKE :b1;
```

Oracle Database performs a full table scan on the `employees` table to execute this statement, even if there is an index on the `last_name` column that is made available by the condition in the `WHERE` clause.

The `employees` table has alias `e` in the `FROM` clause, so the hint must refer to the table by its alias rather than by its name. Do not specify schema names in the hint even if they are specified in the `FROM` clause.

GATHER_OPTIMIZER_STATISTICS Hint

The GATHER_OPTIMIZER_STATISTICS hint instructs the optimizer to enable statistics gathering during the following types of bulk loads:

- CREATE TABLE ... AS SELECT
- INSERT INTO ... SELECT into an empty table using a direct-path insert

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information on statistics gathering for bulk loads

GROUPING Hint

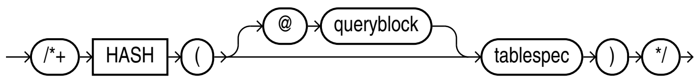
The GROUPING hint applies to data mining scoring functions when scoring partitioned models. This hint results in partitioning the input data set into distinct data slices so that each partition is scored in its entirety before advancing to the next partition; however, parallelism by partition is still available. Data slices are determined by the partitioning key columns that were used when the model was built. This method can be used with any data mining function against a partitioned model. The hint may yield a query performance gain when scoring large data that is associated with many partitions, but may negatively impact performance when scoring large data with few partitions on large systems. Typically, there is no performance gain if you use this hint for single row queries.

In the following example, the GROUPING hint is used in the PREDICTION function.

```
SELECT PREDICTION(/*+ GROUPING */my_model USING *) pred FROM <input
table>;
```

> **See Also:**
>
> Data Mining Functions

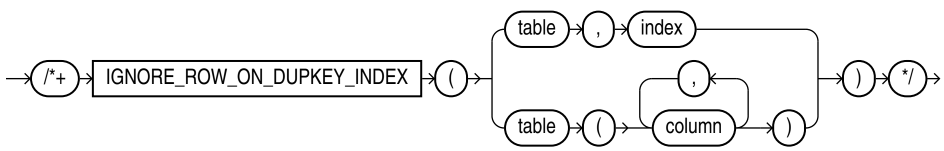
HASH Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `HASH` hint instructs the optimizer to use a hash scan to access the specified table. This hint applies only to tables in a hash cluster.

IGNORE_ROW_ON_DUPKEY_INDEX Hint



> **Note:**
>
> The `CHANGE_DUPKEY_ERROR_INDEX`, `IGNORE_ROW_ON_DUPKEY_INDEX`, and `RETRY_ON_ROW_CHANGE` hints are unlike other hints in that they have a semantic effect. The general philosophy explained in Hints does not apply for these three hints.

The `IGNORE_ROW_ON_DUPKEY_INDEX` hint applies only to single-table `INSERT` operations. It is not supported for `UPDATE`, `DELETE`, `MERGE`, or multitable insert operations. `IGNORE_ROW_ON_DUPKEY_INDEX` causes the statement to ignore a unique key violation for a specified set of columns or for a specified index. When a unique key violation is encountered, a row-level rollback occurs and execution resumes with the next input row. If you specify this hint when inserting data with DML error logging enabled, then the unique key violation is not logged and does not cause statement termination.

The semantic effect of this hint results in error messages if specific rules are violated:

- If you specify *`index`*, then the index must exist and be unique. Otherwise, the statement causes ORA-38913.
- You must specify exactly one index. If you specify no index, then the statement causes ORA-38912. If you specify more than one index, then the statement causes ORA-38915.
- You can specify either a `CHANGE_DUPKEY_ERROR_INDEX` or `IGNORE_ROW_ON_DUPKEY_INDEX` hint in an `INSERT` statement, but not both. If you specify both, then the statement causes ORA-38915.

As with all hints, a syntax error in the hint causes it to be silently ignored. The result will be that ORA-00001 will be caused, just as if no hint were used.

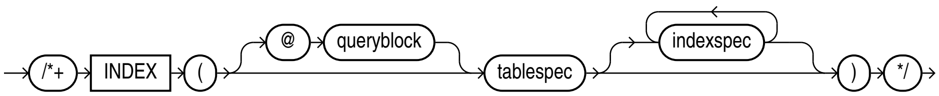
Note:

This hint disables both `APPEND` mode and parallel `DML`.

See Also:

CHANGE_DUPKEY_ERROR_INDEX Hint

INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX` hint instructs the optimizer to use an index scan for the specified table. You can use the `INDEX` hint for function-based, domain, B-tree, bitmap, and bitmap join indexes.

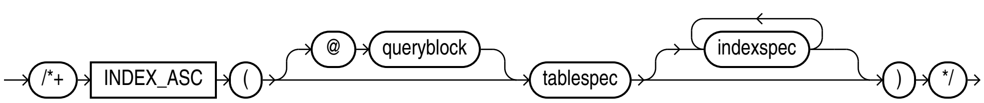
The behavior of the hint depends on the *`indexspec`* specification:

- If the `INDEX` hint specifies a single available index, then the database performs a scan on this index. The optimizer does not consider a full table scan or a scan of another index on the table.
- For a hint on a combination of multiple indexes, Oracle recommends using `INDEX_COMBINE` rather than `INDEX`, because it is a more versatile hint. If the `INDEX` hint specifies a list of available indexes, then the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The database can also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The database does not consider a full table scan or a scan on an index not listed in the hint.
- If the `INDEX` hint specifies no indexes, then the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The database can also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

For example:

```
SELECT /*+ INDEX (employees emp_department_ix)*/ employee_id, department_id
  FROM employees
  WHERE department_id > 50;
```

INDEX_ASC Hint

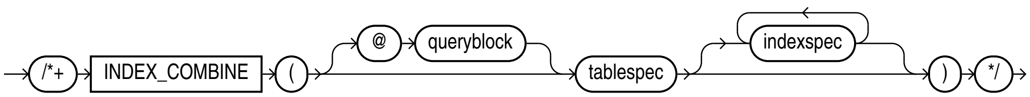


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_ASC` hint instructs the optimizer to use an index scan for the specified table. If the statement uses an index range scan, then Oracle Database scans the index entries in ascending order of their indexed values. Each parameter serves the same purpose as in INDEX Hint .

The default behavior for a range scan is to scan index entries in ascending order of their indexed values, or in descending order for a descending index. This hint does not change the default order of the index, and therefore does not specify anything more than the `INDEX` hint. However, you can use the `INDEX_ASC` hint to specify ascending range scans explicitly should the default behavior change.

INDEX_COMBINE Hint

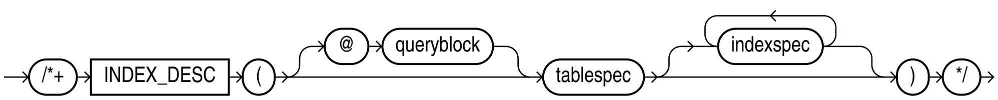


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_COMBINE` hint instructs the optimizer to use a bitmap access path for the table. If *`indexspec`* is omitted from the `INDEX_COMBINE` hint, then the optimizer uses whatever Boolean combination of indexes has the best cost estimate for the table. If you specify `indexspec`, then the optimizer tries to use some Boolean combination of the specified indexes. Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_COMBINE(e emp_manager_ix emp_department_ix) */ *
  FROM employees e
  WHERE manager_id = 108
     OR department_id = 110;
```

INDEX_DESC Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

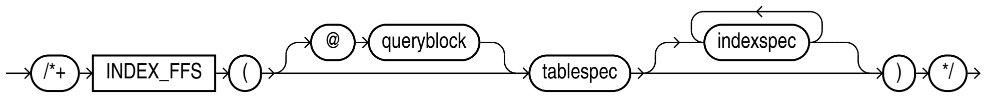
The `INDEX_DESC` hint instructs the optimizer to use a descending index scan for the specified table. If the statement uses an index range scan and the index is ascending, then Oracle scans the index entries in descending order of their indexed values. In a partitioned index, the results are in descending order within each partition. For a descending index, this hint effectively cancels out the descending order, resulting in a scan of the index entries in ascending order. Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_DESC(e emp_name_ix) */ *
  FROM employees e;
```

See Also:

Oracle Database SQL Tuning Guide for information on full scans

INDEX_FFS Hint



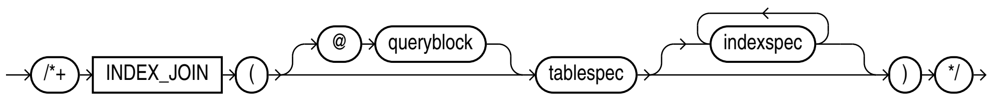
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_FFS` hint instructs the optimizer to perform a fast full index scan rather than a full table scan.

Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_FFS(e emp_name_ix) */ first_name
  FROM employees e;
```

INDEX_JOIN Hint



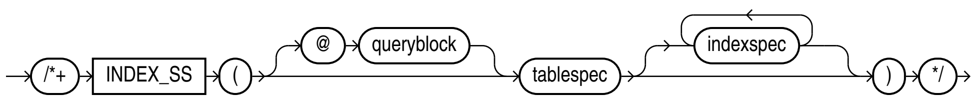
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_JOIN` hint instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.

Each parameter serves the same purpose as in INDEX Hint . For example, the following query uses an index join to access the `manager_id` and `department_id` columns, both of which are indexed in the `employees` table.

```
SELECT /*+ INDEX_JOIN(e emp_manager_ix emp_department_ix) */ department_id
  FROM employees e
  WHERE manager_id < 110
    AND department_id < 50;
```

INDEX_SS Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

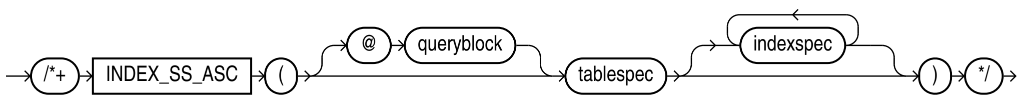
The `INDEX_SS` hint instructs the optimizer to perform an index skip scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in ascending order of their indexed values. In a partitioned index, the results are in ascending order within each partition.

Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_SS(e emp_name_ix) */ last_name
  FROM employees e
  WHERE first_name = 'Steven';
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

INDEX_SS_ASC Hint



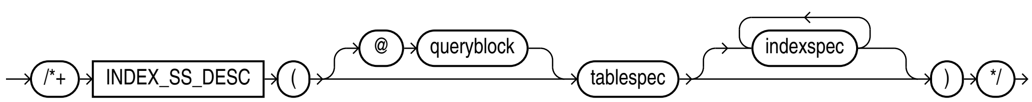
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_SS_ASC` hint instructs the optimizer to perform an index skip scan for the specified table. If the statement uses an index range scan, then Oracle Database scans the index entries in ascending order of their indexed values. In a partitioned index, the results are in ascending order within each partition. Each parameter serves the same purpose as in INDEX Hint .

The default behavior for a range scan is to scan index entries in ascending order of their indexed values, or in descending order for a descending index. This hint does not change the default order of the index, and therefore does not specify anything more than the `INDEX_SS` hint. However, you can use the `INDEX_SS_ASC` hint to specify ascending range scans explicitly should the default behavior change.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

INDEX_SS_DESC Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

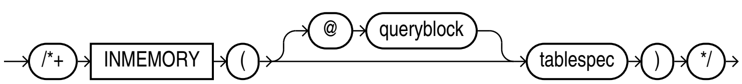
The `INDEX_SS_DESC` hint instructs the optimizer to perform an index skip scan for the specified table. If the statement uses an index range scan and the index is ascending, then Oracle scans the index entries in descending order of their indexed values. In a partitioned index, the results are in descending order within each partition. For a descending index, this hint effectively cancels out the descending order, resulting in a scan of the index entries in ascending order.

Each parameter serves the same purpose as in the INDEX Hint . For example:

```
SELECT /*+ INDEX_SS_DESC(e emp_name_ix) */ last_name
  FROM employees e
  WHERE first_name = 'Steven';
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

INMEMORY Hint

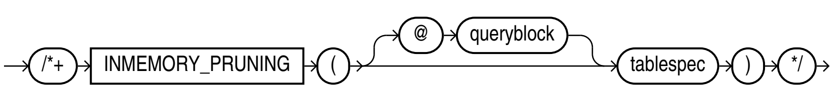


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `INMEMORY` hint enables In-Memory queries.

This hint does not instruct the optimizer to perform a full table scan. If a full table scan is desired, then also specify the FULL Hint .

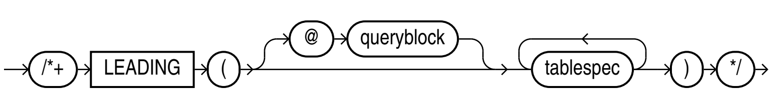
INMEMORY_PRUNING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `INMEMORY_PRUNING` hint enables pruning of In-Memory queries.

LEADING Hint



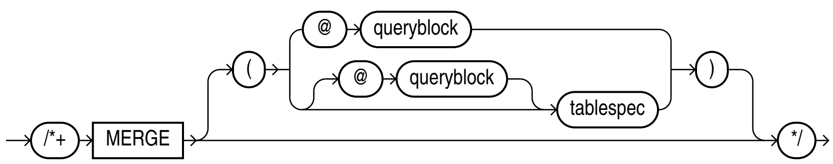
(See Specifying a Query Block in a Hint , *tablespec*::=)

The LEADING hint instructs the optimizer to use the specified set of tables as the prefix in the execution plan. This hint is more versatile than the ORDERED hint. For example:

```
SELECT /*+ LEADING(e j) */ *
    FROM employees e, departments d, job_history j
    WHERE e.department_id = d.department_id
      AND e.hire_date = j.start_date;
```

The LEADING hint is ignored if the tables specified cannot be joined first in the order specified because of dependencies in the join graph. If you specify two or more conflicting LEADING hints, then all of them are ignored. If you specify the ORDERED hint, it overrides all LEADING hints.

MERGE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The MERGE hint lets you merge views in a query.

If a view's query block contains a GROUP BY clause or DISTINCT operator in the SELECT list, then the optimizer can merge the view into the accessing statement only if complex view merging is enabled. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is uncorrelated.

For example:

```
SELECT /*+ MERGE(v) */ e1.last_name, e1.salary, v.avg_salary
   FROM employees e1,
        (SELECT department_id, avg(salary) avg_salary
           FROM employees e2
           GROUP BY department_id) v
   WHERE e1.department_id = v.department_id
     AND e1.salary > v.avg_salary
   ORDER BY e1.last_name;
```

When the MERGE hint is used without an argument, it should be placed in the view query block. When MERGE is used with the view name as an argument, it should be placed in the surrounding query.

MODEL_MIN_ANALYSIS Hint

The MODEL_MIN_ANALYSIS hint instructs the optimizer to omit some compile-time optimizations of spreadsheet rules—primarily detailed dependency graph analysis. Other spreadsheet optimizations, such as creating filters to selectively populate spreadsheet access structures and limited rule pruning, are still used by the optimizer.

This hint reduces compilation time because spreadsheet analysis can be lengthy if the number of spreadsheet rules is more than several hundreds.

MONITOR Hint

The `MONITOR` hint forces real-time SQL monitoring for the query, even if the statement is not long running. This hint is valid only when the parameter `CONTROL_MANAGEMENT_PACK_ACCESS` is set to `DIAGNOSTIC+TUNING`.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about real-time SQL monitoring

NATIVE_FULL_OUTER_JOIN Hint

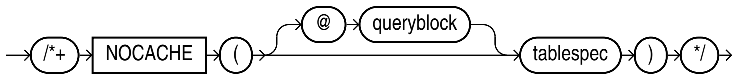
The `NATIVE_FULL_OUTER_JOIN` hint instructs the optimizer to use native full outer join, which is a native execution method based on a hash join.

> **See Also:**
>
> - NO_NATIVE_FULL_OUTER_JOIN Hint
> - *Oracle Database SQL Tuning Guide* for more information about native full outer joins

NOAPPEND Hint

The `NOAPPEND` hint instructs the optimizer to use conventional `INSERT` by disabling parallel mode for the duration of the `INSERT` statement. Conventional `INSERT` is the default in serial mode, and direct-path `INSERT` is the default in parallel mode.

NOCACHE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NOCACHE` hint instructs the optimizer to place the blocks retrieved for the table at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache. For example:

```
SELECT /*+ FULL(hr_emp) NOCACHE(hr_emp) */ last_name
  FROM employees hr_emp;
```

The `CACHE` and `NOCACHE` hints affect system statistics `table scans(long tables)` and `table scans(short tables)`, as shown in the `V$SYSSTAT` view.

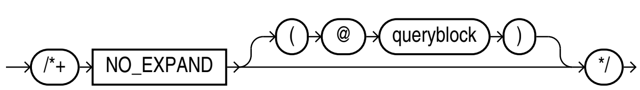
NO_CLUSTERING Hint

This hint is valid only for `INSERT` and `MERGE` operations on tables that are enabled for attribute clustering. The `NO_CLUSTERING` hint disables attribute clustering for direct-path inserts (serial or parallel). This hint overrides a `YES ON LOAD` setting in the DDL that created or altered the table. This hint has no effect on tables that are not enabled for attribute clustering.

See Also:

- clustering_when clause of `CREATE TABLE` for more information on the `YES ON LOAD` setting
- CLUSTERING Hint

NO_EXPAND Hint



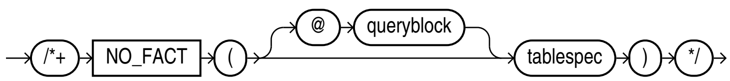
(See Specifying a Query Block in a Hint)

The `NO_EXPAND` hint instructs the optimizer not to consider `OR`-expansion for queries having `OR` conditions or `IN`-lists in the `WHERE` clause. Usually, the optimizer considers using `OR` expansion and uses this method if it decides that the cost is lower than not using it. For example:

```
SELECT /*+ NO_EXPAND */ *
  FROM employees e, departments d
  WHERE e.manager_id = 108
     OR d.department_id = 110;
```

> **See Also:**
>
> The USE_CONCAT Hint , which is the opposite of this hint

NO_FACT Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_FACT` hint is used in the context of the star transformation. It instruct the optimizer that the queried table should not be considered as a fact table.

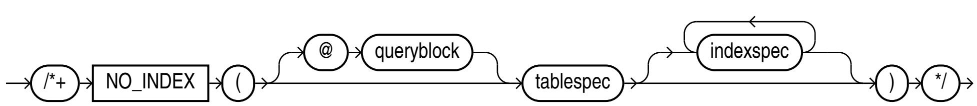
NO_GATHER_OPTIMIZER_STATISTICS Hint

The `NO_GATHER_OPTIMIZER_STATISTICS` hint instructs the optimizer to disable statistics gathering during the following types of bulk loads:

- `CREATE TABLE AS SELECT`
- `INSERT INTO` ... `SELECT` into an empty table using a direct path insert

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information on statistics gathering for bulk loads

NO_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The NO_INDEX hint instructs the optimizer not to use one or more indexes for the specified table. For example:

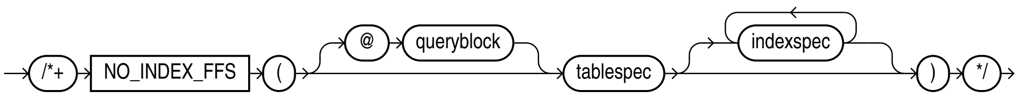
```
SELECT /*+ NO_INDEX(employees emp_empid) */ employee_id
  FROM employees
  WHERE employee_id > 200;
```

Each parameter serves the same purpose as in INDEX Hint with the following modifications:

- If this hint specifies a single available index, then the optimizer does not consider a scan on this index. Other indexes not specified are still considered.
- If this hint specifies a list of available indexes, then the optimizer does not consider a scan on any of the specified indexes. Other indexes not specified in the list are still considered.
- If this hint specifies no indexes, then the optimizer does not consider a scan on any index on the table. This behavior is the same as a NO_INDEX hint that specifies a list of all available indexes for the table.

The NO_INDEX hint applies to function-based, B-tree, bitmap, cluster, or domain indexes. If a NO_INDEX hint and an index hint (INDEX, INDEX_ASC, INDEX_DESC, INDEX_COMBINE, or INDEX_FFS) both specify the same indexes, then the database ignores both the NO_INDEX hint and the index hint for the specified indexes and considers those indexes for use during execution of the statement.

NO_INDEX_FFS Hint

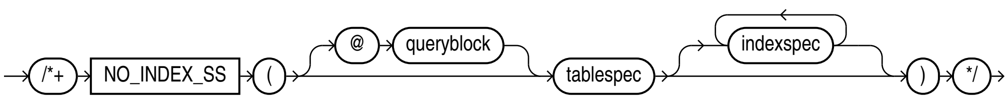


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The NO_INDEX_FFS hint instructs the optimizer to exclude a fast full index scan of the specified indexes on the specified table. Each parameter serves the same purpose as in the NO_INDEX Hint . For example:

```
SELECT /*+ NO_INDEX_FFS(items item_order_ix) */ order_id
  FROM order_items items;
```

NO_INDEX_SS Hint

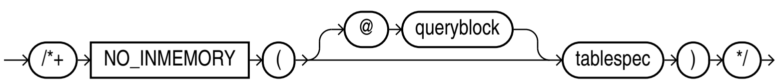


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The NO_INDEX_SS hint instructs the optimizer to exclude a skip scan of the specified indexes on the specified table. Each parameter serves the same purpose as in the NO_INDEX Hint .

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

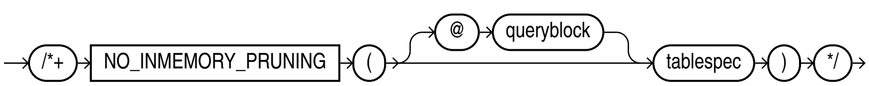
NO_INMEMORY Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_INMEMORY` hint disables In-Memory queries.

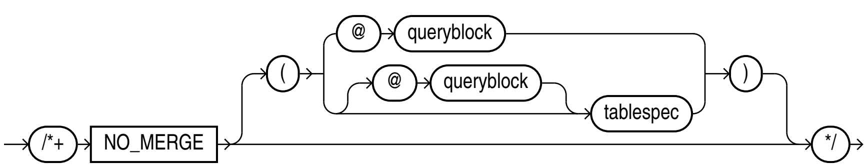
NO_INMEMORY_PRUNING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_INMEMORY_PRUNING` hint disables pruning of In-Memory queries.

NO_MERGE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_MERGE` hint instructs the optimizer not to combine the outer query and any inline view queries into a single query.

This hint lets you have more influence over the way in which the view is accessed. For example, the following statement causes view `seattle_dept` not to be merged:

```
SELECT /*+ NO_MERGE(seattle_dept) */ e1.last_name, seattle_dept.department_name
  FROM employees e1,
       (SELECT location_id, department_id, department_name
          FROM departments
          WHERE location_id = 1700) seattle_dept
  WHERE e1.department_id = seattle_dept.department_id;
```

When you use the NO_MERGE hint in the view query block, specify it without an argument. When you specify NO_MERGE in the surrounding query, specify it with the view name as an argument.

NO_MONITOR Hint

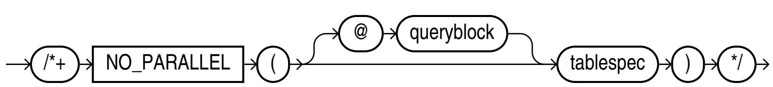
The NO_MONITOR hint disables real-time SQL monitoring for the query, even if the query is long running.

NO_NATIVE_FULL_OUTER_JOIN Hint

The NO_NATIVE_FULL_OUTER_JOIN hint instructs the optimizer to exclude the native execution method when joining each specified table. Instead, the full outer join is executed as a union of left outer join and anti-join.

> **See Also:**
>
> NATIVE_FULL_OUTER_JOIN Hint

NO_PARALLEL Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The NO_PARALLEL hint instructs the optimizer to run the statement serially. This hint overrides the value of the PARALLEL_DEGREE_POLICY initialization parameter. It also overrides a PARALLEL parameter in the DDL that created or altered the table. For example, the following SELECT statement will run serially:

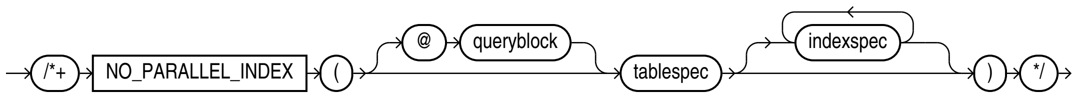
```
ALTER TABLE employees PARALLEL 8;
SELECT /*+ NO_PARALLEL(hr_emp) */ last_name
  FROM employees hr_emp;
```

> **See Also:**
>
> - Note on Parallel Hints for more information on the parallel hints
> - *Oracle Database Reference* for more information on the `PARALLEL_DEGREE_POLICY` initialization parameter

NOPARALLEL Hint

The `NOPARALLEL` hint has been deprecated. Use the `NO_PARALLEL` hint instead.

NO_PARALLEL_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

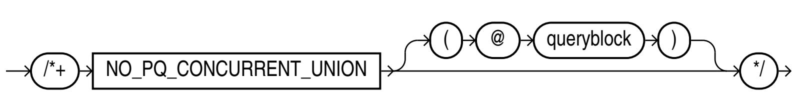
The `NO_PARALLEL_INDEX` hint overrides a `PARALLEL` parameter in the DDL that created or altered the index, thus avoiding a parallel index scan operation.

> **See Also:**
>
> Note on Parallel Hints for more information on the parallel hints

NOPARALLEL_INDEX Hint

The `NOPARALLEL_INDEX` hint has been deprecated. Use the `NO_PARALLEL_INDEX` hint instead.

NO_PQ_CONCURRENT_UNION Hint

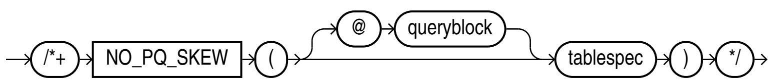


(See Specifying a Query Block in a Hint)

The `NO_PQ_CONCURRENT_UNION` hint instructs the optimizer to disable concurrent processing of `UNION` and `UNION ALL` operations.

> **See Also:**
>
> - PQ_CONCURRENT_UNION Hint
> - *Oracle Database VLDB and Partitioning Guide* for information about using this hint

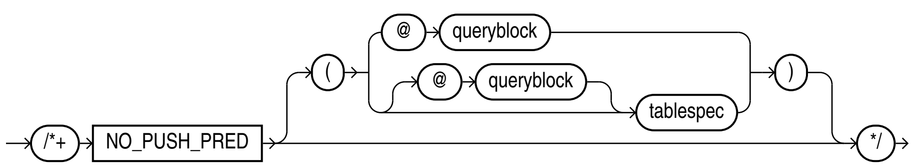
NO_PQ_SKEW Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_PQ_SKEW` hint advises the optimizer that the distribution of the values of the join keys for a parallel join is not skewed—that is, a high percentage of rows do not have the same join key values. The table specified in *`tablespec`* is the probe table of the hash join.

NO_PUSH_PRED Hint

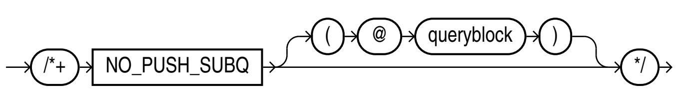


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_PUSH_PRED` hint instructs the optimizer not to push a join predicate into the view. For example:

```
SELECT /*+ NO_MERGE(v) NO_PUSH_PRED(v) */ *
  FROM employees e,
       (SELECT manager_id
          FROM employees) v
  WHERE e.manager_id = v.manager_id(+)
    AND e.employee_id = 100;
```

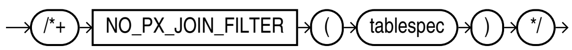
NO_PUSH_SUBQ Hint



(See Specifying a Query Block in a Hint)

The NO_PUSH_SUBQ hint instructs the optimizer to evaluate nonmerged subqueries as the last step in the execution plan. Doing so can improve performance if the subquery is relatively expensive or does not reduce the number of rows significantly.

NO_PX_JOIN_FILTER Hint



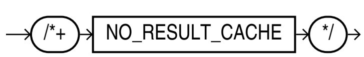
This hint prevents the optimizer from using parallel join bitmap filtering.

NO_QUERY_TRANSFORMATION Hint

The NO_QUERY_TRANSFORMATION hint instructs the optimizer to skip all query transformations, including but not limited to OR-expansion, view merging, subquery unnesting, star transformation, and materialized view rewrite. For example:

```
SELECT /*+ NO_QUERY_TRANSFORMATION */ employee_id, last_name
  FROM (SELECT * FROM employees e) v
  WHERE v.last_name = 'Smith';
```

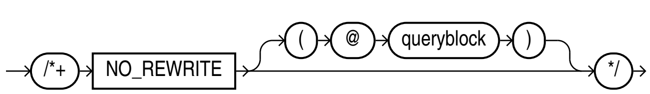
NO_RESULT_CACHE Hint



The optimizer caches query results in the result cache if the RESULT_CACHE_MODE initialization parameter is set to FORCE. In this case, the NO_RESULT_CACHE hint disables such caching for the current query.

If the query is executed from OCI client and OCI client result cache is enabled, then the NO_RESULT_CACHE hint disables caching for the current query.

NO_REWRITE Hint



(See Specifying a Query Block in a Hint)

The NO_REWRITE hint instructs the optimizer to disable query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED. For example:

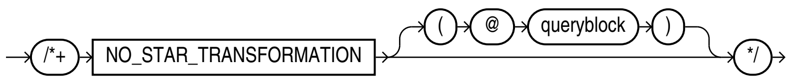
```
SELECT /*+ NO_REWRITE */ sum(s.amount_sold) AS dollars
  FROM sales s, times t
```

```
  WHERE s.time_id = t.time_id
  GROUP BY t.calendar_month_desc;
```

NOREWRITE Hint

The NOREWRITE hint has been deprecated. Use the NO_REWRITE hint instead.

NO_STAR_TRANSFORMATION Hint



(See Specifying a Query Block in a Hint)

The NO_STAR_TRANSFORMATION hint instructs the optimizer not to perform star query transformation.

NO_STATEMENT_QUEUING Hint

The NO_STATEMENT_QUEUING hint influences whether or not a statement is queued with parallel statement queuing.

When PARALLEL_DEGREE_POLICY is set to AUTO, this hint enables a statement to bypass the parallel statement queue. However, a statement that bypasses the statement queue can potentially cause the system to exceed the maximum number of parallel execution servers defined by the value of the PARALLEL_SERVERS_TARGET initialization parameter, which determines the limit at which parallel statement queuing is initiated.

There is no guarantee that the statement that bypasses the parallel statement queue receives the number of parallel execution servers requested because only the number of parallel execution servers available on the system, up to the value of the PARALLEL_MAX_SERVERS initialization parameter, can be allocated.

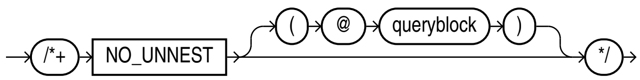
For example:

```
SELECT /*+ NO_STATEMENT_QUEUING */ emp.last_name, dpt.department_name
  FROM employees emp, departments dpt
  WHERE emp.department_id = dpt.department_id;
```

See Also:

STATEMENT_QUEUING Hint

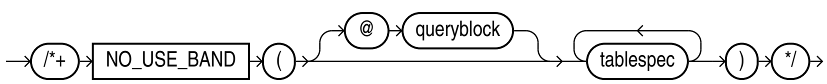
NO_UNNEST Hint



(See Specifying a Query Block in a Hint)

Use of the `NO_UNNEST` hint turns off unnesting .

NO_USE_BAND Hint

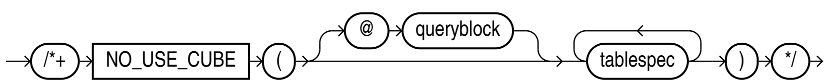


(See Specifying a Query Block in a Hint , *tablespec::=*)

The `NO_USE_BAND` hint instructs the optimizer to exclude band joins when joining each specified table to another row source. For example:

```
SELECT /*+ NO_USE_BAND(e1 e2) */
  e1.last_name
  || ' has salary between 100 less and 100 more than '
  || e2.last_name AS "SALARY COMPARISON"
FROM employees e1, employees e2
WHERE e1.salary BETWEEN e2.salary - 100 AND e2.salary + 100;
```

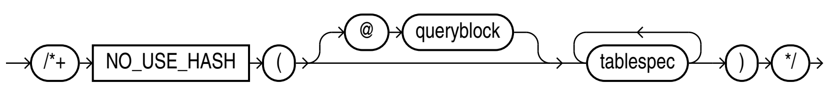
NO_USE_CUBE Hint



(See Specifying a Query Block in a Hint , *tablespec::=*)

The `NO_USE_CUBE` hint instructs the optimizer to exclude cube joins when joining each specified table to another row source using the specified table as the inner table.

NO_USE_HASH Hint

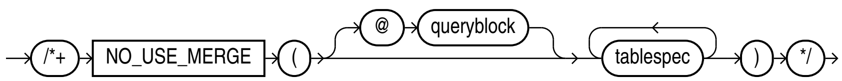


(See Specifying a Query Block in a Hint , *tablespec::=*)

The NO_USE_HASH hint instructs the optimizer to exclude hash joins when joining each specified table to another row source using the specified table as the inner table. For example:

```
SELECT /*+ NO_USE_HASH(e d) */ *
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;
```

NO_USE_MERGE Hint

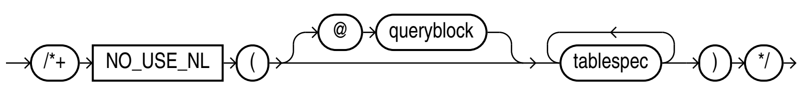


(See Specifying a Query Block in a Hint , *tablespec*::=)

The NO_USE_MERGE hint instructs the optimizer to exclude sort-merge joins when joining each specified table to another row source using the specified table as the inner table. For example:

```
SELECT /*+ NO_USE_MERGE(e d) */ *
   FROM employees e, departments d
   WHERE e.department_id = d.department_id
   ORDER BY d.department_id;
```

NO_USE_NL Hint



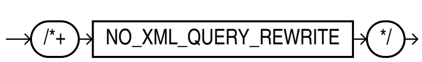
(See Specifying a Query Block in a Hint , *tablespec*::=)

The NO_USE_NL hint instructs the optimizer to exclude nested loops joins when joining each specified table to another row source using the specified table as the inner table. For example:

```
SELECT /*+ NO_USE_NL(l h) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 2400;
```

When this hint is specified, only hash join and sort-merge joins are considered for the specified tables. However, in some cases tables can be joined only by using nested loops. In such cases, the optimizer ignores the hint for those tables.

NO_XML_QUERY_REWRITE Hint



The NO_XML_QUERY_REWRITE hint instructs the optimizer to prohibit the rewriting of XPath expressions in SQL statements. By prohibiting the rewriting of XPath expressions, this hint also prohibits the use of any XMLIndexes for the current query. For example:

```
SELECT /*+NO_XML_QUERY_REWRITE*/ XMLQUERY('<A/>' RETURNING CONTENT)
  FROM DUAL;
```

> **See Also:**
>
> NO_XMLINDEX_REWRITE Hint

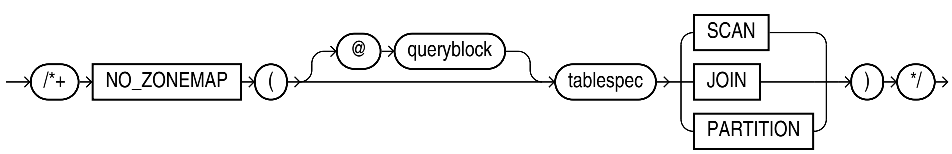
NO_XMLINDEX_REWRITE Hint

The NO_XMLINDEX_REWRITE hint instructs the optimizer not to use any XMLIndex indexes for the current query. For example:

```
SELECT /*+NO_XMLINDEX_REWRITE*/ count(*)
  FROM warehouses
  WHERE existsNode(warehouse_spec, '/Warehouse/Building') = 1;
```

> **See Also:**
>
> NO_XML_QUERY_REWRITE Hint for another way to disable the use of XMLIndexes

NO_ZONEMAP Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The NO_ZONEMAP hint disables the use of a zone map for different types of pruning. This hint overrides an ENABLE PRUNING setting in the DDL that created or altered the zone map.

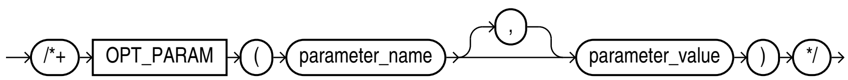
Specify one of the following options:

- SCAN - Disables the use of a zone map for scan pruning.

- `JOIN` - Disables the use of a zone map for join pruning.
- `PARTITION` - Disables the use of a zone map for partition pruning.

> **See Also:**
>
> - ENABLE | DISABLE PRUNING clause of `CREATE MATERIALIZED ZONEMAP`
> - *Oracle Database Data Warehousing Guide* for more information on pruning with zone maps

OPT_PARAM Hint



The `OPT_PARAM` hint lets you set an initialization parameter for the duration of the current query only. This hint is valid only for the following parameters: `APPROX_FOR_AGGREGATION`, `APPROX_FOR_COUNT_DISTINCT`, `APPROX_FOR_PERCENTILE`, `OPTIMIZER_DYNAMIC_SAMPLING`, `OPTIMIZER_INDEX_CACHING`, `OPTIMIZER_INDEX_COST_ADJ`, `OPTIMIZER_SECURE_VIEW_MERGING`, and `STAR_TRANSFORMATION_ENABLED`.

For example, the following hint sets the parameter `STAR_TRANSFORMATION_ENABLED` to `TRUE` for the statement to which it is added:

```
SELECT /*+ OPT_PARAM('star_transformation_enabled' 'true') */ *
  FROM ... ;
```

Parameter values that are strings are enclosed in single quotation marks. Numeric parameter values are specified without quotation marks.

ORDERED Hint

The `ORDERED` hint instructs Oracle to join tables in the order in which they appear in the `FROM` clause. Oracle recommends that you use the `LEADING` hint, which is more versatile than the `ORDERED` hint.

When you omit the `ORDERED` hint from a SQL statement requiring a join, the optimizer chooses the order in which to join the tables. You might want to use the `ORDERED` hint to specify a join order if you know something that the optimizer does not know about the number of rows selected from each table. Such information lets you choose an inner and outer table better than the optimizer could.

The following query is an example of the use of the `ORDERED` hint:

```
SELECT  /*+ ORDERED */ o.order_id, c.customer_id, l.unit_price * l.quantity
  FROM customers c, order_items l, orders o
```

```
WHERE c.cust_last_name = 'Taylor'
  AND o.customer_id = c.customer_id
  AND o.order_id = l.order_id;
```

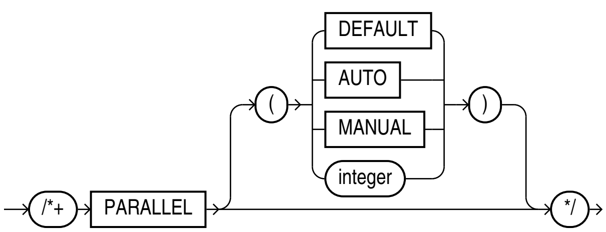
PARALLEL Hint

Note on Parallel Hints

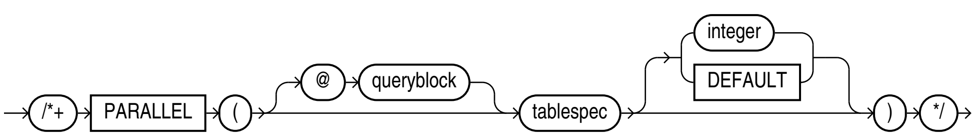
Beginning with Oracle Database 11*g* Release 2, the `PARALLEL` and `NO_PARALLEL` hints are statement-level hints and supersede the earlier object-level hints: `PARALLEL_INDEX`, `NO_PARALLEL_INDEX`, and previously specified `PARALLEL` and `NO_PARALLEL` hints. For `PARALLEL`, if you specify *`integer`*, then that degree of parallelism will be used for the statement. If you omit *`integer`*, then the database computes the degree of parallelism. All the access paths that can use parallelism will use the specified or computed degree of parallelism.

In the syntax diagrams below, *`parallel_hint_statement`* shows the syntax for statement-level hints, and *`parallel_hint_object`* shows the syntax for object-level hints. Object-level hints are supported for backward compatibility, and are superseded by statement-level hints.

***parallel_hint_statement*::=**



***parallel_hint_object*::=**



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `PARALLEL` hint instructs the optimizer to use the specified number of concurrent servers for a parallel operation. This hint overrides the value of the `PARALLEL_DEGREE_POLICY` initialization parameter. It applies to the `SELECT`, `INSERT`, `MERGE`, `UPDATE`, and `DELETE` portions of a statement, as well as to the table scan portion. If any parallel restrictions are violated, then the hint is ignored.

> **Note:**
>
> The number of servers that can be used is twice the value in the `PARALLEL` hint, if sorting or grouping operations also take place.

For a **statement-level PARALLEL hint**:

- `PARALLEL`: The statement always is run parallel, and the database computes the degree of parallelism, which can be 2 or greater.
- `PARALLEL` (`DEFAULT`): The optimizer calculates a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.
- `PARALLEL` (`AUTO`): The database computes the degree of parallelism, which can be 1 or greater. If the computed degree of parallelism is 1, then the statement runs serially.
- `PARALLEL` (`MANUAL`): The optimizer is forced to use the parallel settings of the objects in the statement.
- `PARALLEL` (*`integer`*): The optimizer uses the degree of parallelism specified by *`integer`*.

In the following example, the optimizer calculates the degree of parallelism. The statement always runs in parallel.

```
SELECT /*+ PARALLEL */ last_name
  FROM employees;
```

In the following example, the optimizer calculates the degree of parallelism, but that degree may be 1, in which case the statement will run serially.

```
SELECT /*+ PARALLEL (AUTO) */ last_name
  FROM employees;
```

In the following example, the `PARALLEL` hint advises the optimizer to use the degree of parallelism currently in effect for the table itself, which is 5:

```
CREATE TABLE parallel_table (col1 number, col2 VARCHAR2(10)) PARALLEL 5;

SELECT /*+ PARALLEL (MANUAL) */ col2
  FROM parallel_table;
```

For an **object-level PARALLEL hint**:

- `PARALLEL`: The query coordinator should examine the settings of the initialization parameters to determine the default degree of parallelism.
- `PARALLEL` (*`integer`*): The optimizer uses the degree of parallelism specified by *`integer`*.
- `PARALLEL` (`DEFAULT`): The optimizer calculates a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.

In the following example, the `PARALLEL` hint overrides the degree of parallelism specified in the `employees` table definition:

```
SELECT /*+ FULL(hr_emp) PARALLEL(hr_emp, 5) */ last_name
  FROM employees hr_emp;
```

In the next example, the `PARALLEL` hint overrides the degree of parallelism specified in the `employees` table definition and instructs the optimizer to calculate a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.

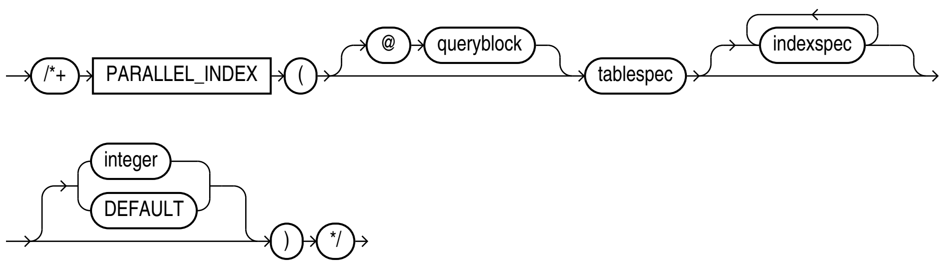
```
SELECT /*+ FULL(hr_emp) PARALLEL(hr_emp, DEFAULT) */ last_name
  FROM employees hr_emp;
```

Refer to CREATE TABLE and *Oracle Database Concepts* for more information on parallel execution.

See Also:

- CREATE TABLE and *Oracle Database Concepts* for more information on parallel execution.
- *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_PARALLEL_EXECUTE` package, which provides methods to apply table changes in chunks of rows. Changes to each chunk are independently committed when there are no errors.
- *Oracle Database Reference* for more information on the `PARALLEL_DEGREE_POLICY` initialization parameter
- NO_PARALLEL Hint

PARALLEL_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `PARALLEL_INDEX` hint instructs the optimizer to use the specified number of concurrent servers to parallelize index range scans, full scans, and fast full scans for partitioned indexes.

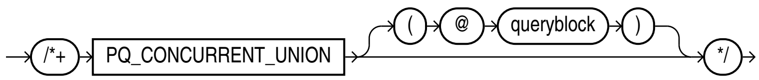
The *`integer`* value indicates the degree of parallelism for the specified index. Specifying `DEFAULT` or no value signifies that the query coordinator should examine the settings of the initialization parameters to determine the default degree of parallelism. For example, the following hint indicates three parallel execution processes are to be used:

```
SELECT /*+ PARALLEL_INDEX(table1, index1, 3) */
```

See Also:

Note on Parallel Hints for more information on the parallel hints

PQ_CONCURRENT_UNION Hint



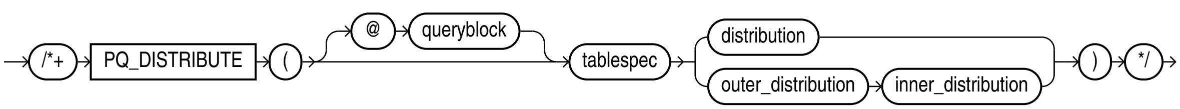
(See Specifying a Query Block in a Hint)

The PQ_CONCURRENT_UNION hint instructs the optimizer to enable concurrent processing of UNION and UNION ALL operations.

See Also:

- NO_PQ_CONCURRENT_UNION Hint
- *Oracle Database VLDB and Partitioning Guide* for information about using this hint

PQ_DISTRIBUTE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The PQ_DISTRIBUTE hint instructs the optimizer how to distribute rows among producer and consumer query servers. You can control the distribution of rows for either joins or for load.

Control of Distribution for Load

You can control the distribution of rows for parallel INSERT ... SELECT and parallel CREATE TABLE ... AS SELECT statements to direct how rows should be distributed between the producer (query) and the consumer (load) servers. Use the upper branch of the syntax by specifying a single distribution method. The values of the distribution methods and their semantics are described in Table 2-24.

Table 2-24 Distribution Values for Load

Distribution	Description
`NONE`	No distribution. That is the query and load operation are combined into each query server. All servers will load all partitions. This lack of distribution is useful to avoid the overhead of distributing rows where there is no skew. Skew can occur due to empty segments or to a predicate in the statement that filters out all rows evaluated by the query. If skew occurs due to using this method, then use either `RANDOM` or `RANDOM_LOCAL` distribution instead. **Note**: Use this distribution with care. Each partition loaded requires a minimum of 512 KB per process of PGA memory. If you also use compression, then approximately 1.5 MB of PGA memory is consumer per server.
`PARTITION`	This method uses the partitioning information of *`tablespec`* to distribute the rows from the query servers to the load servers. Use this distribution method when it is not possible or desirable to combine the query and load operations, when the number of partitions being loaded is greater than or equal to the number of load servers, and the input data will be evenly distributed across the partitions being loaded—that is, there is no skew.
`RANDOM`	This method distributes the rows from the producers in a round-robin fashion to the consumers. Use this distribution method when the input data is highly skewed.
`RANDOM_LOCAL`	This method distributes the rows from the producers to a set of servers that are responsible for maintaining a given set of partitions. Two or more servers can be loading the same partition, but no servers are loading all partitions. Use this distribution method when the input data is skewed and combining query and load operations is not possible due to memory constraints.

For example, in the following direct-path insert operation, the query and load portions of the operation are combined into each query server:

```
INSERT /*+ APPEND PARALLEL(target_table, 16) PQ_DISTRIBUTE(target_table, NONE) */
  INTO target_table
  SELECT * FROM source_table;
```

In the following table creation example, the optimizer uses the partitioning of target_table to distribute the rows:

```
CREATE /*+ PQ_DISTRIBUTE(target_table, PARTITION) */ TABLE target_table
  NOLOGGING PARALLEL 16
  PARTITION BY HASH (l_orderkey) PARTITIONS 512
  AS SELECT * FROM source_table;
```

Control of Distribution for Joins

You control the distribution method for joins by specifying two distribution methods, as shown in the lower branch of the syntax diagram, one distribution for the outer table and one distribution for the inner table.

- *`outer_distribution`* is the distribution for the outer table.

- *inner_distribution* is the distribution for the inner table.

The values of the distributions are HASH, BROADCAST, PARTITION, and NONE. Only six combinations table distributions are valid, as described in Table 2-25:

Table 2-25   Distribution Values for Joins

Distribution	Description
HASH, HASH	The rows of each table are mapped to consumer query servers, using a hash function on the join keys. When mapping is complete, each query server performs the join between a pair of resulting partitions. This distribution is recommended when the tables are comparable in size and the join operation is implemented by hash-join or sort merge join.
BROADCAST, NONE	All rows of the outer table are broadcast to each query server. The inner table rows are randomly partitioned. This distribution is recommended when the outer table is very small compared with the inner table. As a general rule, use this distribution when the inner table size multiplied by the number of query servers is greater than the outer table size.
NONE, BROADCAST	All rows of the inner table are broadcast to each consumer query server. The outer table rows are randomly partitioned. This distribution is recommended when the inner table is very small compared with the outer table. As a general rule, use this distribution when the inner table size multiplied by the number of query servers is less than the outer table size.
PARTITION, NONE	The rows of the outer table are mapped using the partitioning of the inner table. The inner table must be partitioned on the join keys. This distribution is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers; for example, 14 partitions and 15 query servers. **Note**: The optimizer ignores this hint if the inner table is not partitioned or not equijoined on the partitioning key.
NONE, PARTITION	The rows of the inner table are mapped using the partitioning of the outer table. The outer table must be partitioned on the join keys. This distribution is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers; for example, 14 partitions and 15 query servers. **Note**: The optimizer ignores this hint if the outer table is not partitioned or not equijoined on the partitioning key.
NONE, NONE	Each query server performs the join operation between a pair of matching partitions, one from each table. Both tables must be equipartitioned on the join keys.

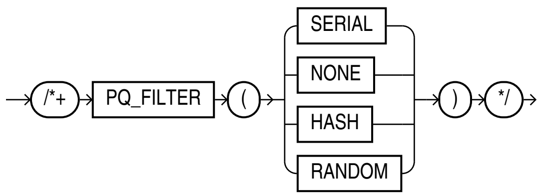
For example, given two tables r and s that are joined using a hash join, the following query contains a hint to use hash distribution:

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s HASH, HASH) USE_HASH (s)*/ column_list
  FROM r,s
  WHERE r.c=s.c;
```

To broadcast the outer table r, the query is:

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s BROADCAST, NONE) USE_HASH (s) */ column_list
  FROM r,s
  WHERE r.c=s.c;
```

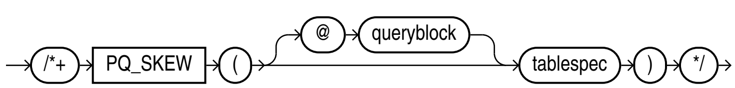
PQ_FILTER Hint



The PQ_FILTER hint instructs the optimizer on how to process rows when filtering correlated subqueries.

- SERIAL: Process rows serially on the left and right sides of the filter. Use this option when the overhead of parallelization is too high for the query, for example, when the left side has very few rows.
- NONE: Process rows in parallel on the left and right sides of the filter. Use this option when there is no skew in the distribution of the data on the left side of the filter and you would like to avoid distribution of the left side, for example, due to the large size of the left side.
- HASH: Process rows in parallel on the left side of the filter using a hash distribution. Process rows serially on the right side of the filter. Use this option when there is no skew in the distribution of data on the left side of the filter.
- RANDOM: Process rows in parallel on the left side of the filter using a random distribution. Process rows serially on the right side of the filter. Use this option when there is skew in the distribution of data on the left side of the filter.

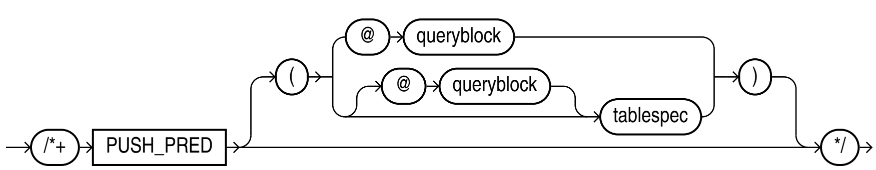
PQ_SKEW Hint



(See Specifying a Query Block in a Hint , *tablespec::=*)

The PQ_SKEW hint advises the optimizer that the distribution of the values of the join keys for a parallel join is highly skewed—that is, a high percentage of rows have the same join key values. The table specified in *tablespec* is the probe table of the hash join.

PUSH_PRED Hint

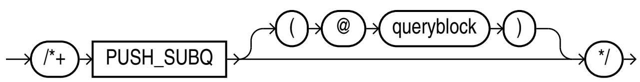


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `PUSH_PRED` hint instructs the optimizer to push a join predicate into the view. For example:

```
SELECT /*+ NO_MERGE(v) PUSH_PRED(v) */ *
  FROM employees e,
    (SELECT manager_id
      FROM employees) v
  WHERE e.manager_id = v.manager_id(+)
    AND e.employee_id = 100;
```

PUSH_SUBQ Hint

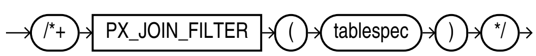


(See Specifying a Query Block in a Hint)

The `PUSH_SUBQ` hint instructs the optimizer to evaluate nonmerged subqueries at the earliest possible step in the execution plan. Generally, subqueries that are not merged are executed as the last step in the execution plan. If the subquery is relatively inexpensive and reduces the number of rows significantly, then evaluating the subquery earlier can improve performance.

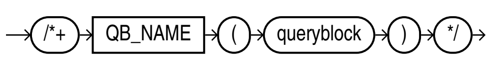
This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.

PX_JOIN_FILTER Hint



This hint forces the optimizer to use parallel join bitmap filtering.

QB_NAME Hint



(See Specifying a Query Block in a Hint)

Use the `QB_NAME` hint to define a name for a query block. This name can then be used in a hint in the outer query or even in a hint in an inline view to affect query execution on the tables appearing in the named query block.

If two or more query blocks have the same name, or if the same query block is hinted twice with different names, then the optimizer ignores all the names and the hints referencing that query block. Query blocks that are not named using this hint have unique system-generated names. These names can be displayed in the plan table and can also be used in hints within the query block, or in query block hints. For example:

```
SELECT /*+ QB_NAME(qb) FULL(@qb e) */ employee_id, last_name
  FROM employees e
  WHERE last_name = 'Smith';
```

RESULT_CACHE Hint

The `RESULT_CACHE` hint instructs the database to cache the results of the current query or query fragment in memory and then to use the cached results in future executions of the query or query fragment. The hint is recognized in the top-level query, the *`subquery_factoring_clause`*, or `FROM` clause inline view. The cached results reside in the result cache memory portion of the shared pool.

A cached result is automatically invalidated whenever a database object used in its creation is successfully modified. This hint takes precedence over settings of the `RESULT_CACHE_MODE` initialization parameter.

The query is eligible for result caching only if all functions entailed in the query—for example, built-in or user-defined functions or virtual columns—are deterministic.

If the query is executed from OCI client and OCI client result cache is enabled, then `RESULT_CACHE` hint enables client caching for the current query.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for information about using this hint, *Oracle Database Reference* for information about the `RESULT_CACHE_MODE` initialization parameter, and *Oracle Call Interface Programmer's Guide* for more information about the OCI result cache and usage guidelines

RETRY_ON_ROW_CHANGE Hint

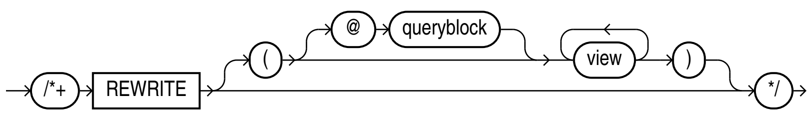
> **Note:**
>
> The `CHANGE_DUPKEY_ERROR_INDEX`, `IGNORE_ROW_ON_DUPKEY_INDEX`, and `RETRY_ON_ROW_CHANGE` hints are unlike other hints in that they have a semantic effect. The general philosophy explained in Hints does not apply for these three hints.

This hint is valid only for `UPDATE` and `DELETE` operations. It is not supported for `INSERT` or `MERGE` operations. When you specify this hint, the operation is retried when the

ORA_ROWSCN for one or more rows in the set has changed from the time the set of rows to be modified is determined to the time the block is actually modified.

> **See Also:**
>
> IGNORE_ROW_ON_DUPKEY_INDEX Hint and CHANGE_DUPKEY_ERROR_INDEX Hint

REWRITE Hint



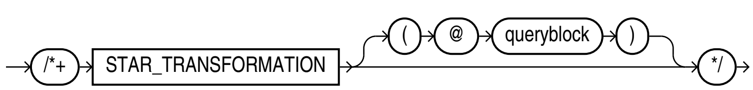
(See Specifying a Query Block in a Hint)

The REWRITE hint instructs the optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost.

Oracle does not consider views outside of the list. If you do not specify a view list, then Oracle searches for an eligible materialized view and always uses it regardless of the cost of the final plan.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on materialized views
> - *Oracle Database Data Warehousing Guide* for more information on using REWRITE with materialized views

STAR_TRANSFORMATION Hint



(See Specifying a Query Block in a Hint)

The STAR_TRANSFORMATION hint instructs the optimizer to use the best plan in which the transformation has been used. Without the hint, the optimizer could make a query optimization decision to use the best plan generated without the transformation, instead of the best plan for the transformed query. For example:

```
SELECT /*+ STAR_TRANSFORMATION */ s.time_id, s.prod_id, s.channel_id
  FROM sales s, times t, products p, channels c
```

```
  WHERE s.time_id = t.time_id
    AND s.prod_id = p.prod_id
    AND s.channel_id = c.channel_id
    AND c.channel_desc = 'Tele Sales';
```

Even if the hint is specified, there is no guarantee that the transformation will take place. The optimizer generates the subqueries only if it seems reasonable to do so. If no subqueries are generated, then there is no transformed query, and the best plan for the untransformed query is used, regardless of the hint.

See Also:

- *Oracle Database Data Warehousing Guide* for a full discussion of star transformation.
- *Oracle Database Reference* for more information on the `STAR_TRANSFORMATION_ENABLED` initialization parameter.

STATEMENT_QUEUING Hint

The `NO_STATEMENT_QUEUING` hint influences whether or not a statement is queued with parallel statement queuing.

When `PARALLEL_DEGREE_POLICY` is not set to `AUTO`, this hint enables a statement to be considered for parallel statement queuing, but to run only when enough parallel processes are available to run at the requested DOP. The number of available parallel execution servers, before queuing is enabled, is equal to the difference between the number of parallel execution servers in use and the maximum number allowed in the system, which is defined by the `PARALLEL_SERVERS_TARGET` initialization parameter.

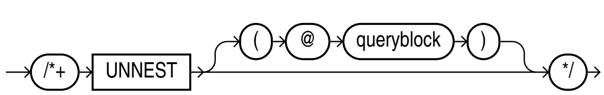
For example:

```
SELECT /*+ STATEMENT_QUEUING */ emp.last_name, dpt.department_name
  FROM employees emp, departments dpt
  WHERE emp.department_id = dpt.department_id;
```

See Also:

NO_STATEMENT_QUEUING Hint

UNNEST Hint



(See Specifying a Query Block in a Hint)

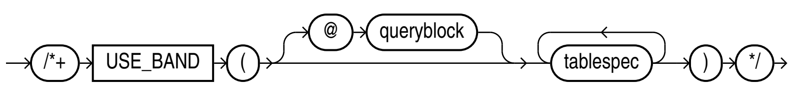
The UNNEST hint instructs the optimizer to unnest and merge the body of the subquery into the body of the query block that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

Before a subquery is unnested, the optimizer first verifies whether the statement is valid. The statement must then pass heuristic and query optimization tests. The UNNEST hint instructs the optimizer to check the subquery block for validity only. If the subquery block is valid, then subquery unnesting is enabled without checking the heuristics or costs.

See Also:

- Collection Unnesting: Examples for more information on unnesting nested subqueries and the conditions that make a subquery block valid
- *Oracle Database SQL Tuning Guide* for additional information on subquery unnesting

USE_BAND Hint

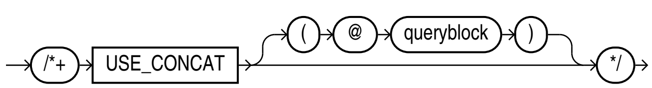


(See Specifying a Query Block in a Hint , *tablespec*::=)

The USE_BAND hint instructs the optimizer to join each specified table with another row source using a band join. For example:

```
SELECT /*+ USE_BAND(e1 e2) */
  e1.last_name
  || ' has salary between 100 less and 100 more than '
  || e2.last_name AS "SALARY COMPARISON"
FROM employees e1, employees e2
WHERE e1.salary BETWEEN e2.salary - 100 AND e2.salary + 100;
```

USE_CONCAT Hint



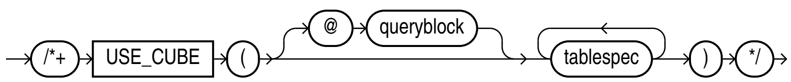
(See Specifying a Query Block in a Hint)

The USE_CONCAT hint instructs the optimizer to transform combined OR-conditions in the WHERE clause of a query into a compound query using the UNION ALL set operator. Without this hint, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them. The USE_CONCAT hint overrides the cost consideration. For example:

```
SELECT /*+ USE_CONCAT */ *
  FROM employees e
  WHERE manager_id = 108
     OR department_id = 110;
```

> **See Also:**
>
> The NO_EXPAND Hint , which is the opposite of this hint

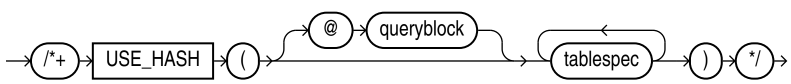
USE_CUBE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

When the right-hand side of the join is a cube, the `USE_CUBE` hint instructs the optimizer to join each specified table with another row source using a cube join. If the optimizer decides not to use the cube join based on statistical analysis, then you can use `USE_CUBE` to override that decision.

USE_HASH Hint

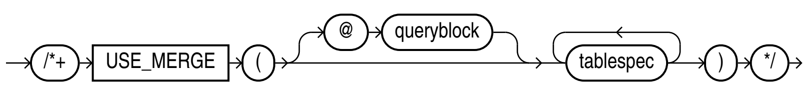


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `USE_HASH` hint instructs the optimizer to join each specified table with another row source using a hash join. For example:

```
SELECT /*+ USE_HASH(l h) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 2400;
```

USE_MERGE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

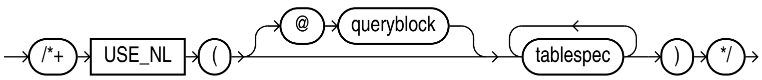
The `USE_MERGE` hint instructs the optimizer to join each specified table with another row source using a sort-merge join. For example:

```
SELECT /*+ USE_MERGE(employees departments) */ *
  FROM employees, departments
  WHERE employees.department_id = departments.department_id;
```

Use of the `USE_NL` and `USE_MERGE` hints is recommended with the `LEADING` and `ORDERED` hints. The optimizer uses those hints when the referenced table is forced to be the inner table of a join. The hints are ignored if the referenced table is the outer table.

USE_NL Hint

The `USE_NL` hint instructs the optimizer to join each specified table to another row source with a nested loops join, using the specified table as the inner table.



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `USE_NL` hint instructs the optimizer to join each specified table to another row source with a nested loops join, using the specified table as the inner table.

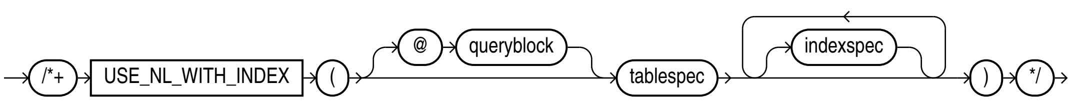
Use of the `USE_NL` and `USE_MERGE` hints is recommended with the `LEADING` and `ORDERED` hints. The optimizer uses those hints when the referenced table is forced to be the inner table of a join. The hints are ignored if the referenced table is the outer table.

In the following example, where a nested loop is forced through a hint, `orders` is accessed through a full table scan and the filter condition `l.order_id = h.order_id` is applied to every row. For every row that meets the filter condition, `order_items` is accessed through the index `order_id`.

```
SELECT /*+ USE_NL(l h) */ h.customer_id, l.unit_price * l.quantity
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id;
```

Adding an `INDEX` hint to the query could avoid the full table scan on `orders`, resulting in an execution plan similar to one used on larger systems, even though it might not be particularly efficient here.

USE_NL_WITH_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `USE_NL_WITH_INDEX` hint instructs the optimizer to join the specified table to another row source with a nested loops join using the specified table as the inner table. For example:

```
SELECT /*+ USE_NL_WITH_INDEX(l item_product_ix) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 2400;
```

The following conditions apply:

- If no index is specified, then the optimizer must be able to use some index with at least one join predicate as the index key.
- If an index is specified, then the optimizer must be able to use that index with at least one join predicate as the index key.

Database Objects

Oracle Database recognizes objects that are associated with a particular schema and objects that are not associated with any particular schema, as described in the sections that follow.

Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

Analytic views
Attribute dimensions
Clusters
Constraints
Database links
Database triggers
Dimensions
External procedure libraries
Hierarchies
Index-organized tables
Indexes
Indextypes
Java classes
Java resources
Java sources
Join groups
Materialized views
Materialized view logs
Mining models
Object tables
Object types
Object views
Operators
Packages
Sequences
Stored functions
Stored procedures
Synonyms
Tables
Views

Zone maps

Nonschema Objects

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

Contexts
Directories
Editions
Flashback archives
Lockdown profiles
Profiles
Restore points
Roles
Rollback segments
Tablespaces
Tablespace sets
Unified audit policies
Users

In this reference, each type of object is described in the section devoted to the statement that creates the database object. These statements begin with the keyword `CREATE`. For example, for the definition of a cluster, see CREATE CLUSTER .

> **See Also:**
>
> *Oracle Database Concepts* for an overview of database objects

You must provide names for most types of database objects when you create them. These names must follow the rules listed in the sections that follow.

Database Object Names and Qualifiers

Some database objects are made up of parts that you can or must name, such as the columns in a table or view, index and table partitions and subpartitions, integrity constraints on a table, and objects that are stored within a package, including procedures and stored functions. This section provides:

- Rules for naming database objects and database object location qualifiers
- Guidelines for naming database objects and qualifiers

> **Note:**
>
> Oracle uses system-generated names beginning with "`SYS_`" for implicitly generated database objects and subobjects, and names beginning with "`ORA_`" for some Oracle-supplied objects. Oracle discourages you from using these prefixes in the names you explicitly provide to your database objects and subobjects to avoid possible conflict in name resolution.

Database Object Naming Rules

Every database object has a name. In a SQL statement, you represent the name of an object with a **quoted identifier** or a **nonquoted identifier**.

- A quoted identifier begins and ends with double quotation marks ("). If you name a schema object using a quoted identifier, then you must use the double quotation marks whenever you refer to that object.
- A nonquoted identifier is not surrounded by any punctuation.

You can use either quoted or nonquoted identifiers to name any database object. However, database names, global database names, database link names, disk group names, and pluggable database (PDB) names are always case insensitive and are stored as uppercase. If you specify such names as quoted identifiers, then the quotation marks are silently ignored.

> **See Also:**
>
> CREATE USER for additional rules for naming users and passwords

> **Note:**
>
> Oracle does not recommend using quoted identifiers for database object names. These quoted identifiers are accepted by SQL*Plus, but they may not be valid when using other tools that manage database objects.

The following list of rules applies to both quoted and nonquoted identifiers unless otherwise indicated:

1. The maximum length of identifier names depends on the value of the `COMPATIBLE` initialization parameter.
   - **If COMPATIBLE is set to a value of 12.2 or higher,** then names must be from 1 to 128 bytes long with these exceptions:
     – Names of databases are limited to 8 bytes.
     – Names of disk groups, pluggable databases (PDBs), rollback segments, tablespaces, and tablespace sets are limited to 30 bytes.

If an identifier includes multiple parts separated by periods, then each attribute can be up to 128 bytes long. Each period separator, as well as any surrounding double quotation marks, counts as one byte. For example, suppose you identify a column like this:

```
"schema"."table"."column"
```

The schema name can be 128 bytes, the table name can be 128 bytes, and the column name can be 128 bytes. Each of the quotation marks and periods is a single-byte character, so the total length of the identifier in this example can be up to 392 bytes.

- **If COMPATIBLE is set to a value lower than 12.2,** then names must be from 1 to 30 bytes long with these exceptions:
  - Names of databases are limited to 8 bytes.
  - Names of database links can be as long as 128 bytes.

  If an identifier includes multiple parts separated by periods, then each attribute can be up to 30 bytes long. Each period separator, as well as any surrounding double quotation marks, counts as one byte. For example, suppose you identify a column like this:

  ```
  "schema"."table"."column"
  ```

  The schema name can be 30 bytes, the table name can be 30 bytes, and the column name can be 30 bytes. Each of the quotation marks and periods is a single-byte character, so the total length of the identifier in this example can be up to 98 bytes.

2. Nonquoted identifiers cannot be Oracle SQL reserved words. Quoted identifiers can be reserved words, although this is not recommended.

   Depending on the Oracle product you plan to use to access a database object, names might be further restricted by other product-specific reserved words.

   > **Note:**
   >
   > The reserved word `ROWID` is an exception to this rule. You cannot use the uppercase word `ROWID`, either quoted or nonquoted, as a column name. However, you can use the uppercase word as a quoted identifier that is not a column name, and you can use the word with one or more lowercase letters (for example, "`Rowid`" or "`rowid`") as any quoted identifier, including a column name.

   > **See Also:**
   >
   > - Oracle SQL Reserved Words for a listing of all Oracle SQL reserved words
   > - The manual for a specific product, such as *Oracle Database PL/SQL Language Reference*, for a list of the reserved words of that product

3. The Oracle SQL language contains other words that have special meanings. These words include data types, schema names, function names, the dummy system table `DUAL`, and keywords (the uppercase words in SQL statements, such as `DIMENSION`, `SEGMENT`, `ALLOCATE`, `DISABLE`, and so forth). These words are not reserved. However,

Oracle uses them internally in specific ways. Therefore, if you use these words as names for objects and object parts, then your SQL statements may be more difficult to read and may lead to unpredictable results.

In particular, do not use words beginning with `SYS_` or `ORA_` as schema object names, and do not use the names of SQL built-in functions for the names of schema objects or user-defined functions.

> **See Also:**
>
> - Oracle SQL Keywords for information how to obtain a list of keywords
> - Data Types , About SQL Functions , and Selecting from the DUAL Table

4. You should use characters from the ASCII repertoire in database names, global database names, and database link names, because these characters provide optimal compatibility across different platforms and operating systems. You must use only characters from the ASCII repertoire in names of common users and common roles in a multitenant container database (CDB).
5. You can include multibyte characters in passwords.
6. Nonquoted identifiers must begin with an alphabetic character from your database character set. Quoted identifiers can begin with any character.
7. Nonquoted identifiers can only contain alphanumeric characters from your database character set and the underscore (_), dollar sign ($), and pound sign (#). Database links can also contain periods (.) and "at" signs (@). Oracle strongly discourages you from using $ and # in nonquoted identifiers.

   Quoted identifiers can contain any characters and punctuations marks as well as spaces. However, neither quoted nor nonquoted identifiers can contain double quotation marks or the null character (`\0`).
8. Within a namespace, no two objects can have the same name.

   The following schema objects share one namespace:

   - Packages
   - Private synonyms
   - Sequences
   - Stand-alone procedures
   - Stand-alone stored functions
   - Tables
   - User-defined operators
   - User-defined types
   - Views

   Each of the following schema objects has its own namespace:

   - Clusters
   - Constraints

- Database triggers
- Dimensions
- Indexes
- Materialized views (When you create a materialized view, the database creates an internal table of the same name. This table has the same namespace as the other tables in the schema. Therefore, a schema cannot contain a table and a materialized view of the same name.)
- Private database links

Because tables and sequences are in the same namespace, a table and a sequence in the same schema cannot have the same name. However, tables and indexes are in different namespaces. Therefore, a table and an index in the same schema can have the same name.

Each schema in the database has its own namespaces for the objects it contains. This means, for example, that two tables in different schemas are in different namespaces and can have the same name.

Each of the following nonschema objects also has its own namespace:

- Editions
- Parameter files (`PFILE`s) and server parameter files (`SPFILE`s)
- Profiles
- Public database links
- Public synonyms
- Tablespaces
- User roles

Because the objects in these namespaces are not contained in schemas, these namespaces span the entire database.

9. Nonquoted identifiers are not case sensitive. Oracle interprets them as uppercase. Quoted identifiers are case sensitive.

   By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

```
"employees"
"Employees"
"EMPLOYEES"
```

   Note that Oracle interprets the following names the same, so they cannot be used for different objects in the same namespace:

```
employees
EMPLOYEES
"EMPLOYEES"
```

10. When Oracle stores or compares identifiers in uppercase, the uppercase form of each character in the identifiers is determined by applying the uppercasing rules of the database character set. Language-specific rules determined by the session setting `NLS_SORT` are not considered. This behavior corresponds to applying the SQL function `UPPER` to the identifier rather than the function `NLS_UPPER`.

   The database character set uppercasing rules can yield results that are incorrect when viewed as being in a certain natural language. For example, small letter sharp s ("ß"), used in German, does not have an uppercase form according to the database character set uppercasing rules. It is not modified when an identifier is converted into uppercase, while the expected uppercase form in German is the sequence of two characters capital letter S ("SS"). Similarly, the uppercase form of small letter i, according to the database character set uppercasing rules, is capital letter I. However, the expected uppercase form in Turkish and Azerbaijani is capital letter I with dot above.

The database character set uppercasing rules ensure that identifiers are interpreted the same in any linguistic configuration of a session. If you want an identifier to look correctly in a certain natural language, then you can quote it to preserve the lowercase form or you can use the linguistically correct uppercase form whenever you use that identifier.

11. Columns in the same table or view cannot have the same name. However, columns in different tables or views can have the same name.
12. Procedures or functions contained in the same package can have the same name, if their arguments are not of the same number and data types. Creating multiple procedures or functions with the same name in the same package with different arguments is called **overloading** the procedure or function.

Schema Object Naming Examples

The following examples are valid schema object names:

```
last_name
horse
hr.hire_date
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

All of these examples adhere to the rules listed in Database Object Naming Rules .

Schema Object Naming Guidelines

Here are several helpful guidelines for naming objects and their parts:

- Use full, descriptive, pronounceable names (or well-known abbreviations).
- Use consistent naming rules.
- Use the same name to describe the same entity or attribute across tables.

When naming objects, balance the objective of keeping names short and easy to use with the objective of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in the database may be used by many people over a period of time. Your counterpart ten years from now may have difficulty understanding a table column with a name like `pmdd` instead of `payment_due_date`.

Using consistent naming rules helps users understand the part that each table plays in your application. One such rule might be to begin the names of all tables belonging to the `FINANCE` application with `fin_`.

Use the same names to describe the same things across tables. For example, the department number columns of the sample `employees` and `departments` tables are both named `department_id`.

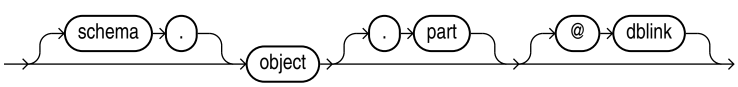
Syntax for Schema Objects and Parts in SQL Statements

This section tells you how to refer to schema objects and their parts in the context of a SQL statement. This section shows you:

- The general syntax for referring to an object
- How Oracle resolves a reference to an object
- How to refer to objects in schemas other than your own
- How to refer to objects in remote databases
- How to refer to table and index partitions and subpartitions

The following diagram shows the general syntax for referring to an object or a part:

***database_object_or_part*::=**



(*dblink::=*)

where:

- `object` is the name of the object.
- `schema` is the schema containing the object. The schema qualifier lets you refer to an object in a schema other than your own. You must be granted privileges to refer to objects in other schemas. If you omit `schema`, then Oracle assumes that you are referring to an object in your own schema.

  Only schema objects can be qualified with `schema`. Schema objects are shown with list item 8. Nonschema objects, also shown with list item 8, cannot be qualified with `schema` because they are not schema objects. An exception is public synonyms, which can optionally be qualified with "`PUBLIC`". The quotation marks are required.
- `part` is a part of the object. This identifier lets you refer to a part of a schema object, such as a column or a partition of a table. Not all types of objects have parts.
- `dblink` applies only when you are using the Oracle Database distributed functionality. This is the name of the database containing the object. The `dblink` qualifier lets you refer to an object in a database other than your local database. If you omit `dblink`, then Oracle assumes that you are referring to an object in your local database. Not all SQL statements allow you to access objects on remote databases.

You can include spaces around the periods separating the components of the reference to the object, but it is conventional to omit them.

How Oracle Database Resolves Schema Object References

When you refer to an object in a SQL statement, Oracle considers the context of the SQL statement and locates the object in the appropriate namespace. After locating the object, Oracle performs the operation specified by the statement on the object. If the named object cannot be found in the appropriate namespace, then Oracle returns an error.

The following example illustrates how Oracle resolves references to objects within SQL statements. Consider this statement that adds a row of data to a table identified by the name `departments`:

```
INSERT INTO departments
  VALUES (280, 'ENTERTAINMENT_CLERK', 206, 1700);
```

Based on the context of the statement, Oracle determines that `departments` can be:

- A table in your own schema
- A view in your own schema
- A private synonym for a table or view
- A public synonym

Oracle always attempts to resolve an object reference within the namespaces in your own schema before considering namespaces outside your schema. In this example, Oracle attempts to resolve the name `departments` as follows:

1. First, Oracle attempts to locate the object in the namespace in your own schema containing tables, views, and private synonyms. If the object is a private synonym, then Oracle locates the object for which the synonym stands. This object could be in your own schema, another schema, or on another database. The object could also be another synonym, in which case Oracle locates the object for which this synonym stands.
2. If the object is in the namespace, then Oracle attempts to perform the statement on the object. In this example, Oracle attempts to add the row of data to `departments`. If the object is not of the correct type for the statement, then Oracle returns an error. In this example, `departments` must be a table, view, or a private synonym resolving to a table or view. If `departments` is a sequence, then Oracle returns an error.
3. If the object is not in any namespace searched in thus far, then Oracle searches the namespace containing public synonyms. If the object is in that namespace, then Oracle attempts to perform the statement on it. If the object is not of the correct type for the statement, then Oracle returns an error. In this example, if `departments` is a public synonym for a sequence, then Oracle returns an error.

If a public synonym has any dependent tables or user-defined types, then you cannot create an object with the same name as the synonym in the same schema as the dependent objects.

If a synonym does not have any dependent tables or user-defined types, then you can create an object with the same name in the same schema as the dependent objects. Oracle invalidates any dependent objects and attempts to revalidate them when they are next accessed.

See Also:

Oracle Database PL/SQL Language Reference for information about how PL/SQL resolves identifier names

References to Objects in Other Schemas

To refer to objects in schemas other than your own, prefix the object name with the schema name:

```
schema.object
```

For example, this statement drops the `employees` table in the sample schema `hr`:

```
DROP TABLE hr.employees;
```

References to Objects in Remote Databases

To refer to objects in databases other than your local database, follow the object name with the name of the database link to that database. A database link is a schema object that causes Oracle to connect to a remote database to access an object there. This section tells you:

- How to create database links
- How to use database links in your SQL statements

Creating Database Links

You create a database link with the statement CREATE DATABASE LINK . The statement lets you specify this information about the database link:

- The name of the database link
- The database connect string to access the remote database
- The username and password to connect to the remote database

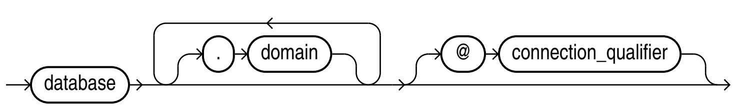
Oracle stores this information in the data dictionary.

Database Link Names

When you create a database link, you must specify its name. Database link names are different from names of other types of objects. They can be as long as 128 bytes and can contain periods (.) and the "at" sign (@).

The name that you give to a database link must correspond to the name of the database to which the database link refers and the location of that database in the hierarchy of database names. The following syntax diagram shows the form of the name of a database link:

***dblink*::=**



where:

- `database` should specify the `name` portion of the global name of the remote database to which the database link connects. This global name is stored in the data dictionary of the remote database. You can see this name in the `GLOBAL_NAME` data dictionary view.

- *domain* should specify the *domain* portion of the global name of the remote database to which the database link connects. If you omit *domain* from the name of a database link, then Oracle qualifies the database link name with the domain of your local database as it currently exists in the data dictionary.
- *connection_qualifier* lets you further qualify a database link. Using connection qualifiers, you can create multiple database links to the same database. For example, you can use connection qualifiers to create multiple database links to different instances of the Oracle Real Application Clusters that access the same database.

> **See Also:**
>
> *Oracle Database Administrator's Guide*for more information on connection qualifiers

The combination *database.domain* is sometimes called the **service name**.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide*

Username and Password

Oracle uses the username and password to connect to the remote database. The username and password for a database link are optional.

Database Connect String

The database connect string is the specification used by Oracle Net to access the remote database. For information on writing database connect strings, see the Oracle Net documentation for your specific network protocol. The database connect string for a database link is optional.

References to Database Links

Database links are available only if you are using Oracle distributed functionality. When you issue a SQL statement that contains a database link, you can specify the database link name in one of these forms:

- The complete database link name as stored in the data dictionary, including the *database*, *domain*, and optional *connection_qualifier* components.
- The *partial* database link name is the *database* and optional *connection_qualifier* components, but not the *domain* component.

Oracle performs these tasks before connecting to the remote database:

1. If the database link name specified in the statement is partial, then Oracle expands the name to contain the domain of the local database as found in the global database name stored in the data dictionary. (You can see the current global database name in the `GLOBAL_NAME` data dictionary view.)

2. Oracle first searches for a private database link in your own schema with the same name as the database link in the statement. Then, if necessary, it searches for a public database link with the same name.
   - Oracle always determines the username and password from the first matching database link (either private or public). If the first matching database link has an associated username and password, then Oracle uses it. If it does not have an associated username and password, then Oracle uses your current username and password.
   - If the first matching database link has an associated database string, then Oracle uses it. Otherwise Oracle searches for the next matching (public) database link. If no matching database link is found, or if no matching link has an associated database string, then Oracle returns an error.
3. Oracle uses the database string to access the remote database. After accessing the remote database, if the value of the `GLOBAL_NAMES` parameter is `true`, then Oracle verifies that the *`database.domain`* portion of the database link name matches the complete global name of the remote database. If this condition is true, then Oracle proceeds with the connection, using the username and password chosen in Step 2. If not, Oracle returns an error.
4. If the connection using the database string, username, and password is successful, then Oracle attempts to access the specified object on the remote database using the rules for resolving object references and referring to objects in other schemas discussed earlier in this section.

You can disable the requirement that the *`database.domain`* portion of the database link name must match the complete global name of the remote database by setting to `FALSE` the initialization parameter `GLOBAL_NAMES` or the `GLOBAL_NAMES` parameter of the `ALTER SYSTEM` or `ALTER SESSION` statement.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on remote name resolution

References to Partitioned Tables and Indexes

Tables and indexes can be partitioned. When partitioned, these schema objects consist of a number of parts called **partitions**, all of which have the same logical attributes. For example, all partitions in a table share the same column and constraint definitions, and all partitions in an index share the same index columns.

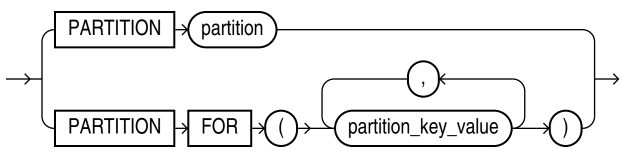
Partition-extended and subpartition-extended names let you perform some partition-level and subpartition-level operations, such as deleting all rows from a partition or subpartition, on only one partition or subpartition. Without extended names, such operations would require that you specify a predicate (`WHERE` clause). For range- and list-partitioned tables, trying to phrase a partition-level operation with a predicate can be cumbersome, especially when the range partitioning key uses more than one column. For hash partitions and subpartitions, using a predicate is more difficult still, because these partitions and subpartitions are based on a system-defined hash function.

Partition-extended names let you use partitions as if they were tables. An advantage of this method, which is most useful for range-partitioned tables, is that you can build partition-level access control mechanisms by granting (or revoking) privileges on these views to (or from) other users or roles. To use a partition as a table, create a view by selecting data from a single partition, and then use the view as a table.

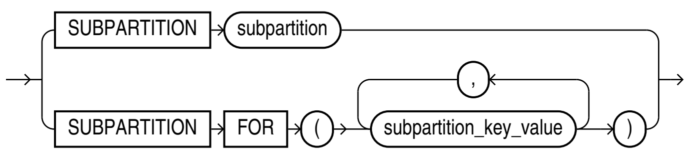
Syntax

You can specify partition-extended or subpartition-extended table names in any SQL statement in which the *partition_extended_name* or *subpartition_extended_name* element appears in the syntax.

***partition_extended_name*::=**

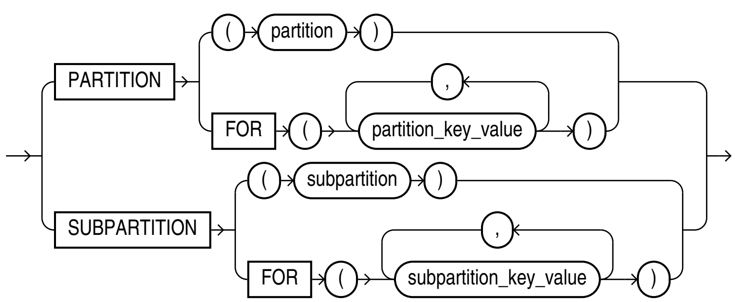


***subpartition_extended_name*::=**



The DML statements `INSERT`, `UPDATE`, and `DELETE` and the `ANALYZE` statement require parentheses around the partition or subpartition name. This small distinction is reflected in the *partition_extension_clause*:

***partition_extension_clause*::=**



In *partition_extended_name*, *subpartition_extended_name*, and *partition_extension_clause*, the `PARTITION FOR` and `SUBPARTITION FOR` clauses let you refer to a partition without using its name. They are valid with any type of partitioning and are especially useful for interval partitions. Interval partitions are created automatically as needed when data is inserted into a table.

For the respective *partition_key_value* or *subpartition_key_value*, specify one value for each partitioning key column. For multicolumn partitioning keys, specify one value for each partitioning key. For composite partitions, specify one value for each partitioning key, followed by one value for each subpartitioning key. All partitioning key values are comma separated. For interval partitions, you can specify only one *partition_key_value*, and it must be a valid NUMBER or datetime value. Your SQL statement will operate on the partition or subpartitions that contain the values you specify.

> **See Also:**
>
> The CREATE TABLE INTERVAL Clause for more information on interval partitions

Restrictions on Extended Names

Currently, the use of partition-extended and subpartition-extended table names has the following restrictions:

- No remote tables: A partition-extended or subpartition-extended table name cannot contain a database link (dblink) or a synonym that translates to a table with a dblink. To use remote partitions and subpartitions, create a view at the remote site that uses the extended table name syntax and then refer to the remote view.
- No synonyms: A partition or subpartition extension must be specified with a base table. You cannot use synonyms, views, or any other objects.
- The PARTITION FOR and SUBPARTITION FOR clauses are not valid for DDL operations on views.
- In the PARTITION FOR and SUBPARTITION FOR clauses, you cannot specify the keywords DEFAULT or MAXVALUE or a bind variable for the *partition_key_value* or *subpartition_key_value*.
- In the PARTITION and SUBPARTITION clauses, you cannot specify a bind variable for the partition or subpartition name.

Example

In the following statement, sales is a partitioned table with partition sales_q1_2000. You can create a view of the single partition sales_q1_2000, and then use it as if it were a table. This example deletes rows from the partition.

```
CREATE VIEW Q1_2000_sales AS
  SELECT *
    FROM sales PARTITION (SALES_Q1_2000);

DELETE FROM Q1_2000_sales
  WHERE amount_sold < 0;
```

References to Object Type Attributes and Methods

To refer to object type attributes or methods in a SQL statement, you must fully qualify the reference with a table alias. Consider the following example from the sample schema oe, which contains a type cust_address_typ and a table customers with a cust_address column based on the cust_address_typ:

```
CREATE TYPE cust_address_typ
  OID '82A4AF6A4CD1656DE034080020E0EE3D'
  AS OBJECT
```

```
    (street_address    VARCHAR2(40),
     postal_code       VARCHAR2(10),
     city              VARCHAR2(30),
     state_province    VARCHAR2(10),
     country_id        CHAR(2));
/
CREATE TABLE customers
  (customer_id         NUMBER(6),
   cust_first_name     VARCHAR2(20) CONSTRAINT cust_fname_nn NOT NULL,
   cust_last_name      VARCHAR2(20) CONSTRAINT cust_lname_nn NOT NULL,
   cust_address        cust_address_typ,
. . .
```

In a SQL statement, reference to the `postal_code` attribute must be fully qualified using a table alias, as illustrated in the following example:

```
SELECT c.cust_address.postal_code
  FROM customers c;

UPDATE customers c
  SET c.cust_address.postal_code = '14621-2604'
  WHERE c.cust_address.city = 'Rochester'
    AND c.cust_address.state_province = 'NY';
```

To reference a member method that does not accept arguments, you must provide empty parentheses. For example, the sample schema `oe` contains an object table `categories_tab`, based on `catalog_typ`, which contains the member function `getCatalogName`. In order to call this method in a SQL statement, you must provide empty parentheses as shown in this example:

```
SELECT TREAT(VALUE(c) AS catalog_typ).getCatalogName() "Catalog Type"
  FROM categories_tab c
  WHERE category_id = 90;

Catalog Type
------------------------------------
online catalog
```

3 Pseudocolumns

A **pseudocolumn** behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. A pseudocolumn is also similar to a function without arguments (refer to Functions). However, functions without arguments typically return the same value for every row in the result set, whereas pseudocolumns typically return a different value for each row.

This chapter contains the following sections:

- Hierarchical Query Pseudocolumns
- Sequence Pseudocolumns
- Version Query Pseudocolumns
- COLUMN_VALUE Pseudocolumn
- OBJECT_ID Pseudocolumn
- OBJECT_VALUE Pseudocolumn
- ORA_ROWSCN Pseudocolumn
- ROWID Pseudocolumn
- ROWNUM Pseudocolumn
- XMLDATA Pseudocolumn

Hierarchical Query Pseudocolumns

The hierarchical query pseudocolumns are valid only in hierarchical queries. The hierarchical query pseudocolumns are:

- CONNECT_BY_ISCYCLE Pseudocolumn
- CONNECT_BY_ISLEAF Pseudocolumn
- LEVEL Pseudocolumn

To define a hierarchical relationship in a query, you must use the `CONNECT BY` clause.

CONNECT_BY_ISCYCLE Pseudocolumn

The `CONNECT_BY_ISCYCLE` pseudocolumn returns 1 if the current row has a child which is also its ancestor. Otherwise it returns 0.

You can specify `CONNECT_BY_ISCYCLE` only if you have specified the `NOCYCLE` parameter of the `CONNECT BY` clause. `NOCYCLE` enables Oracle to return the results of a query that would otherwise fail because of a `CONNECT BY` loop in the data.

See Also:

Hierarchical Queries for more information about the NOCYCLE parameter and Hierarchical Query Examples for an example that uses the CONNECT_BY_ISCYCLE pseudocolumn

CONNECT_BY_ISLEAF Pseudocolumn

The CONNECT_BY_ISLEAF pseudocolumn returns 1 if the current row is a leaf of the tree defined by the CONNECT BY condition. Otherwise it returns 0. This information indicates whether a given row can be further expanded to show more of the hierarchy.

CONNECT_BY_ISLEAF Example

The following example shows the first three levels of the hr.employees table, indicating for each row whether it is a leaf row (indicated by 1 in the IsLeaf column) or whether it has child rows (indicated by 0 in the IsLeaf column):

```
SELECT last_name "Employee", CONNECT_BY_ISLEAF "IsLeaf",
       LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
  FROM employees
  WHERE LEVEL <= 3 AND department_id = 80
  START WITH employee_id = 100
  CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 4
  ORDER BY "Employee", "IsLeaf";

Employee                      IsLeaf      LEVEL Path
------------------------- ---------- ---------- -------------------------
Abel                               1          3 /King/Zlotkey/Abel
Ande                               1          3 /King/Errazuriz/Ande
Banda                              1          3 /King/Errazuriz/Banda
Bates                              1          3 /King/Cambrault/Bates
Bernstein                          1          3 /King/Russell/Bernstein
Bloom                              1          3 /King/Cambrault/Bloom
Cambrault                          0          2 /King/Cambrault
Cambrault                          1          3 /King/Russell/Cambrault
Doran                              1          3 /King/Partners/Doran
Errazuriz                          0          2 /King/Errazuriz
Fox                                1          3 /King/Cambrault/Fox
. . .
```

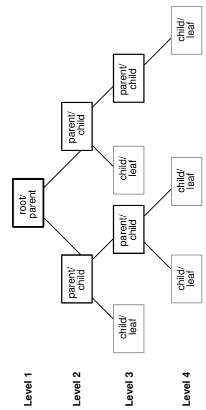
See Also:

Hierarchical Queries and SYS_CONNECT_BY_PATH

LEVEL Pseudocolumn

For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on. A **root row** is the highest row within an inverted tree. A **child row** is any nonroot row. A **parent row** is any row that has children. A **leaf row** is any row without children. Figure 3-1 shows the nodes of an inverted tree with their `LEVEL` values.

Figure 3-1   Hierarchical Tree



> **See Also:**
>
> Hierarchical Queries for information on hierarchical queries in general and IN Condition for restrictions on using the `LEVEL` pseudocolumn

Sequence Pseudocolumns

A **sequence** is a schema object that can generate unique sequential values. These values are often used for primary and unique keys. You can refer to sequence values in SQL statements with these pseudocolumns:

- `CURRVAL`: Returns the current value of a sequence
- `NEXTVAL`: Increments the sequence and returns the next value

You must qualify `CURRVAL` and `NEXTVAL` with the name of the sequence:

```
sequence.CURRVAL
sequence.NEXTVAL
```

To refer to the current or next value of a sequence in the schema of another user, you must have been granted either `SELECT` object privilege on the sequence or `SELECT ANY SEQUENCE` system privilege, and you must qualify the sequence with the schema containing it:

```
schema.sequence.CURRVAL
schema.sequence.NEXTVAL
```

To refer to the value of a sequence on a remote database, you must qualify the sequence with a complete or partial name of a database link:

```
schema.sequence.CURRVAL@dblink
schema.sequence.NEXTVAL@dblink
```

A sequence can be accessed by many users concurrently with no waiting or locking.

> **See Also:**
>
> References to Objects in Remote Databases for more information on referring to database links

Where to Use Sequence Values

You can use CURRVAL and NEXTVAL in the following locations:

- The select list of a SELECT statement that is not contained in a subquery, materialized view, or view
- The select list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

Restrictions on Sequence Values

You cannot use CURRVAL and NEXTVAL in the following constructs:

- A subquery in a DELETE, SELECT, or UPDATE statement
- A query of a view or of a materialized view
- A SELECT statement with the DISTINCT operator
- A SELECT statement with a GROUP BY clause or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- The condition of a CHECK constraint

Within a single SQL statement that uses CURRVAL or NEXTVAL, all referenced LONG columns, updated tables, and locked tables must be located on the same database.

How to Use Sequence Values

When you create a sequence, you can define its initial value and the increment between its values. The first reference to NEXTVAL returns the initial value of the sequence. Subsequent references to NEXTVAL increment the sequence value by the defined increment and return the new value. Any reference to CURRVAL always returns the current value of the sequence, which is the value returned by the last reference to NEXTVAL.

Before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL. Refer to CREATE SEQUENCE for information on sequences.

Within a single SQL statement containing a reference to NEXTVAL, Oracle increments the sequence once:

- For each row returned by the outer query block of a SELECT statement. Such a query block can appear in the following places:

  - A top-level `SELECT` statement
  - An `INSERT` ... `SELECT` statement (either single-table or multitable). For a multitable insert, the reference to `NEXTVAL` must appear in the `VALUES` clause, and the sequence is updated once for each row returned by the subquery, even though `NEXTVAL` may be referenced in multiple branches of the multitable insert.
  - A `CREATE TABLE` ... `AS SELECT` statement
  - A `CREATE MATERIALIZED VIEW` ... `AS SELECT` statement
- For each row updated in an `UPDATE` statement
- For each `INSERT` statement containing a `VALUES` clause
- For each `INSERT` ... [`ALL` | `FIRST`] statement (multitable insert). A multitable insert is considered a single SQL statement. Therefore, a reference to the `NEXTVAL` of a sequence will increase the sequence only once for each input record coming from the `SELECT` portion of the statement. If `NEXTVAL` is specified more than once in any part of the `INSERT` ... [`ALL` | `FIRST`] statement, then the value will be the same for all insert branches, regardless of how often a given record might be inserted.
- For each row merged by a `MERGE` statement. The reference to `NEXTVAL` can appear in the *`merge_insert_clause`* or the *`merge_update_clause`* or both. The `NEXTVALUE` value is incremented for each row updated and for each row inserted, even if the sequence number is not actually used in the update or insert operation. If `NEXTVAL` is specified more than once in any of these locations, then the sequence is incremented once for each row and returns the same value for all occurrences of `NEXTVAL` for that row.
- For each input row in a multitable `INSERT ALL` statement. `NEXTVAL` is incremented once for each row returned by the subquery, regardless of how many occurrences of the *`insert_into_clause`* map to each row.

If any of these locations contains more than one reference to `NEXTVAL`, then Oracle increments the sequence once and returns the same value for all occurrences of `NEXTVAL`.

If any of these locations contains references to both `CURRVAL` and `NEXTVAL`, then Oracle increments the sequence and returns the same value for both `CURRVAL` and `NEXTVAL`.

Finding the next value of a sequence: Example

This example selects the next value of the employee sequence in the sample schema `hr`:

```
SELECT employees_seq.nextval
  FROM DUAL;
```

Inserting sequence values into a table: Example

This example increments the employee sequence and uses its value for a new employee inserted into the sample table `hr.employees`:

```
INSERT INTO employees
  VALUES (employees_seq.nextval, 'John', 'Doe', 'jdoe', '555-1212',
          TO_DATE(SYSDATE), 'PU_CLERK', 2500, null, null, 30);
```

Reusing the current value of a sequence: Example

This example adds a new order with the next order number to the master order table. It then adds suborders with this number to the detail order table:

```
INSERT INTO orders (order_id, order_date, customer_id)
  VALUES (orders_seq.nextval, TO_DATE(SYSDATE), 106);

INSERT INTO order_items (order_id, line_item_id, product_id)
  VALUES (orders_seq.currval, 1, 2359);

INSERT INTO order_items (order_id, line_item_id, product_id)
  VALUES (orders_seq.currval, 2, 3290);

INSERT INTO order_items (order_id, line_item_id, product_id)
  VALUES (orders_seq.currval, 3, 2381);
```

Version Query Pseudocolumns

The version query pseudocolumns are valid only in Oracle Flashback Version Query, which is a form of Oracle Flashback Query. The version query pseudocolumns are:

- `VERSIONS_STARTSCN` and `VERSIONS_STARTTIME`: Starting System Change Number (SCN) or `TIMESTAMP` when the row version was created. This pseudocolumn identifies the time when the data first had the values reflected in the row version. Use this pseudocolumn to identify the past target time for Oracle Flashback Table or Oracle Flashback Query. If this pseudocolumn is `NULL`, then the row version was created before start.
- `VERSIONS_ENDSCN` and `VERSIONS_ENDTIME`: SCN or `TIMESTAMP` when the row version expired. If the pseudocolumn is `NULL`, then either the row version was current at the time of the query or the row corresponds to a `DELETE` operation.
- `VERSIONS_XID`: Identifier (a `RAW` number) of the transaction that created the row version.
- `VERSIONS_OPERATION`: Operation performed by the transaction: `I` for insertion, `D` for deletion, or `U` for update. The version is that of the row that was inserted, deleted, or updated; that is, the row after an `INSERT` operation, the row before a `DELETE` operation, or the row affected by an `UPDATE` operation.

  For user updates of an index key, Oracle Flashback Version Query might treat an `UPDATE` operation as two operations, `DELETE` plus `INSERT`, represented as two version rows with a `D` followed by an `I` `VERSIONS_OPERATION`.

See Also:

- *flashback_query_clause* for more information on version queries
- *Oracle Database Development Guide* for more information on using Oracle Flashback Version Query
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for values of the `VERSIONS_OPERATION` pseudocolumn

COLUMN_VALUE Pseudocolumn

When you refer to an XMLTable construct without the COLUMNS clause, or when you use the TABLE collection expression to refer to a scalar nested table type, the database returns a virtual table with a single column. This name of this pseudocolumn is COLUMN_VALUE.

In the context of XMLTable, the value returned is of data type XMLType. For example, the following two statements are equivalent, and the output for both shows COLUMN_VALUE as the name of the column being returned:

```
SELECT *
  FROM XMLTABLE('<a>123</a>');

COLUMN_VALUE
---------------------------------------
<a>123</a>

SELECT COLUMN_VALUE
  FROM (XMLTable('<a>123</a>'));

COLUMN_VALUE
----------------------------------------
<a>123</a>
```

In the context of a TABLE collection expression, the value returned is the data type of the collection element. The following statements create the two levels of nested tables illustrated in Creating a Table: Multilevel Collection Example to show the uses of COLUMN_VALUE in this context:

```
CREATE TYPE phone AS TABLE OF NUMBER;
/
CREATE TYPE phone_list AS TABLE OF phone;
/
```

The next statement uses COLUMN_VALUE to select from the phone type:

```
SELECT t.COLUMN_VALUE
  FROM TABLE(phone(1,2,3)) t;

COLUMN_VALUE
------------
           1
           2
           3
```

In a nested type, you can use the COLUMN_VALUE pseudocolumn in both the select list and the TABLE collection expression:

```
SELECT t.COLUMN_VALUE
  FROM TABLE(phone_list(phone(1,2,3))) p, TABLE(p.COLUMN_VALUE) t;

COLUMN_VALUE
------------
           1
           2
           3
```

The keyword COLUMN_VALUE is also the name that Oracle Database generates for the scalar value of an inner nested table without a column or attribute name, as shown in the example that follows. In this context, COLUMN_VALUE is not a pseudocolumn, but an actual column name.

```
CREATE TABLE my_customers (
    cust_id       NUMBER,
    name          VARCHAR2(25),
    phone_numbers phone_list,
    credit_limit  NUMBER)
  NESTED TABLE phone_numbers STORE AS outer_ntab
  (NESTED TABLE COLUMN_VALUE STORE AS inner_ntab);
```

> **See Also:**
>
> - XMLTABLE for information on that function
> - *table_collection_expression::=* for information on the TABLE collection expression
> - ALTER TABLE examples in Nested Tables: Examples
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for values of the COLUMN_VALUE pseudocolumn

OBJECT_ID Pseudocolumn

The OBJECT_ID pseudocolumn returns the object identifier of a column of an object table or view. Oracle uses this pseudocolumn as the primary key of an object table. OBJECT_ID is useful in INSTEAD OF triggers on views and for identifying the ID of a substitutable row in an object table.

> **Note:**
>
> In earlier releases, this pseudocolumn was called SYS_NC_OID$. That name is still supported for backward compatibility. However, Oracle recommends that you use the more intuitive name OBJECT_ID.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for examples of the use of this pseudocolumn

OBJECT_VALUE Pseudocolumn

The OBJECT_VALUE pseudocolumn returns system-generated names for the columns of an object table, XMLType table, object view, or XMLType view. This pseudocolumn is useful for identifying the value of a substitutable row in an object table and for creating object views with the WITH OBJECT IDENTIFIER clause.

> **Note:**
>
> In earlier releases, this pseudocolumn was called SYS_NC_ROWINFO$. That name is still supported for backward compatibility. However, Oracle recommends that you use the more intuitive name OBJECT_VALUE.

> **See Also:**
>
> - *object_table* and *object_view_clause* for more information on the use of this pseudocolumn
> - *Oracle Database Object-Relational Developer's Guide* for examples of the use of this pseudocolumn

ORA_ROWSCN Pseudocolumn

ORA_ROWSCN reflects the system change-number (SCN) of the most recent change to a row. This change can be at the level of a block (coarse) or at the level of a row (fine-grained). The latter is provided by row-level dependency tracking. Refer to CREATE TABLE ... NOROWDEPENDENCIES | ROWDEPENDENCIES for more information on row-level dependency tracking. In the absence of row-level dependencies, ORA_ROWSCN reflects block-level dependencies.

Whether at the block level or at the row level, the ORA_ROWSCN should not be considered to be an exact SCN. For example, if a transaction changed row R in a block and committed at SCN 10, it is not always true that the ORA_ROWSCN for the row would return 10. While a value less than 10 would never be returned, any value greater than or equal to 10 could be returned. That is, the ORA_ROWSCN of a row is not always guaranteed to be the exact commit SCN of the transaction that last modified that row. However, with fine-grained ORA_ROWSCN, if two transactions T1 and T2 modified the same row R, one after another, and committed, a query on the ORA_ROWSCN of row R after the commit of T1 will return a value lower than the value returned after the commit of T2. If a block is queried twice, then it is possible for the value of ORA_ROWSCN to change between the queries even though rows have not been updated in the time between the queries. The only guarantee is that the value of ORA_ROWSCN in both queries is greater than the commit SCN of the transaction that last modified that row.

You cannot use the ORA_ROWSCN pseudocolumn in a query to a view. However, you can use it to refer to the underlying table when creating a view. You can also use this pseudocolumn in the WHERE clause of an UPDATE or DELETE statement.

ORA_ROWSCN is not supported for Flashback Query. Instead, use the version query pseudocolumns, which are provided explicitly for Flashback Query. Refer to the SELECT ... *flashback_query_clause* for information on Flashback Query and Version Query Pseudocolumns for additional information on those pseudocolumns.

Restriction on ORA_ROWSCN: This pseudocolumn is not supported for external tables.

Example

The first statement below uses the ORA_ROWSCN pseudocolumn to get the system change number of the last operation on the employees table. The second statement uses the pseudocolumn with the SCN_TO_TIMESTAMP function to determine the timestamp of the operation:

```
SELECT ORA_ROWSCN, last_name
  FROM employees
  WHERE employee_id = 188;

SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN), last_name
  FROM employees
  WHERE employee_id = 188;
```

> **See Also:**
>
> SCN_TO_TIMESTAMP

ROWID Pseudocolumn

For each row in the database, the ROWID pseudocolumn returns the address of the row. Oracle Database rowid values contain information necessary to locate a row:

- The data object number of the object
- The data block in the data file in which the row resides
- The position of the row in the data block (first row is 0)
- The data file in which the row resides (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.

Values of the ROWID pseudocolumn have the data type ROWID or UROWID. Refer to Rowid Data Types and UROWID Data Type for more information.

Rowid values have several important uses:

- They are the fastest way to access a single row.
- They can show you how the rows in a table are stored.
- They are unique identifiers for rows in a table.

You should not use ROWID as the primary key of a table. If you delete and reinsert a row with the Import and Export utilities, for example, then its rowid may change. If you delete a row, then Oracle may reassign its rowid to a new row inserted later.

Although you can use the ROWID pseudocolumn in the SELECT and WHERE clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the ROWID pseudocolumn.

Example

This statement selects the address of all rows that contain data for employees in department 20:

```
SELECT ROWID, last_name
  FROM employees
  WHERE department_id = 20;
```

ROWNUM Pseudocolumn

> **Note:**
>
> - The `ROW_NUMBER` built-in SQL function provides superior support for ordering the results of a query. Refer to ROW_NUMBER for more information.
> - The *`row_limiting_clause`* of the `SELECT` statement provides superior support for limiting the number of rows returned by a query. Refer to *row_limiting_clause* for more information.

For each row returned by a query, the `ROWNUM` pseudocolumn returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. The first row selected has a `ROWNUM` of 1, the second has 2, and so on.

You can use `ROWNUM` to limit the number of rows returned by a query, as in this example:

```
SELECT *
  FROM employees
  WHERE ROWNUM < 11;
```

If an `ORDER BY` clause follows `ROWNUM` in the same query, then the rows will be reordered by the `ORDER BY` clause. The results can vary depending on the way the rows are accessed. For example, if the `ORDER BY` clause causes Oracle to use an index to access the data, then Oracle may retrieve the rows in a different order than without the index. Therefore, the following statement does not necessarily return the same rows as the preceding example:

```
SELECT *
  FROM employees
  WHERE ROWNUM < 11
  ORDER BY last_name;
```

If you embed the `ORDER BY` clause in a subquery and place the `ROWNUM` condition in the top-level query, then you can force the `ROWNUM` condition to be applied after the ordering of the rows. For example, the following query returns the employees with the 10 smallest employee numbers. This is sometimes referred to as **top-N reporting**:

```
SELECT *
  FROM (SELECT * FROM employees ORDER BY employee_id)
  WHERE ROWNUM < 11;
```

In the preceding example, the `ROWNUM` values are those of the top-level `SELECT` statement, so they are generated after the rows have already been ordered by `employee_id` in the subquery.

Conditions testing for ROWNUM values greater than a positive integer are always false. For example, this query returns no rows:

```
SELECT *
  FROM employees
  WHERE ROWNUM > 1;
```

The first row fetched is assigned a ROWNUM of 1 and makes the condition false. The second row to be fetched is now the first row and is also assigned a ROWNUM of 1 and makes the condition false. All rows subsequently fail to satisfy the condition, so no rows are returned.

You can also use ROWNUM to assign unique values to each row of a table, as in this example:

```
UPDATE my_table
  SET column1 = ROWNUM;
```

Refer to the function ROW_NUMBER for an alternative method of assigning unique numbers to rows.

> **Note:**
>
> Using ROWNUM in a query can affect view optimization.

XMLDATA Pseudocolumn

Oracle stores XMLType data either in LOB or object-relational columns, based on XMLSchema information and how you specify the storage clause. The XMLDATA pseudocolumn lets you access the underlying LOB or object relational column to specify additional storage clause parameters, constraints, indexes, and so forth.

Example

The following statements illustrate the use of this pseudocolumn. Suppose you create a simple table of XMLType with one CLOB column:

```
CREATE TABLE xml_lob_tab of XMLTYPE
  XMLTYPE STORE AS CLOB;
```

To change the storage characteristics of the underlying LOB column, you can use the following statement:

```
ALTER TABLE xml_lob_tab
  MODIFY LOB (XMLDATA) (STORAGE (MAXSIZE 2G) CACHE);
```

Now suppose you have created an XMLSchema-based table like the xwarehouses table created in Using XML in SQL Statements . You could then use the XMLDATA column to set the properties of the underlying columns, as shown in the following statement:

```
ALTER TABLE xwarehouses
  ADD (UNIQUE(XMLDATA."WarehouseId"));
```

4 Operators

An **operator** manipulates data items and returns a result. Syntactically, an operator appears before or after an operand or between two operands.

This chapter contains these sections:

- About SQL Operators
- Arithmetic Operators
- COLLATE Operator
- Concatenation Operator
- Hierarchical Query Operators
- Set Operators
- Multiset Operators
- User-Defined Operators

This chapter discusses nonlogical (non-Boolean) operators. These operators cannot by themselves serve as the condition of a `WHERE` or `HAVING` clause in queries or subqueries. For information on logical operators, which serve as conditions, refer to Conditions.

About SQL Operators

Operators manipulate individual data items called **operands** or **arguments**. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*).

If you have installed Oracle Text, then you can use the `SCORE` operator, which is part of that product, in Oracle Text queries. You can also create conditions with the built-in Text operators, including `CONTAINS`, `CATSEARCH`, and `MATCHES`. For more information on these Oracle Text elements, refer to *Oracle Text Reference*.

Unary and Binary Operators

The two general classes of operators are:

- **unary**: A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

  ```
  operator operand
  ```

- **binary**: A binary operator operates on two operands. A binary operator appears with its operands in this format:

  ```
  operand1 operator operand2
  ```

Other operators with special formats accept more than two operands. If an operator is given a null operand, then the result is always null. The only operator that does not follow this rule is concatenation (||).

Operator Precedence

Precedence is the order in which Oracle Database evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

Table 4-1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 4-1 SQL Operator Precedence

Operator	Operation
`+`, `-` (as unary operators), `PRIOR`, `CONNECT_BY_ROOT`, `COLLATE`	Identity, negation, location in hierarchy
`*`, `/`	Multiplication, division
`+`, `-` (as binary operators), `||`	Addition, subtraction, concatenation
SQL conditions are evaluated after SQL operators	See "Condition Precedence"

Precedence Example

In the following expression, multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

```
1+2*3
```

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

SQL also supports set operators (`UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`), which combine sets of rows returned by queries, rather than individual data items. All set operators have equal precedence.

> **See Also:**
>
> Hierarchical Query Operators and Hierarchical Queries for information on the `PRIOR` operator, which is used only in hierarchical queries

Arithmetic Operators

You can use an arithmetic operator with one or two arguments to negate, add, subtract, multiply, and divide numeric values. Some of these operators are also used in datetime and interval arithmetic. The arguments to the operator must resolve to numeric data types or to any data type that can be implicitly converted to a numeric data type.

Unary arithmetic operators return the same data type as the numeric data type of the argument. For binary arithmetic operators, Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type. Table 4-2 lists arithmetic operators.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion, Numeric Precedence for information on numeric precedence, and Datetime/Interval Arithmetic

Table 4-2 Arithmetic Operators

Operator	Purpose	Example

+ -	When these denote a positive or negative expression, they are unary operators.	`SELECT * FROM order_items WHERE quantity = -1 ORDER BY order_id, line_item_id, product_id;`
		
		`SELECT * FROM employees WHERE -salary < 0 ORDER BY employee_id;`

+ -	When they add or subtract, they are binary operators.	`SELECT hire_date FROM employees WHERE SYSDATE - hire_date > 365 ORDER BY hire_date;`
* /	Multiply, divide. These are binary operators.	`UPDATE employees SET salary = salary * 1.1;`

Do not use two consecutive minus signs (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. The characters -- are used to begin comments within SQL statements. You should separate consecutive minus signs with a space or parentheses. Refer to Comments for more information on comments within SQL statements.

COLLATE Operator

The `COLLATE` operator determines the collation for an expression. This operator enables you to override the collation that the database would have derived for the expression using standard collation derivation rules.

`COLLATE` is a postfix unary operator. It has the same precedence as other unary operators, but it is evaluated after all prefix unary operators have been evaluated.

You can apply this operator to expressions of type `VARCHAR2`, `CHAR`, `LONG`, `NVARCHAR`, or `NCHAR`.

The COLLATE operator takes one argument, *collation_name*, for which you can specify a named collation or pseudo-collation. If the collation name contains a space, then you must enclose the name in double quotation marks.

Table 4-3 describes the COLLATE operator.

Table 4-3 COLLATE Operator

Operator	Purpose	Example
COLLATE *collation_name*	Determines the collation for an expression	`SELECT last_name FROM employees ORDER BY last_name COLLATE GENERIC_M;`

See Also:

- Compound Expressions for information on using the COLLATE operator in a compound expression
- *Oracle Database Globalization Support Guide* for more information on the COLLATE operator

Concatenation Operator

The concatenation operator manipulates character strings and CLOB data. Table 4-4 describes the concatenation operator.

Table 4-4 Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings and CLOB data.	`SELECT 'Name is ' || last_name FROM employees ORDER BY last_name;`

The result of concatenating two character strings is another character string. If both character strings are of data type CHAR, then the result has data type CHAR and is limited to 2000 characters. If either string is of data type VARCHAR2, then the result has data type VARCHAR2 and is limited to 32767 characters if the initialization parameter MAX_STRING_SIZE = EXTENDED and 4000 characters if MAX_STRING_SIZE = STANDARD. Refer to Extended Data Types for more information. If either argument is a CLOB, the result is a temporary CLOB. Trailing blanks in character strings are preserved by concatenation, regardless of the data types of the string or CLOB.

On most platforms, the concatenation operator is two solid vertical bars, as shown in Table 4-4. However, some IBM platforms use broken vertical bars for this operator. When moving SQL script files between systems having different character sets, such as between ASCII and EBCDIC, vertical bars might not be translated into the vertical bar required by the target Oracle Database environment. Oracle provides the CONCAT character function as an alternative to the vertical bar operator for cases when it is difficult or impossible to control translation performed by operating system or network utilities. Use this function in applications that will be moved between environments with differing character sets.

Although Oracle treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle Database. To concatenate an expression that might be null, use the `NVL` function to explicitly convert the expression to a zero-length string.

> **See Also:**
>
> - Character Data Types for more information on the differences between the `CHAR` and `VARCHAR2` data types
> - The functions CONCAT and NVL
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about `CLOB`s
> - *Oracle Database Globalization Support Guide* for the collation derivation rules for the concatenation operator

Concatenation Example

This example creates a table with both `CHAR` and `VARCHAR2` columns, inserts values both with and without trailing blanks, and then selects these values and concatenates them. Note that for both `CHAR` and `VARCHAR2` columns, the trailing blanks are preserved.

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(6),
                   col3 VARCHAR2(6), col4 CHAR(6));

INSERT INTO tab1 (col1,  col2,     col3,     col4)
          VALUES ('abc', 'def   ', 'ghi   ', 'jkl');

SELECT col1 || col2 || col3 || col4 "Concatenation"
  FROM tab1;

Concatenation
------------------------
abcdef   ghi   jkl
```

Hierarchical Query Operators

Two operators, `PRIOR` and `CONNECT_BY_ROOT`, are valid only in hierarchical queries.

PRIOR

In a hierarchical query, one expression in the `CONNECT BY` *`condition`* must be qualified by the `PRIOR` operator. If the `CONNECT BY` *`condition`* is compound, then only one condition requires the `PRIOR` operator, although you can have multiple `PRIOR` conditions. `PRIOR` evaluates the immediately following expression for the parent row of the current row in a hierarchical query.

PRIOR is most commonly used when comparing column values with the equality operator. (The PRIOR keyword can be on either side of the operator.) PRIOR causes Oracle to use the value of the parent row in the column. Operators other than the equal sign (=) are theoretically possible in CONNECT BY clauses. However, the conditions created by these other operators can result in an infinite loop through the possible combinations. In this case Oracle detects the loop at run time and returns an error. Refer to Hierarchical Queries for more information on this operator, including examples.

CONNECT_BY_ROOT

CONNECT_BY_ROOT is a unary operator that is valid only in hierarchical queries. When you qualify a column with this operator, Oracle returns the column value using data from the root row. This operator extends the functionality of the CONNECT BY [PRIOR] condition of hierarchical queries.

Restriction on CONNECT_BY_ROOT

You cannot specify this operator in the START WITH condition or the CONNECT BY condition.

> **See Also:**
>
> CONNECT_BY_ROOT Examples

Set Operators

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table 4-5 lists SQL set operators. They are fully described, including examples and restrictions on these operators, in The UNION [ALL], INTERSECT, MINUS Operators.

Table 4-5 Set Operators

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows selected by the first query but not the second

Multiset Operators

Multiset operators combine the results of two nested tables into a single nested table.

The examples related to multiset operators require that two nested tables be created and loaded with data as follows:

First, make a copy of the oe.customers table called customers_demo:

```
CREATE TABLE customers_demo AS
  SELECT * FROM customers;
```

Next, create a table type called `cust_address_tab_typ`. This type will be used when creating the nested table columns.

```
CREATE TYPE cust_address_tab_typ AS
  TABLE OF cust_address_typ;
/
```

Now, create two nested table columns in the `customers_demo` table:

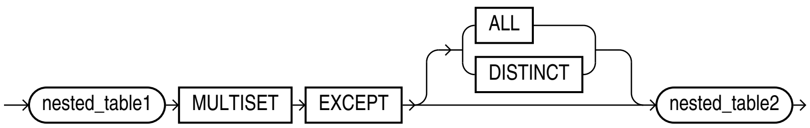
```
ALTER TABLE customers_demo
  ADD (cust_address_ntab cust_address_tab_typ,
       cust_address2_ntab cust_address_tab_typ)
    NESTED TABLE cust_address_ntab STORE AS cust_address_ntab_store
    NESTED TABLE cust_address2_ntab STORE AS cust_address2_ntab_store;
```

Finally, load data into the two new nested table columns using data from the `cust_address` column of the `oe.customers` table:

```
UPDATE customers_demo cd
  SET cust_address_ntab =
    CAST(MULTISET(SELECT cust_address
                    FROM customers c
                    WHERE c.customer_id =
                          cd.customer_id) as cust_address_tab_typ);

UPDATE customers_demo cd
  SET cust_address2_ntab =
    CAST(MULTISET(SELECT cust_address
                    FROM customers c
                    WHERE c.customer_id =
                          cd.customer_id) as cust_address_tab_typ);
```

MULTISET EXCEPT

`MULTISET EXCEPT` takes as arguments two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.



- The `ALL` keyword instructs Oracle to return all elements in *`nested_table1`* that are not in *`nested_table2`*. For example, if a particular element occurs *`m`* times in *`nested_table1`* and *`n`* times in *`nested_table2`*, then the result will have *`(m-n)`* occurrences of the element if *`m >n`* and 0 occurrences if *`m<=n`*. `ALL` is the default.
- The `DISTINCT` keyword instructs Oracle to eliminate any element in *`nested_table1`* which is also in *`nested_table2`*, regardless of the number of occurrences.
- The element types of the nested tables must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

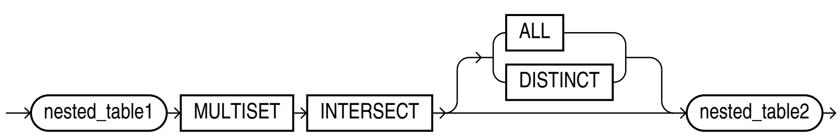
The following example compares two nested tables and returns a nested table of those elements found in the first nested table but not in the second nested table:

```
SELECT customer_id, cust_address_ntab
  MULTISET EXCEPT DISTINCT cust_address2_ntab multiset_except
  FROM customers_demo
  ORDER BY customer_id;

CUSTOMER_ID MULTISET_EXCEPT(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
----------- --------------------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP()
        102 CUST_ADDRESS_TAB_TYP()
        103 CUST_ADDRESS_TAB_TYP()
        104 CUST_ADDRESS_TAB_TYP()
        105 CUST_ADDRESS_TAB_TYP()
. . .
```

The preceding example requires the table `customers_demo` and two nested table columns containing data. Refer to Multiset Operators to create this table and nested table columns.

MULTISET INTERSECT

`MULTISET INTERSECT` takes as arguments two nested tables and returns a nested table whose values are common in the two input nested tables. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.



- The `ALL` keyword instructs Oracle to return all common occurrences of elements that are in the two input nested tables, including duplicate common values and duplicate common `NULL` occurrences. For example, if a particular value occurs `m` times in *`nested_table1`* and `n` times in *`nested_table2`*, then the result would contain the element `min(m,n)` times. `ALL` is the default.
- The `DISTINCT` keyword instructs Oracle to eliminate duplicates from the returned nested table, including duplicates of `NULL`, if they exist.
- The element types of the nested tables must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example compares two nested tables and returns a nested table of those elements found in both input nested tables:

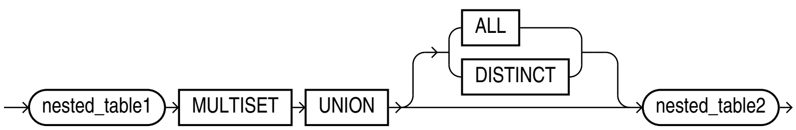
```
SELECT customer_id, cust_address_ntab
  MULTISET INTERSECT DISTINCT cust_address2_ntab multiset_intersect
  FROM customers_demo
  ORDER BY customer_id;

CUSTOMER_ID MULTISET_INTERSECT(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID
----------- ------------------------------------------------------------------------------------
```

```
        101 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'))
        102 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'))
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
        104 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'))
        105 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
. . .
```

The preceding example requires the table `customers_demo` and two nested table columns containing data. Refer to Multiset Operators to create this table and nested table columns.

MULTISET UNION

`MULTISET UNION` takes as arguments two nested tables and returns a nested table whose values are those of the two input nested tables. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.



- The `ALL` keyword instructs Oracle to return all elements that are in the two input nested tables, including duplicate values and duplicate `NULL` occurrences. This is the default.
- The `DISTINCT` keyword instructs Oracle to eliminate duplicates from the returned nested table, including duplicates of `NULL`, if they exist.
- The element types of the nested tables must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example compares two nested tables and returns a nested table of elements from both input nested tables:

```
SELECT customer_id, cust_address_ntab
  MULTISET UNION cust_address2_ntab multiset_union
  FROM customers_demo
  ORDER BY customer_id;

CUSTOMER_ID MULTISET_UNION(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
----------- -------------------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'),
                                 CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'))
        102 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'),
                                 CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN','US'))
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'),
                                 CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
        104 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'),
                                 CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'))
        105 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'),
                                 CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
. . .
```

The preceding example requires the table `customers_demo` and two nested table columns containing data. Refer to Multiset Operators to create this table and nested table columns.

User-Defined Operators

Like built-in operators, user-defined operators take a set of operands as input and return a result. However, you create them with the `CREATE OPERATOR` statement, and they are identified by user-defined names. They reside in the same namespace as tables, views, types, and standalone functions.

After you have defined a new operator, you can use it in SQL statements like any other built-in operator. For example, you can use user-defined operators in the select list of a `SELECT` statement, the condition of a `WHERE` clause, or in `ORDER BY` clauses and `GROUP BY` clauses. However, you must have `EXECUTE` privilege on the operator to do so, because it is a user-defined object.

> **See Also:**
>
> CREATE OPERATOR for an example of creating an operator and *Oracle Database Data Cartridge Developer's Guide* for more information on user-defined operators

5 Expressions

This chapter describes how to combine values, operators, and functions into **expressions**.

This chapter includes these sections:

- About SQL Expressions
- Simple Expressions
- Compound Expressions
- Calculated Measure Expressions
- CASE Expressions
- Column Expressions
- CURSOR Expressions
- Datetime Expressions
- Function Expressions
- Interval Expressions
- JSON Object Access Expressions
- Model Expressions
- Object Access Expressions
- Placeholder Expressions
- Scalar Subquery Expressions
- Type Constructor Expressions
- Expression Lists

About SQL Expressions

An **expression** is a combination of one or more values, operators, and SQL functions that evaluates to a value. An expression generally assumes the data type of its components.

This simple expression evaluates to 4 and has data type `NUMBER` (the same data type as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to `CHAR` data type:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in:

- The select list of the `SELECT` statement

- A condition of the WHERE clause and HAVING clause
- The CONNECT BY, START WITH, and ORDER BY clauses
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

For example, you could use an expression in place of the quoted string 'Smith' in this UPDATE statement SET clause:

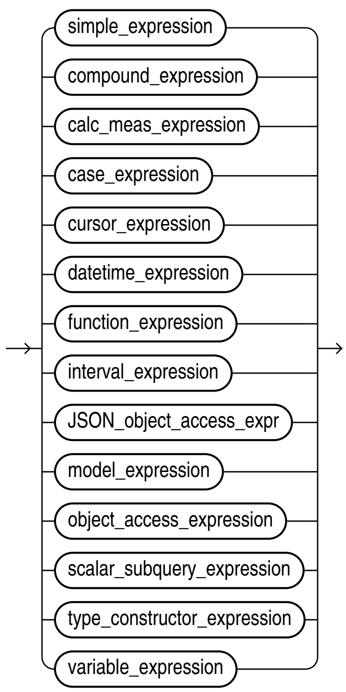
```
SET last_name = 'Smith';
```

This SET clause has the expression INITCAP(last_name) instead of the quoted string 'Smith':

```
SET last_name = INITCAP(last_name);
```

Expressions have several forms, as shown in the following syntax:

***expr*::=**



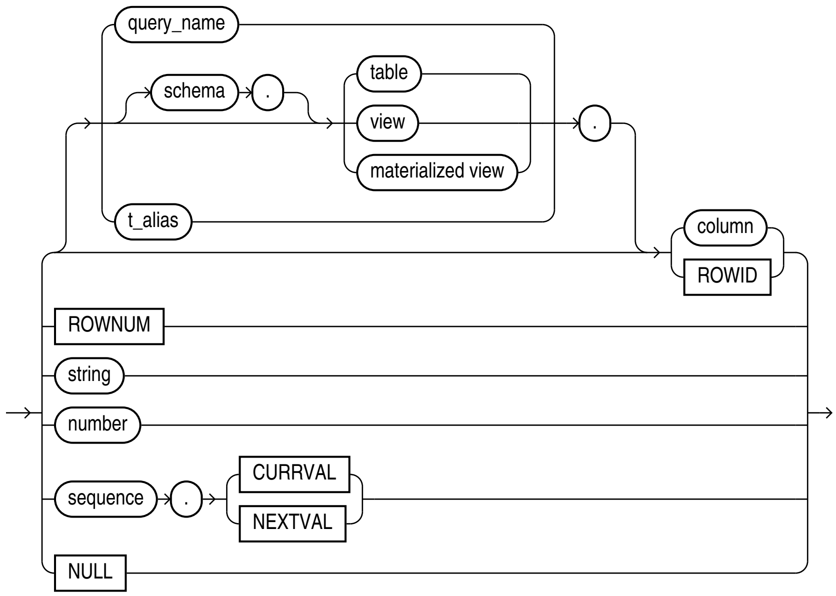
Oracle Database does not accept all forms of expressions in all parts of all SQL statements. Refer to the section devoted to a particular SQL statement in this book for information on restrictions on the expressions in that statement.

You must use appropriate expression notation whenever *expr* appears in conditions, SQL functions, or SQL statements in other parts of this reference. The sections that follow describe and provide examples of the various forms of expressions.

Simple Expressions

A simple expression specifies a column, pseudocolumn, constant, sequence number, or null.

***simple_expression*::=**



In addition to the schema of a user, `schema` can also be "`PUBLIC`" (double quotation marks required), in which case it must qualify a public synonym for a table, view, or materialized view. Qualifying a public synonym with "`PUBLIC`" is supported only in data manipulation language (DML) statements, not data definition language (DDL) statements.

You can specify `ROWID` only with a table, not with a view or materialized view. `NCHAR` and `NVARCHAR2` are not valid pseudocolumn data types.

See Also:

Pseudocolumns for more information on pseudocolumns and subquery_factoring_clause for information on `query_name`

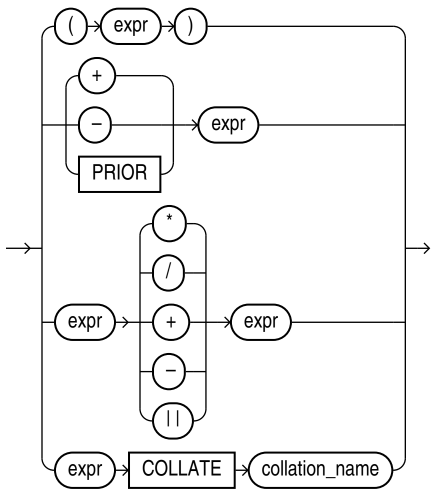
Some valid simple expressions are:

```
employees.last_name
'this is a text string'
10
N'this is an NCHAR string'
```

Compound Expressions

A compound expression specifies a combination of other expressions.

compound_expression::=



You can use any built-in function as an expression (Function Expressions). However, in a compound expression, some combinations of functions are inappropriate and are rejected. For example, the `LENGTH` function is inappropriate within an aggregate function.

The `PRIOR` operator is used in `CONNECT BY` clauses of hierarchical queries.

The `COLLATE` operator determines the collation for an expression. This operator overrides the collation that the database would have derived for the expression using standard collation derivation rules.

See Also:

- Operator Precedence
- Hierarchical Queries
- COLLATE Operator

Some valid compound expressions are:

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate,'DD-MMM-YY'))
name COLLATE BINARY_CI
```

Calculated Measure Expressions

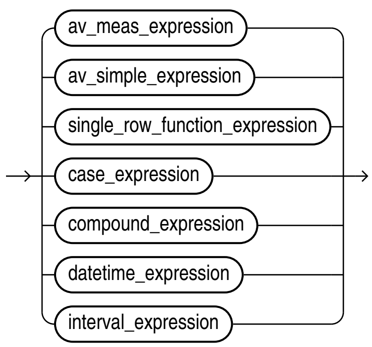
A calculated measure expression defines a calculated measure in an analytic view. You use a calculated measure expression as the *`calc_meas_expression`* parameter in a *`calc_measure_clause`* in a `CREATE ANALYTIC VIEW` statement.

Tip:

You can view and run SQL scripts that create analytic views with calculated measures at the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html. The website has scripts and tutorials that demonstrate the creation and use of analytic views.

Syntax

***calc_meas_expression*::=**



Semantics

calc_meas_expression

The calculated measure expressions that have syntax specific to analytic views are described in the following topics:

- Analytic View Measure Expressions
- Analytic View Simple Expressions
- Single Row Function Expression

For the other types of permissible expressions for a calculated measure, see the following topics:

- CASE Expressions
- Compound Expressions
- Datetime Expressions
- Interval Expressions

See Also:

CREATE ANALYTIC VIEW

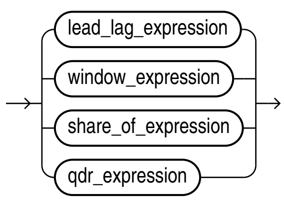
Analytic View Measure Expressions

An analytic view measure expression is based on a measure in an analytic view.

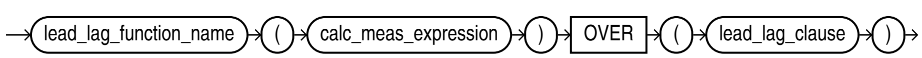
> **Tip:**
>
> You can view and run SQL scripts that create analytic views with calculated measures at the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html. The website has scripts and tutorials that demonstrate the creation and use of analytic views.

Syntax

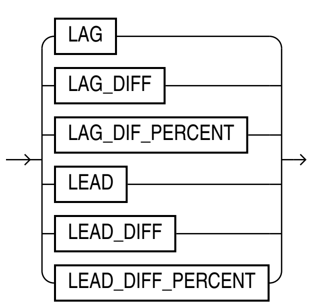
***av_meas_expression*::=**



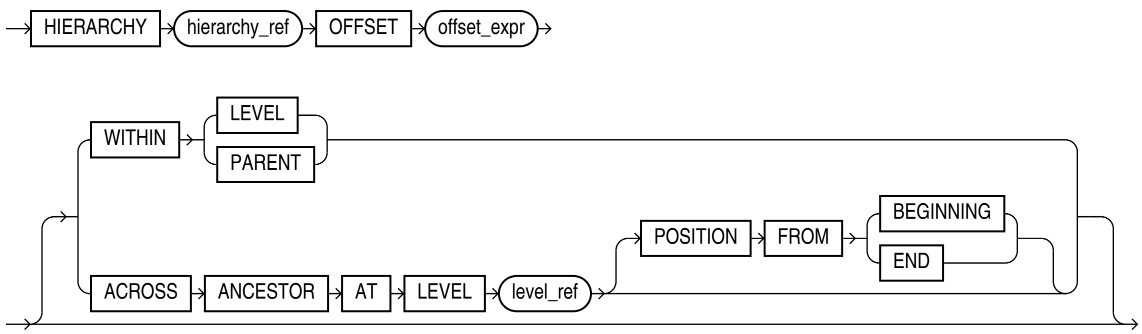
***lead_lag_expression*::=**



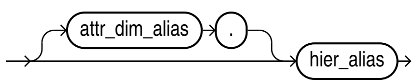
***lead_lag_function_name*::=**



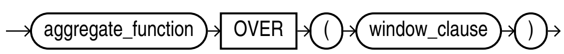
***lead_lag_clause*::=**



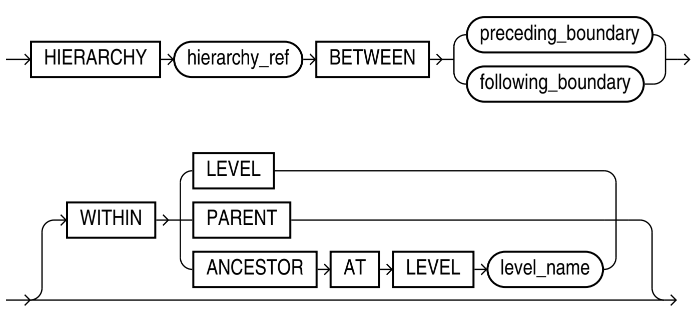
***hierarchy_ref*::=**



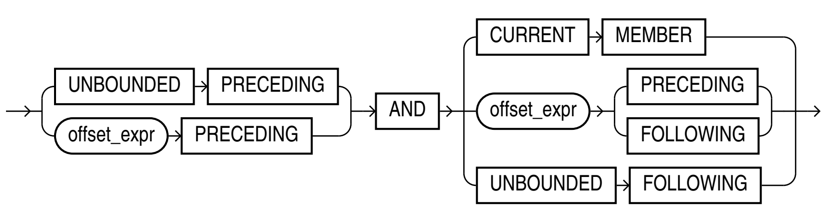
***window_expression*::=**



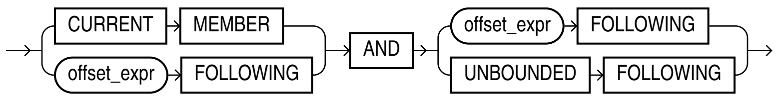
***window_clause*::=**



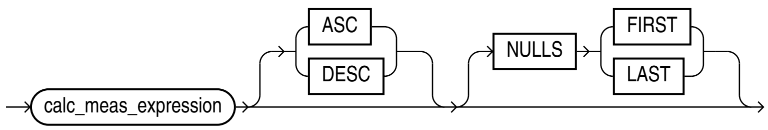
***preceding_boundary* ::=**



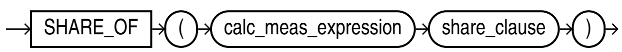
***following_boundary*::=**



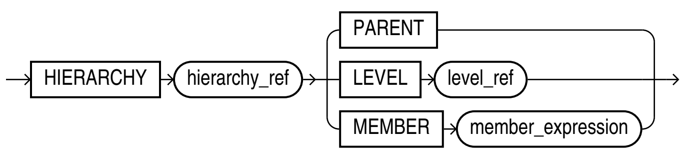
***calc_meas_order_by_clause*::=**



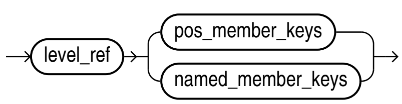
***share_of_expression*::=**



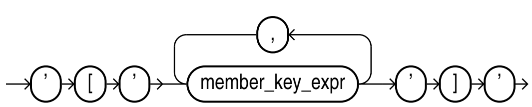
***share_clause*::=**



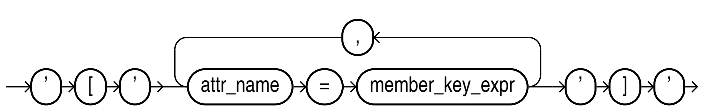
***level_member_literal*::=**



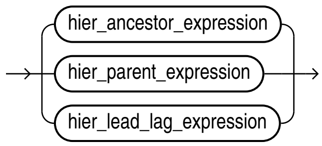
***pos_member_keys*::=**



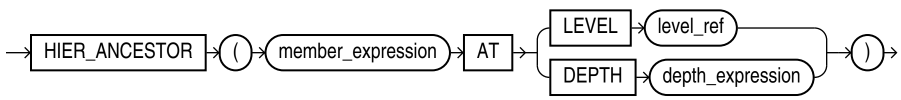
***named_member_keys*::=**



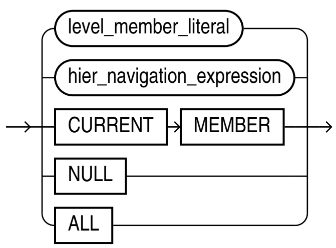
***hier_navigation_expression*::=**



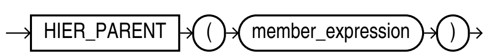
***hier_ancestor_expression*::=**



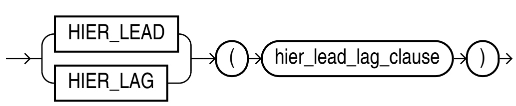
***member_expression*::=**



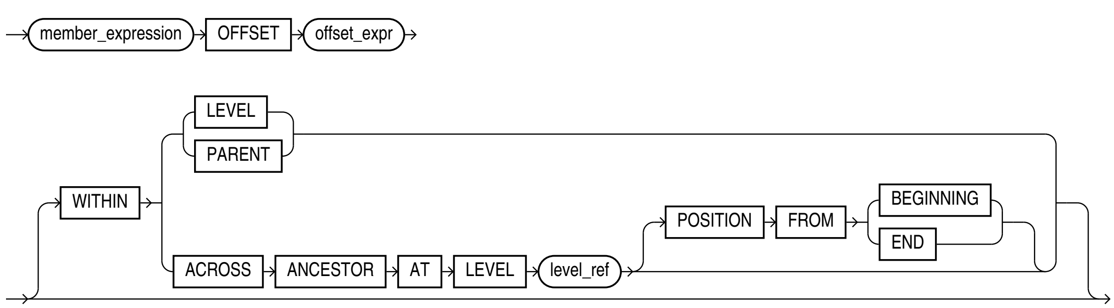
***hier_parent_expression*::=**



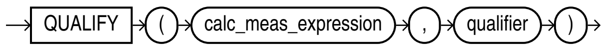
***hier_lead_lag_expression*::=**



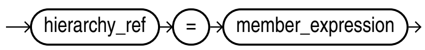
***hier_lead_lag_clause*::=**



***qdr_expression*::=**



***qualifier*::=**



Semantics

av_meas_expression

An expression that performs hierarchical navigation to locate related measure values.

lead_lag_expression

An expression that specifies a lead or lag operation that locates a related measure value by navigating forward or backward by some number of members within a hierarchy.

The *`calc_meas_expression`* parameter is evaluated in the new context created by the *`lead_lag_expression`*. This context has the same members as the outer context, except that the member of the specified hierarchy is changed to the related member specified by the lead or lag operation. The lead or lag function is run over the hierarchy members specified by the *`lead_lag_clause`* parameter.

lead_lag_function_name

The lead or lag function may be one of the following:

- `LAG` returns the measure value of an earlier member.
- `LAG_DIFF` returns the difference between the measure value of the current member and the measure value of an earlier member.
- `LAG_DIFF_PERCENT` returns the percent difference between the measure value of the current member and the measure value of an earlier member.
- `LEAD` returns the measure value of a later member.
- `LEAD_DIFF` returns the difference between the measure value of the current member and the measure value of a later member.
- `LEAD_DIFF_PERCENT` returns the percent difference between the measure value of the current member and the measure value of a later member.

lead_lag_clause

Specifies the hierarchy to evaluate and an offset value. The parameters of the *`lead_lag_clause`* are the following:

- `HIERARCHY` *`hierarchy_ref`* specifies the name of a hierarchy in the analytic view.

- `OFFSET` *`offset_expr`* specifies a *`calc_meas_expression`* that resolves to a number. The number specifies how many members to move either forward or backward from the current member. The ordering of members within a level is determined by the definition of the attribute dimension used by the hierarchy.
- `WITHIN LEVEL` specifies locating the related member by moving forward or backward by the offset number of members within the members that have the same level depth as the current member. The ordering of members within the level is determined by the definition of the attribute dimension used by the hierarchy.

  The `WITHIN LEVEL` operation is the default if neither the `WITHIN LEVEL` nor the `ACROSS ANCESTOR AT LEVEL` keywords are specified.
- `WITHIN PARENT` specifies locating the related member by moving forward or backward by the offset number of members within the members that have the same parent as the current member.
- `ACROSS ANCESTOR AT LEVEL` *`level_ref`* specifies locating the related member by navigating up to the ancestor (or to the member itself if no ancestor exists) of the current member at the level specified by *`level_ref`*, and noting the position of each ancestor member (including the member itself) within its parent. The *`level_ref`* parameter is the name of a level in the specified hierarchy.

  Once the ancestor member is found, navigation moves either forward or backward the offset number of members within the members that have the same depth as the ancestor member. After locating the related ancestor, navigation proceeds back down the hierarchy from this member, matching the position within the parent as recorded on the way up (in reverse order). The position within the parent is either an offset from the first child or the last child depending on whether `POSITION FROM BEGINNING` or `POSITION FROM END` is specified. The default value is `POSITION FROM BEGINNING`. The ordering of members within the level is determined by the definition of the attribute dimension used by the hierarchy.

window_expression

A *`window_expression`* selects the set of members that are in the specified range starting from the current member and that are at the same depth as the current member. You can further restrict the selection of members by specifying a hierarchical relationship using a WITHIN phrase. Aggregation is then performed over the selected measure values to produce a single result for the expression.

The parameters for a *`window_expression`* are the following:

- *`aggregate_function`* is any existing SQL aggregate function except `COLLECT`, `GROUP_ID`, `GROUPING`, `GROUPING_ID`, `SYS_XMLAGG`, `XMLAGG`, and any multi-argument function. A user defined aggregate function is also allowed. The arguments to the aggregate function are *`calc_meas_expression`* expressions. These expressions are evaluated using the outer context, with the member of the specified hierarchy changed to each member in the related range. Therefore, each expression argument is evaluated once per related member. The results are then aggregated using the *`aggregate_function`*.
- `OVER (`*`window_clause`*`)` specifies the hierarchy to use and the boundaries of the window to consider.

> **See Also:**
>
> Aggregate Functions

window_clause

The *window_clause* parameter selects a range of members related to the current member. The range is between the members specified by the *preceding_boundary* or *following_boundary* parameters. The range is always computed over members at the same level as the current member.

The parameters for a *window_clause* are the following:

- `HIERARCHY` *hierarchy_ref* specifies the name of the hierarchy in the analytic view.
- `BETWEEN` *preceding_boundary* or *following_boundary* defines the set of members to relate to the current member.
- `WITHIN LEVEL` selects the related members by applying the boundary clause to all members of the current level. This is the default when the `WITHIN` keyword is not specified.
- `WITHIN PARENT` selects the related members by applying the boundary clause to all members that share a parent with the current member.
- `WITHIN ANCESTOR AT LEVEL` selects the related members by applying the boundary clause to all members at the current depth that share an ancestor (or is the member itself) at the specified level with the current member. The value of the window expression is `NULL` if the current member is above the specified level. If the level is not in the specified hierarchy, then an error occurs.

preceding_boundary

The *preceding_boundary* parameter defines a range of members from the specified number of members backward in the level from the current member and forward to the specified end of the boundary. The following parameters specify the range:

- `UNBOUNDED PRECEDING` begins the range at the first member in the level.
- *offset_expr* `PRECEDING` begins the range at the *offset_expr* number of members backward from the current member. The *offset_expr* expression is a *calc_meas_expression* that resolves to a number. If the offset number is greater than the number of members from the current member to the first member in the level, than the first member is used as the start of the range.
- `CURRENT MEMBER` ends the range at the current member.
- *offset_expr* `PRECEDING` ends the range at the member that is *offset_expr* backward from the current member.
- *offset_expr* `FOLLOWING` ends the range at the member that is *offset_expr* forward from the current member.
- `UNBOUNDED FOLLOWING` ends the range at the last member in the level.

following_boundary

The *following_boundary* parameter defines a range of members from the specified number of members from the current member forward to the specified end of the range. The following parameters specify the range:

- CURRENT MEMBER begins the range at the current member.
- *offset_expr* FOLLOWING begins the range at the member that is *offset_expr* forward from the current member.
- *offset_expr* FOLLOWING ends the range at the member that is *offset_expr* forward from the current member.
- UNBOUNDED FOLLOWING ends the range at the last member in the level.

hierarchy_ref

A reference to a hierarchy of an analytic view. The *hier_alias* parameter specifies the name of a hierarchy in the definition of the analytic view. You may use double quotes to escape special characters or preserve case, or both.

The optional *attr_dim_alias* parameter is the name of an alias specified in the definition of the analytic view. You may use the *attr_dim_alias* parameter to resolve the ambiguity if the specified hierarchy alias conflicts with another hierarchy alias in the analytic view or if an attribute dimension is used more than once in the analytic view definition. You may use the *attr_dim_alias* parameter even when a name conflict does not exist.

share_of_expression

A *share_of_expression* expression calculates the ratio of an expression's value for the current context over the expression's value at a related context. The expression is a *calc_meas_expression* that is evaluated at the current context and the related context. The *share_clause* specification determines the related context to use.

share_clause

A *share_clause* modifies the outer context by setting the member for the specified hierarchy to a related member.

The parameters of the share clause are the following:

- HIERARCHY *hierarchy_ref* specifies the name of the hierarchy that is the outer context for the *share_of_expression* calculations.
- PARENT specifies that the related member is the parent of the current member.
- LEVEL *level_ref* specifies that the related member is the ancestor (or is the member itself) of the current member at the specified level in the hierarchy. If the current member is above the specified level, then NULL is returned for the share expression. If the level is not in the hierarchy, then an error occurs.
- MEMBER *member_expression* specifies that the related member is the member returned after evaluating the *member_expression* in the current context. If the value of the specified member is NULL, then NULL is returned for the share expression.

level_member_literal

A `level_member_literal` is an expression that resolves to a single member of the hierarchy. The expression contains the name of the level and one or more member keys. The member key or keys may be identified by position or by name. If the specified level is not in the context hierarchy, then an error occurs.

pos_member_keys

The `member_key_expr` expression resolves to the key value for the member. When specified by position, all components of the key must be given in the order found in the `ALL_HIER_LEVEL_ID_ATTRS` dictionary view. For a hierarchy in which the specified level is not determined by the child level, then all member key values of all such child levels must be provided preceding the current level's member key or keys. Duplicate key components are only specified the first time they appear.

The primary key is used when `level_member_literal` is specified using the `pos_member_keys` phrase. You can reference an alternate key by using the `named_member_keys` phrase.

named_member_keys

The `member_key_expr` expression resolves to the key value for the member. The `attr_name` parameter is an identifier for the name of the attribute. If all of the attribute names do not make up a key or alternate key of the specified level, then an error occurs.

When specified by name, all components of the key must be given and all must use the attribute *name* = *value* form, in any order. For a hierarchy in which the specified level is not determined by the child level, then all member key values of all such child levels must be provided, also using the named form. Duplicate key components are only specified once.

hier_navigation_expression

A `hier_navigation_expression` expression navigates from the specified member to a different member in the hierarchy.

hier_ancestor_expression

Navigates from the specified member to the ancestor member (or to the member itself) at the specified level or depth. The depth is specified as an expression that must resolve to a number. If the member is at a level or depth above the specified member or the member is `NULL`, then `NULL` is returned for the expression value. If the specified level is not in the context hierarchy, then an error occurs.

member_expression

A `member_expression` evaluates to a member of the specified hierarchy. The hierarchy can always be determined from the outer expression (enforced by the syntax). A `member_expression` can be one of the following:

- `level_member_literal` is an expression that evaluates to a hierarchy member.
- `hier_navigation_expr` is an expression that relates one member of the hierarchy to another member.

- `CURRENT MEMBER` specifies the member of the hierarchy as determined by the outer context.
- `NULL` is a way to specify a non-existent member.
- `ALL` specifies the single topmost member of every hierarchy.

hier_parent_expression

Navigates from the specified member to the parent member.

hier_lead_lag_expression

Navigates from the specified member to a related member by moving forward or backward some number of members within the context hierarchy. The `HIER_LEAD` keyword returns a later member. The `HIER_LAG` keyword returns an earlier member.

hier_lead_lag_clause

Navigates the *`offset_expr`* number of members forward or backward from the specified member. The ordering of members within a level is specified in the definition of the attribute dimension.

The optional parameters of *`hier_lead_lag_clause`* are the following:

- `WITHIN LEVEL` locates the related member by moving forward or backward *`offset_expr`* members within the members that have the same depth as the current member. The ordering of members within the level is determined by the definition of the attribute dimension. The `WITHIN LEVEL` operation is the default if neither the `WITHIN` nor the `ACROSS` keywords are used.
- `WITHIN PARENT` locates the related member by moving forward or backward *`offset_expr`* members within the members that have the same depth as the current member, but only considers members that share a parent with the current member. The ordering of members within the level is determined by the definition of the attribute dimension.
- `WITHIN ACROSS ANCESTOR AT LEVEL` locates the related member by navigating up to the ancestor of the current member (or to the member itself) at the specified level, noting the position of each ancestor member (including the member itself) within its parent. Once the ancestor member is found, navigation moves forward or backward *`offset_expr`* members within the members that have the same depth as the ancestor member.

  After locating the related ancestor, navigation moves back down the hierarchy from that member, matching the position within the parent as recorded on the way up (in reverse order). The position within the parent is either an offset from the first child or the last child depending on whether `POSITION FROM BEGINNING` or `POSITION FROM END` is specified, defaulting to `POSITION FROM BEGINNING`. The ordering of members within the level is determined by the definition of the attribute dimension.

qdr_expression

A *`qdr_expression`* is a qualified data reference that evaluates the specified *`calc_meas_expression`* in a new context and sets the hierarchy member to the new value.

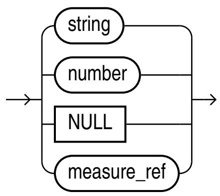
qualifier

A qualifier modifies the outer context by setting the member for the specified hierarchy to the member resulting from evaluating *`member_expression`*. If *`member_expression`* is `NULL`, then the result of the *`qdr_expression`* selection is `NULL`.

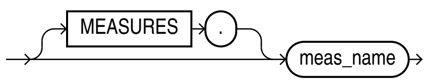
Analytic View Simple Expressions

A calculated measure expression may be an analytic view simple expression. An analytic view simple expression may be any SQL literal or a reference to a measure in an analytic view.

***av_simple_expression*::=**



***measure_ref*::=**



measure_ref

A reference to a measure in an analytic view. The `meas_name` parameter must be the name of a measure in the definition of an analytic view. You may use double quotes to escape special characters or preserve case, or both. If the measure name conflicts with the name of an attribute dimension used by the analytic view, then you may use the optional `MEASURES` keyword to resolve the ambiguity. You may use the `MEASURES` keyword even when a name conflict does not exist.

Single Row Function Expression

A calculated measure expression may be any SQL single row function expression.

> **See Also:**
>
> - Function Expressions
> - Single-Row Functions

Examples of Calculated Measure Expressions

This topic contains examples that show calculated measures defined in the `MEASURES` clause of an analytic view. For more examples, see the tutorials on analytic views at the SQL Live website at https://livesql.oracle.com/apex/livesql/file/index.html.

Examples of LAG Expressions

These calculated measures different `LAG` operations.

```
-- These calculated measures are from the measures_clause of the
-- sales_av analytic view.
MEASURES
 (sales FACT sales,                          -- A base measure
  units FACT units,                          -- A base measure
  sales_prior_period AS                      -- Calculated measures
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1)),
  sales_year_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  chg_sales_year_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  pct_chg_sales_year_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  sales_qtr_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  chg_sales_qtr_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  pct_chg_sales_qtr_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter))
 )
```

Example of a Window Expression

This calculated measure uses a window operation.

```
MEASURES
 (sales FACT sales,
  units FACT units,
  sales_qtd AS
    (SUM(sales) OVER (HIERARCHY time_hier
     BETWEEN UNBOUNDED PRECEDING AND CURRENT MEMBER
     WITHIN ANCESTOR AT LEVEL QUARTER)),
  sales_ytd AS
    (SUM(sales) OVER (HIERARCHY time_hier
     BETWEEN UNBOUNDED PRECEDING AND CURRENT MEMBER
     WITHIN ANCESTOR AT LEVEL YEAR))
 )
```

Examples of SHARE OF Expressions

These calculated measures use SHARE OF expressions.

```
MEASURES
 (sales FACT sales,
  units FACT units,
 sales_shr_parent_prod AS
   (SHARE_OF(sales HIERARCHY product_hier PARENT)),
 sales_shr_parent_geog AS
   (SHARE_OF(sales HIERARCHY geography_hier PARENT)),
 sales_shr_region AS
   (SHARE_OF(sales HIERARCHY geography_hier LEVEL REGION))
 )
```

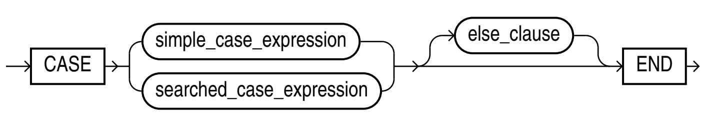
Examples of QDR Expressions

These calculated measures use the QUALIFY keyword to specify qualified data reference expressions.

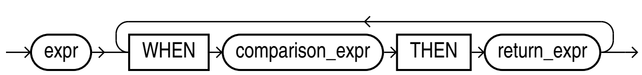
```
MEASURES
 (sales FACT sales,
  units FACT units,
  sales_2011 AS
    (QUALIFY (sales, time_hier = year['11'])),
  sales_pct_chg_2011 AS
    ((sales - (QUALIFY (sales, time_hier = year['11']))) /
    (QUALIFY (sales, time_hier = year['11'])))
 )
```

CASE Expressions

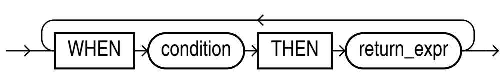
CASE expressions let you use IF ... THEN ... ELSE logic in SQL statements without having to invoke procedures. The syntax is:



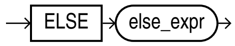
***simple_case_expression*::=**



***searched_case_expression*::=**



***else_clause*::=**



In a simple `CASE` expression, Oracle Database searches for the first `WHEN` ... `THEN` pair for which *`expr`* is equal to *`comparison_expr`* and returns *`return_expr`*. If none of the `WHEN` ... `THEN` pairs meet this condition, and an `ELSE` clause exists, then Oracle returns *`else_expr`*. Otherwise, Oracle returns null.

In a searched `CASE` expression, Oracle searches from left to right until it finds an occurrence of *`condition`* that is true, and then returns *`return_expr`*. If no *`condition`* is found to be true, and an `ELSE` clause exists, then Oracle returns *`else_expr`*. Otherwise, Oracle returns null.

Oracle Database uses **short-circuit evaluation**. For a simple `CASE` expression, the database evaluates each *`comparison_expr`* value only before comparing it to *`expr`*, rather than evaluating all *`comparison_expr`* values before comparing any of them with *`expr`*. Consequently, Oracle never evaluates a *`comparison_expr`* if a previous *`comparison_expr`* is equal to *`expr`*. For a searched `CASE` expression, the database evaluates each *`condition`* to determine whether it is true, and never evaluates a *`condition`* if the previous *`condition`* was true.

For a simple `CASE` expression, the *`expr`* and all *`comparison_expr`* values must either have the same data type (`CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`, `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`) or must all have a numeric data type. If all expressions have a numeric data type, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

For both simple and searched `CASE` expressions, all of the *`return_expr`*s must either have the same data type (`CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`, `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`) or must all have a numeric data type. If all return expressions have a numeric data type, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

The maximum number of arguments in a `CASE` expression is 65535. All expressions count toward this limit, including the initial expression of a simple `CASE` expression and the optional `ELSE` expression. Each `WHEN` ... `THEN` pair counts as two arguments. To avoid exceeding this limit, you can nest `CASE` expressions so that the *`return_expr`* itself is a `CASE` expression.

The comparison performed by the simple `CASE` expression is collation-sensitive if the compared arguments have a character data type (`CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`). The collation determination rules determine the collation to use.

> **See Also:**
>
> - Table 2-8 for more information on implicit conversion
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation and determination rules for the `CASE` expression
> - Numeric Precedence for information on numeric precedence
> - COALESCE and NULLIF for alternative forms of `CASE` logic
> - *Oracle Database Data Warehousing Guide* for examples using various forms of the `CASE` expression

Simple CASE Example

For each customer in the sample `oe.customers` table, the following statement lists the credit limit as "Low" if it equals $100, "High" if it equals $5000, and "Medium" if it equals anything else.

```
SELECT cust_last_name,
   CASE credit_limit WHEN 100 THEN 'Low'
   WHEN 5000 THEN 'High'
   ELSE 'Medium' END AS credit
   FROM customers
   ORDER BY cust_last_name, credit;

CUST_LAST_NAME       CREDIT
-------------------- ------
Adjani               Medium
Adjani               Medium
Alexander            Medium
Alexander            Medium
Altman               High
Altman               Medium
. . .
```

Searched CASE Example

The following statement finds the average salary of the employees in the sample table `oe.employees`, using $2000 as the lowest salary possible:

```
SELECT AVG(CASE WHEN e.salary > 2000 THEN e.salary
   ELSE 2000 END) "Average Salary" FROM employees e;

Average Salary
--------------
    6461.68224
```

Column Expressions

A column expression, which is designated as *`column_expression`* in subsequent syntax diagrams, is a limited form of *`expr`*. A column expression can be a simple expression, compound expression, function expression, or expression list, but it can contain only the following forms of expression:

- Columns of the subject table — the table being created, altered, or indexed

- Constants (strings or numbers)
- Deterministic functions — either SQL built-in functions or user-defined functions

No other expression forms described in this chapter are valid. In addition, compound expressions using the `PRIOR` keyword are not supported, nor are aggregate functions.

You can use a column expression for these purposes:

- To create a function-based index.
- To explicitly or implicitly define a virtual column. When you define a virtual column, the defining *`column_expression`* must refer only to columns of the subject table that have already been defined, in the current statement or in a prior statement.

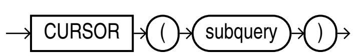
The combined components of a column expression must be deterministic. That is, the same set of input values must return the same set of output values.

See Also:

Simple Expressions , Compound Expressions , Function Expressions , and Expression Lists for information on these forms of *`expr`*

CURSOR Expressions

A `CURSOR` expression returns a nested cursor. This form of expression is equivalent to the PL/SQL `REF CURSOR` and can be passed as a `REF CURSOR` argument to a function.



A nested cursor is implicitly opened when the cursor expression is evaluated. For example, if the cursor expression appears in a select list, a nested cursor will be opened for each row fetched by the query. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is reexecuted
- The parent cursor is closed
- The parent cursor is cancelled
- An error arises during fetch on one of its parent cursors (it is closed as part of the clean-up)

Restrictions on CURSOR Expressions

The following restrictions apply to `CURSOR` expressions:

- If the enclosing statement is not a `SELECT` statement, then nested cursors can appear only as `REF CURSOR` arguments of a procedure.
- If the enclosing statement is a `SELECT` statement, then nested cursors can also appear in the outermost select list of the query specification or in the outermost select list of another nested cursor.

- Nested cursors cannot appear in views.
- You cannot perform `BIND` and `EXECUTE` operations on nested cursors.

Examples

The following example shows the use of a `CURSOR` expression in the select list of a query:

```
SELECT department_name, CURSOR(SELECT salary, commission_pct
   FROM employees e
   WHERE e.department_id = d.department_id)
   FROM departments d
   ORDER BY department_name;
```

The next example shows the use of a `CURSOR` expression as a function argument. The example begins by creating a function in the sample `OE` schema that can accept the `REF CURSOR` argument. (The PL/SQL function body is shown in italics.)

```
CREATE FUNCTION f(cur SYS_REFCURSOR, mgr_hiredate DATE)
   RETURN NUMBER IS
   emp_hiredate DATE;
   before number :=0;
   after number:=0;
begin
  loop
    fetch cur into emp_hiredate;
    exit when cur%NOTFOUND;
    if emp_hiredate > mgr_hiredate then
      after:=after+1;
    else
      before:=before+1;
    end if;
  end loop;
  close cur;
  if before > after then
    return 1;
  else
    return 0;
  end if;
end;
/
```

The function accepts a cursor and a date. The function expects the cursor to be a query returning a set of dates. The following query uses the function to find those managers in the sample `employees` table, most of whose employees were hired before the manager.

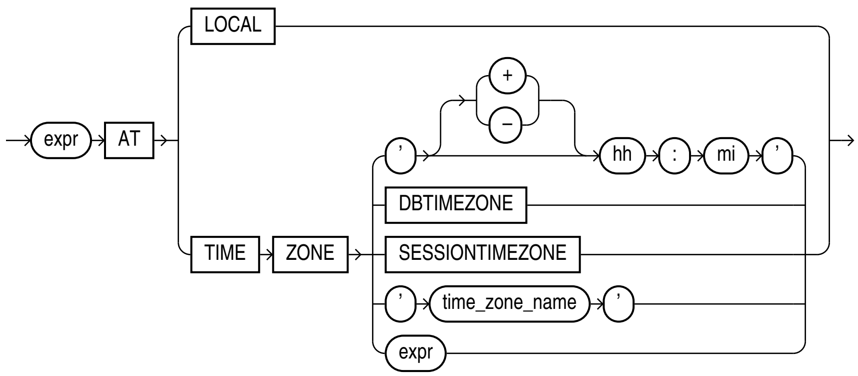
```
SELECT e1.last_name FROM employees e1
   WHERE f(
   CURSOR(SELECT e2.hire_date FROM employees e2
   WHERE e1.employee_id = e2.manager_id),
   e1.hire_date) = 1
   ORDER BY last_name;

LAST_NAME
-------------------------
Cambrault
Higgins
Hunold
Kochhar
```

```
Mourgos
Zlotkey
```

Datetime Expressions

A datetime expression yields a value of one of the datetime data types.

***datetime_expression*::=**



The initial *expr* is any expression, except a scalar subquery expression, that evaluates to a value of data type TIMESTAMP, TIMESTAMP WITH TIME ZONE, or TIMESTAMP WITH LOCAL TIME ZONE. The DATE data type is not supported. If this *expr* is itself a *datetime_expression*, then it must be enclosed in parentheses.

Datetimes and intervals can be combined according to the rules defined in Table 2-5. The three combinations that yield datetime values are valid in a datetime expression.

If you specify AT LOCAL, then Oracle uses the current session time zone.

The settings for AT TIME ZONE are interpreted as follows:

- The string '[+|-]*hh*:*mi*' specifies a time zone as an offset from UTC. For *hh*, specify the number of hours. For *mi*, specify the number of minutes.
- DBTIMEZONE: Oracle uses the database time zone established (explicitly or by default) during database creation.
- SESSIONTIMEZONE: Oracle uses the session time zone established by default or in the most recent ALTER SESSION statement.
- *time_zone_name*: Oracle returns the *datetime_value_expr* in the time zone indicated by *time_zone_name*. For a listing of valid time zone region names, query the V$TIMEZONE_NAMES dynamic performance view.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for a complete listing of the time zone region names in both files
> - *Oracle Database Reference* for information on the dynamic performance views

- `expr`: If `expr` returns a character string with a valid time zone format, then Oracle returns the input in that time zone. Otherwise, Oracle returns an error.

Example

The following example converts the datetime value of one time zone to another time zone:

```
SELECT FROM_TZ(CAST(TO_DATE('1999-12-01 11:00:00',
      'YYYY-MM-DD HH:MI:SS') AS TIMESTAMP), 'America/New_York')
   AT TIME ZONE 'America/Los_Angeles' "West Coast Time"
   FROM DUAL;

West Coast Time
------------------------------------------------
01-DEC-99 08.00.00.000000 AM AMERICA/LOS_ANGELES
```

Function Expressions

You can use any built-in SQL function or user-defined function as an expression. Some valid built-in function expressions are:

```
LENGTH('BLAKE')
ROUND(1234.567*43)
SYSDATE
```

> **See Also:**
>
> About SQL Functions ' and Aggregate Functions for information on built-in functions

A user-defined function expression specifies a call to:

- A function in an Oracle-supplied package (see *Oracle Database PL/SQL Packages and Types Reference*)
- A function in a user-defined package or type or in a standalone user-defined function (see About User-Defined Functions)
- A user-defined function or operator (see CREATE OPERATOR , CREATE FUNCTION , and *Oracle Database Data Cartridge Developer's Guide*)

Some valid user-defined function expressions are:

```
circle_area(radius)
payroll.tax_rate(empno)
```

```
hr.employees.comm_pct@remote(dependents, empno)
DBMS_LOB.getlength(column_name)
my_function(a_column)
```

In a user-defined function being used as an expression, positional, named, and mixed notation are supported. For example, all of the following notations are correct:

```
CALL my_function(arg1 => 3, arg2 => 4) ...

CALL my_function(3, 4) ...

CALL my_function(3, arg2 => 4) ...
```

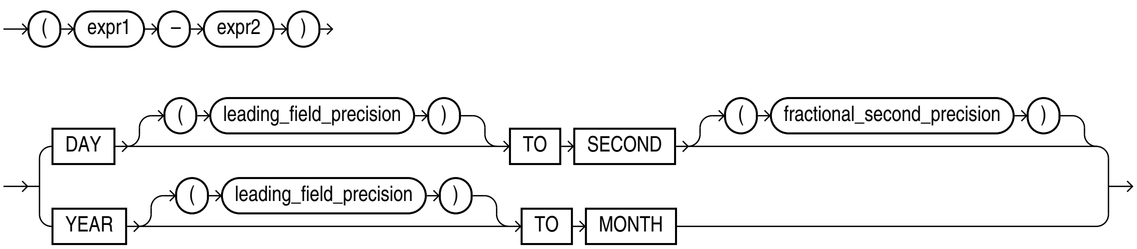
Restriction on User-Defined Function Expressions

You cannot pass arguments of object type or `XMLType` to remote functions and procedures.

Interval Expressions

An interval expression yields a value of `INTERVAL YEAR TO MONTH` or `INTERVAL DAY TO SECOND`.

***interval_expression*::=**



The expressions *`expr1`* and *`expr2`* can be any expressions that evaluate to values of data type `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `TIMESTAMP WITH LOCAL TIME ZONE`.

Datetimes and intervals can be combined according to the rules defined in Table 2-5. The six combinations that yield interval values are valid in an interval expression.

Both *`leading_field_precision`* and *`fractional_second_precision`* can be any integer from 0 to 9. If you omit the *`leading_field_precision`* for either `DAY` or `YEAR`, then Oracle Database uses the default value of 2. If you omit the *`fractional_second_precision`* for second, then the database uses the default value of 6. If the value returned by a query contains more digits that the default precision, then Oracle Database returns an error. Therefore, it is good practice to specify a precision that you know will be at least as large as any value returned by the query.

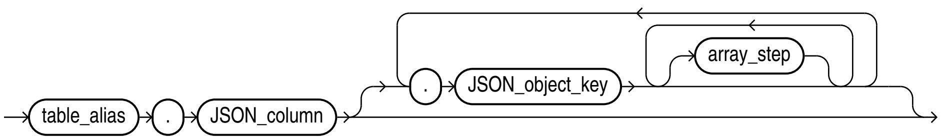
For example, the following statement subtracts the value of the `order_date` column in the sample table `orders` (a datetime value) from the system timestamp (another datetime value) to yield an interval value expression. It is not known how many days ago the oldest order was placed, so the maximum value of 9 for the `DAY` leading field precision is specified:

```
SELECT (SYSTIMESTAMP - order_date) DAY(9) TO SECOND FROM orders
   WHERE order_id = 2458;
```

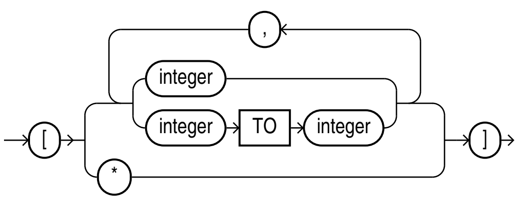
JSON Object Access Expressions

A JSON object access expression is used only when querying a column of JavaScript Object Notation (JSON) data. It yields a character string that contains one or more JSON values found in that data. The syntax for this type of expression is called dot-notation syntax.

JSON_object_access_expr::=



array_step::=



- For *`table_alias`*, specify the alias for the table that contains the column of JSON data. This table alias is required and must be assigned to the table elsewhere in the SQL statement.
- For *`JSON_column`*, specify the name of the column of JSON data. The column must be of data type `VARCHAR2`, `CLOB`, or `BLOB` and an `IS JSON` check constraint must be defined on the column.
- You can optionally specify one or more JSON object keys. The object keys allow you to target specific JSON values in the JSON data. The first *`JSON_object_key`* must be a case-sensitive match to the key (property) name of an object member in the top level of the JSON data. If the value of that object member is another JSON object, then you can specify a second *`JSON_object_key`* that matches the key name of a member of that object, and so on. If a JSON array is encountered during any of these iterations, and you do not specify an *`array_step`*, then the array is implicitly unwrapped and the elements of the array are evaluated using the *`JSON_object_key`*.
- If the JSON value is an array, then you can optionally specify one or more *`array_step`* clauses. This allows you to access specific elements of the JSON array.
  - Use *`integer`* to specify the element at index *`integer`* in a JSON array. Use *`integer`* `TO` *`integer`* to specify the range of elements between the two index *`integer`* values, inclusive. If the specified elements exist in the JSON array being evaluated, then the array step results in a match to those elements. Otherwise, the array step does not result in a match. The first element in a JSON array has index 0.

  - Use the asterisk wildcard symbol (*) to specify all elements in a JSON array. If the JSON array being evaluated contains at least one element, then the array step results in a match to all elements in the JSON array. Otherwise, the array step does not result in a match.

A JSON object access expression yields a character string of data type `VARCHAR2(4000)`, which contains the targeted JSON value(s) as follows:

- For a single targeted value, the character string contains that value, whether it is a JSON scalar value, object, or array.
- For multiple targeted values, the character string contains a JSON array whose elements are those values.

If you omit *`JSON_object_key`*, then the expression yields a character string that contains the JSON data in its entirety. In this case, the character string is of the same data type as the column of JSON data being queried.

A JSON object access expression cannot return a value larger than 4K bytes. If the value surpasses this limit, then the expression returns null. To obtain the actual value, instead use the JSON_QUERY function or the JSON_VALUE function and specify an appropriate return type with the `RETURNING` clause.

The collation derivation rules for the JSON object access expression are the same as for the `JSON_QUERY` function.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for the `JSON_QUERY` function

Examples

The following examples use the `j_purchaseorder` table, which is created in Creating a Table That Contains a JSON Document: Example. This table contains a column of JSON data called `po_document`. These examples return JSON values from column `po_document`.

The following statement returns the value of the property with key name `PONumber`:

```
SELECT po.po_document.PONumber
  FROM j_purchaseorder po;

PONumber
--------
1600
```

The following statement first targets the property with key name `ShippingInstructions`, whose value is a JSON object. The statement then targets the property with key name `Phone` within that object. The statement returns the value of `Phone`, which is a JSON array.

```
SELECT po.po_document.ShippingInstructions.Phone
  FROM j_purchaseorder po;

SHIPPINGINSTRUCTIONS
-------------------------------------------------------------------------------------
[{"type":"Office","number":"909-555-7307"},{"type":"Mobile","number":"415-555-1234"}]
```

The following statement first targets the property with key name `LineItems`, whose value is a JSON array. The expression implicitly unwraps the array and evaluates its elements, which are JSON objects. Next, the statement targets the properties with key name `Part`, within the unwrapped objects, and finds two objects. The statement then targets the properties with key name `Description` within those two objects and finds string values. Because more than one value is returned, the values are returned as elements of a JSON array.

```
SELECT po.po_document.LineItems.Part.Description
  FROM j_purchaseorder po;

LINEITEMS
-----------------------------------
[One Magic Christmas,Lethal Weapon]
```

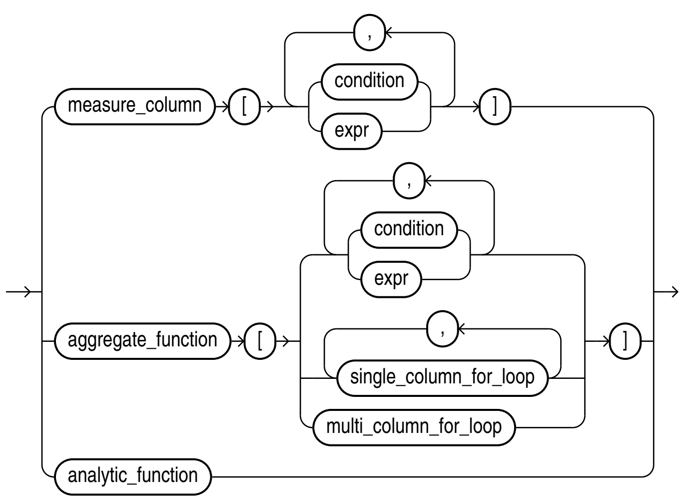
See Also:

Oracle Database JSON Developer's Guide for more information on querying JSON data using dot-notation syntax

Model Expressions

A model expression is used only in the *`model_clause`* of a `SELECT` statement and then only on the right-hand side of a model rule. It yields a value for a cell in a measure column previously defined in the *`model_clause`*. For additional information, refer to *model_clause*.

model_expression::=



When you specify a measure column in a model expression, any conditions and expressions you specify must resolve to single values.

When you specify an aggregate function in a model expression, the argument to the function is a measure column that has been previously defined in the *`model_clause`*. An aggregate function can be used only on the right-hand side of a model rule.

Specifying an analytic function on the right-hand side of the model rule lets you express complex calculations directly in the *model_clause*. The following restrictions apply when using an analytic function in a model expression:

- Analytic functions can be used only in an UPDATE rule.
- You cannot specify an analytic function on the right-hand side of the model rule if the left-hand side of the rule contains a FOR loop or an ORDER BY clause.
- The arguments in the OVER clause of the analytic function cannot contain an aggregate.
- The arguments before the OVER clause of the analytic function cannot contain a cell reference.

> **See Also:**
>
> The MODEL clause: Examples for an example of using an analytic function on the right-hand side of a model rule

When *expr* is itself a model expression, it is referred to as a **nested cell reference**. The following restrictions apply to nested cell references:

- Only one level of nesting is allowed.
- A nested cell reference must be a single-cell reference.
- When AUTOMATIC ORDER is specified in the *model_rules_clause*, a nested cell reference can be used on the left-hand side of a model rule only if the measures used in the nested cell reference remain static.

The model expressions shown below are based on the *model_clause* of the following SELECT statement:

```
SELECT country,prod,year,s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[prod='Mouse Pad', year=2000] =
        s['Mouse Pad', 1998] + s['Mouse Pad', 1999],
      s['Standard Mouse', 2001] = s['Standard Mouse', 2000]
    )
  ORDER BY country, prod, year;
```

The following model expression represents a single cell reference using symbolic notation. It represents the sales of the Mouse Pad for the year 2000.

```
s[prod='Mouse Pad',year=2000]
```

The following model expression represents a multiple cell reference using positional notation, using the CV function. It represents the sales of the current value of the dimension column prod for the year 2001.

```
s[CV(prod), 2001]
```

The following model expression represents an aggregate function. It represents the sum of sales of the Mouse Pad for the years between the current value of the dimension column `year` less two and the current value of the dimension column `year` less one.

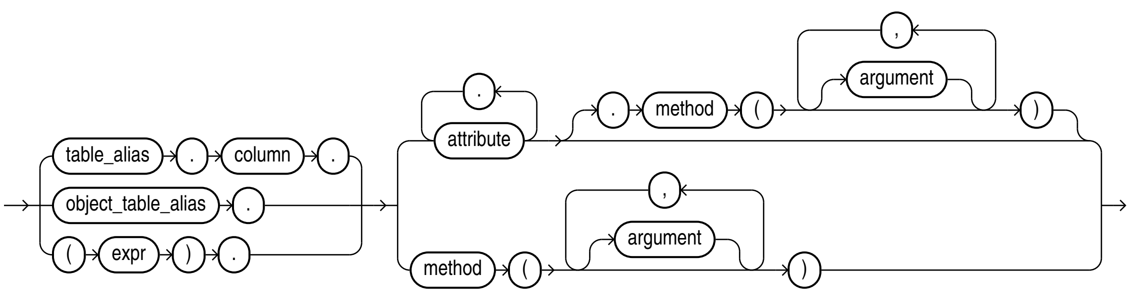
```
SUM(s)['Mouse Pad',year BETWEEN CV()-2 AND CV()-1]
```

> **See Also:**
>
> CV and *model_clause*

Object Access Expressions

An object access expression specifies attribute reference and method invocation.

***object_access_expression*::=**



The column parameter can be an object or `REF` column. If you specify *`expr`*, then it must resolve to an object type.

When a type's member function is invoked in the context of a SQL statement, if the `SELF` argument is null, Oracle returns null and the function is not invoked.

Examples

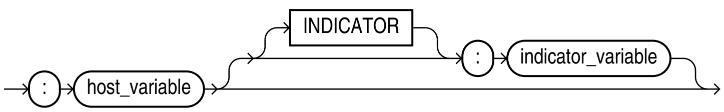
The following example creates a table based on the sample `oe.order_item_typ` object type, and then shows how you would update and select from the object column attributes.

```
CREATE TABLE short_orders (
   sales_rep VARCHAR2(25), item order_item_typ);

UPDATE short_orders s SET sales_rep = 'Unassigned';

SELECT o.item.line_item_id, o.item.quantity FROM short_orders o;
```

Placeholder Expressions

A placeholder expression provides a location in a SQL statement for which a third-generation language bind variable will provide a value. You can specify the placeholder expression with an optional indicator variable. This form of expression can appear only in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

placeholder_expression::=



Some valid placeholder expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules for the placeholder expression with a character data type

Scalar Subquery Expressions

A scalar subquery expression is a subquery that returns exactly one column value from one row. The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, then the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, then Oracle returns an error.

You can use a scalar subquery expression in most syntax that calls for an expression (*`expr`*). In all cases, a scalar subquery must be enclosed in its own parentheses, even if its syntactic location already positions it within parentheses (for example, when the scalar subquery is used as the argument to a built-in function).

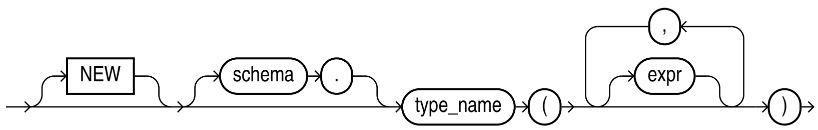
Scalar subqueries are not valid expressions in the following places:

- As default values for columns
- As hash expressions for clusters
- In the `RETURNING` clause of DML statements
- As the basis of a function-based index
- In `CHECK` constraints
- In `GROUP BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

Type Constructor Expressions

A type constructor expression specifies a call to a constructor method. The argument to the type constructor is any expression. Type constructors can be invoked anywhere functions are invoked.

type_constructor_expression::=



The `NEW` keyword applies to constructors for object types but not for collection types. It instructs Oracle to construct a new object by invoking an appropriate constructor. The use of the `NEW` keyword is optional, but it is good practice to specify it.

If *`type_name`* is an **object type**, then the expressions must be an ordered list, where the first argument is a value whose type matches the first attribute of the object type, the second argument is a value whose type matches the second attribute of the object type, and so on. The total number of arguments to the constructor must match the total number of attributes of the object type.

If *`type_name`* is a **varray** or **nested table type**, then the expression list can contain zero or more arguments. Zero arguments implies construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

Restriction on Type Constructor Invocation

In an invocation of a type constructor method, the number of parameters (*`expr`*) specified cannot exceed 999, even if the object type has more than 999 attributes. This limitation applies only when the constructor is called from SQL. For calls from PL/SQL, the PL/SQL limitations apply.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for additional information on constructor methods and *Oracle Database PL/SQL Language Reference* for information on PL/SQL limitations on calls to type constructors

Expression Example

This example uses the `cust_address_typ` type in the sample `oe` schema to show the use of an expression in the call to a constructor method (the PL/SQL is shown in italics):

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;
DECLARE
   myaddr cust_address_typ := cust_address_typ(
     '500 Oracle Parkway', 94065, 'Redwood Shores', 'CA','USA');
   alladdr address_book_t := address_book_t();
BEGIN
   INSERT INTO customers VALUES (
      666999, 'Joe', 'Smith', myaddr, NULL, NULL, NULL, NULL,
      NULL, NULL, NULL, NULL, NULL, NULL, NULL);
END;
/
```

Subquery Example

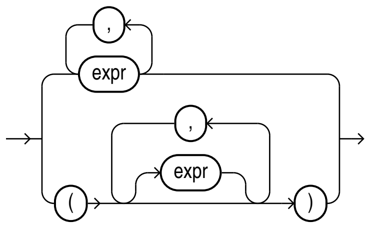
This example uses the `warehouse_typ` type in the sample schema `oe` to illustrate the use of a subquery in the call to the constructor method.

```
CREATE TABLE warehouse_tab OF warehouse_typ;

INSERT INTO warehouse_tab
   VALUES (warehouse_typ(101, 'new_wh', 201));

CREATE TYPE facility_typ AS OBJECT (
   facility_id NUMBER,
   warehouse_ref REF warehouse_typ);

CREATE TABLE buildings (b_id NUMBER, building facility_typ);

INSERT INTO buildings VALUES (10, facility_typ(102,
   (SELECT REF(w) FROM warehouse_tab w
      WHERE warehouse_name = 'new_wh')));

SELECT b.b_id, b.building.facility_id "FAC_ID",
   DEREF(b.building.warehouse_ref) "WH" FROM buildings b;

      B_ID     FAC_ID WH(WAREHOUSE_ID, WAREHOUSE_NAME, LOCATION_ID)
---------- ---------- ---------------------------------------------
        10        102 WAREHOUSE_TYP(101, 'new_wh', 201)
```

Expression Lists

An expression list is a combination of other expressions.

expression_list::=



Expression lists can appear in comparison and membership conditions and in `GROUP BY` clauses of queries and subqueries. An expression lists in a comparision or membership condition is sometimes referred to as a **row value constructor** or **row constructor**.

Comparison and membership conditions appear in the conditions of `WHERE` clauses. They can contain either one or more comma-delimited expressions or one or more sets of expressions where each set contains one or more comma-delimited expressions. In the latter case (multiple sets of expressions):

- Each set is bounded by parentheses
- Each set must contain the same number of expressions

- The number of expressions in each set must match the number of expressions before the operator in the comparison condition or before the `IN` keyword in the membership condition.

A comma-delimited list of expressions can contain no more than 1000 expressions. A comma-delimited list of sets of expressions can contain any number of sets, but each set can contain no more than 1000 expressions.

The following are some valid expression lists in conditions:

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
( ('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA') )
```

In the third example, the number of expressions in each set must equal the number of expressions in the first part of the condition. For example:

```
SELECT * FROM employees
  WHERE (first_name, last_name, email) IN
  (('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA'))
```

See Also:

Comparison Conditions and IN Condition conditions

In a simple `GROUP BY` clause, you can use either the upper or lower form of expression list:

```
SELECT department_id, MIN(salary) min, MAX(salary) max FROM employees
   GROUP BY department_id, salary
   ORDER BY department_id, min, max;

SELECT department_id, MIN(salary) min, MAX(salary) max FROM employees
   GROUP BY (department_id, salary)
   ORDER BY department_id, min, max;
```

In `ROLLUP`, `CUBE`, and `GROUPING SETS` clauses of `GROUP BY` clauses, you can combine individual expressions with sets of expressions in the same expression list. The following example shows several valid grouping sets expression lists in one SQL statement:

```
SELECT
prod_category, prod_subcategory, country_id, cust_city, count(*)
   FROM  products, sales, customers
   WHERE sales.prod_id = products.prod_id
   AND sales.cust_id=customers.cust_id
   AND sales.time_id = '01-oct-00'
   AND customers.cust_year_of_birth BETWEEN 1960 and 1970
GROUP BY GROUPING SETS
  (
   (prod_category, prod_subcategory, country_id, cust_city),
   (prod_category, prod_subcategory, country_id),
   (prod_category, prod_subcategory),
    country_id
  )
ORDER BY prod_category, prod_subcategory, country_id, cust_city;
```

See Also:

SELECT

6 Conditions

A **condition** specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

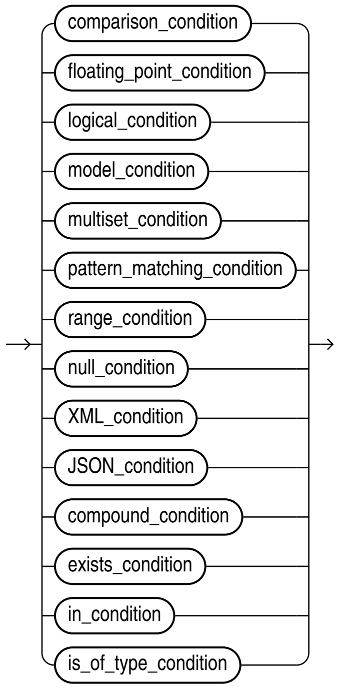
This chapter contains the following sections:

- About SQL Conditions
- Comparison Conditions
- Floating-Point Conditions
- Logical Conditions
- Model Conditions
- Multiset Conditions
- Pattern-matching Conditions
- Null Conditions
- XML Conditions
- SQL/JSON Conditions
- Compound Conditions
- BETWEEN Condition
- EXISTS Condition
- IN Condition
- IS OF *type* Condition

About SQL Conditions

Conditions can have several forms, as shown in the following syntax.

***condition*::=**



If you have installed Oracle Text, then you can create conditions with the built-in operators that are part of that product, including `CONTAINS`, `CATSEARCH`, and `MATCHES`. For more information on these Oracle Text elements, refer to *Oracle Text Reference*.

The sections that follow describe the various forms of conditions. You must use appropriate condition syntax whenever *`condition`* appears in SQL statements.

You can use a condition in the `WHERE` clause of these statements:

- `DELETE`
- `SELECT`
- `UPDATE`

You can use a condition in any of these clauses of the `SELECT` statement:

- `WHERE`
- `START WITH`
- `CONNECT BY`
- `HAVING`

A condition could be said to be of a logical data type, although Oracle Database does not formally support such a data type.

The following simple condition always evaluates to `TRUE`:

```
1 = 1
```

The following more complex condition adds the `salary` value to the `commission_pct` value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 25000:

```
NVL(salary, 0) + NVL(salary + (salary*commission_pct, 0) > 25000)
```

Logical conditions can combine multiple conditions into a single condition. For example, you can use the `AND` condition to combine two conditions:

```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
employees.department_id = departments.department_id
hire_date > '01-JAN-08'
job_id IN ('SA_MAN', 'SA_REP')
salary BETWEEN 5000 AND 10000
commission_pct IS NULL AND salary = 2100
```

Oracle Database does not accept all conditions in all parts of all SQL statements. Refer to the section devoted to a particular SQL statement in this book for information on restrictions on the conditions in that statement.

Condition Precedence

Precedence is the order in which Oracle Database evaluates different conditions in the same expression. When evaluating an expression containing multiple conditions, Oracle evaluates conditions with higher precedence before evaluating those with lower precedence. Oracle evaluates conditions with equal precedence from left to right within an expression, with the following exceptions:

- Left to right evaluation is not guaranteed for multiple conditions connected using `AND`
- Left to right evaluation is not guaranteed for multiple conditions connected using `OR`

Table 6-1 lists the levels of precedence among SQL condition from high to low. Conditions listed on the same line have the same precedence. As the table indicates, Oracle evaluates operators before conditions.

Table 6-1 SQL Condition Precedence

Type of Condition	Purpose
SQL operators are evaluated before SQL conditions	See Operator Precedence
`=, !=, <, >, <=, >=,`	comparison
`IS [NOT] NULL, LIKE, [NOT] BETWEEN, [NOT] IN, EXISTS, IS OF` *`type`*	comparison
`NOT`	exponentiation, logical negation
`AND`	conjunction
`OR`	disjunction

Comparison Conditions

Comparison conditions compare one expression with another. The result of such a comparison can be `TRUE`, `FALSE`, or `UNKNOWN`.

Large objects (LOBs) are not supported in comparison conditions. However, you can use PL/SQL programs for comparisons on `CLOB` data.

When comparing numeric expressions, Oracle uses numeric precedence to determine whether the condition compares `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE` values. Refer to Numeric Precedence for information on numeric precedence.

When comparing character expressions, Oracle uses the rules described in Data Type Comparison Rules . The rules define how the character sets of the expressions are aligned before the comparison, the use of binary or linguistic comparison (collation), the use of blank-padded comparison semantics, and the restrictions resulting from limits imposed on collation keys, including reporting of the error `ORA-12742: unable to create the collation key`.

Two objects of nonscalar type are comparable if they are of the same named type and there is a one-to-one correspondence between their elements. In addition, nested tables of user-defined object types, even if their elements are comparable, must have `MAP` methods defined on them to be used in equality or `IN` conditions.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for information on using `MAP` methods to compare objects

Table 6-2 lists comparison conditions.

Table 6-2 Comparison Conditions

Type of Condition	Purpose	Example

=	Equality test.	`SELECT *`
		`  FROM employees`
		`  WHERE salary = 2500`
		`  ORDER BY employee_id;`

!=
 ^=
 <>
 logical_negation_symbol=
 (Note 1)
	Inequality test. Some forms of the inequality condition may be unavailable on some platforms.	`SELECT *`
		`  FROM employees`
		`  WHERE salary != 2500`
		`  ORDER BY employee_id;`

Table 6-2 (Cont.) Comparison Conditions

Type of Condition	Purpose	Example

>
 	
<
	Greater-than and less-than tests.	`SELECT * FROM employees`
		`  WHERE salary > 2500`
		`  ORDER BY employee_id;`
		`SELECT * FROM employees`
		`  WHERE salary < 2500`
		`  ORDER BY employee_id;`

>=
 	
<=
	Greater-than-or-equal-to and less-than-or-equal-to tests.	`SELECT * FROM employees`
		`  WHERE salary >= 2500`
		`  ORDER BY employee_id;`
		`SELECT * FROM employees`
		`  WHERE salary <= 2500`
		`  ORDER BY employee_id;`

`op ANY`
 	
`op SOME`
	"op" must be one of =, !=, >, <, <=, or >=.
	
	`op ANY` compares a value on the left side either to each value in a list, or to each value returned by a query, whichever is specified on the right side, using the condition `op`.
	
	If any of these comparisons returns TRUE, `op ANY` returns `TRUE`.
	
	If all of these comparisons return `FALSE`, or the subquery on the right side returns no rows, `op ANY` returns `FALSE`. Otherwise, the return value is `UNKNOWN`.
	
	`op ANY` and `op SOME` are synonymous.
		`SELECT * FROM employees`
		`  WHERE salary = ANY`
		`  (SELECT salary`
		`   FROM employees`
		`  WHERE department_id = 30)`
		`  ORDER BY employee_id;`

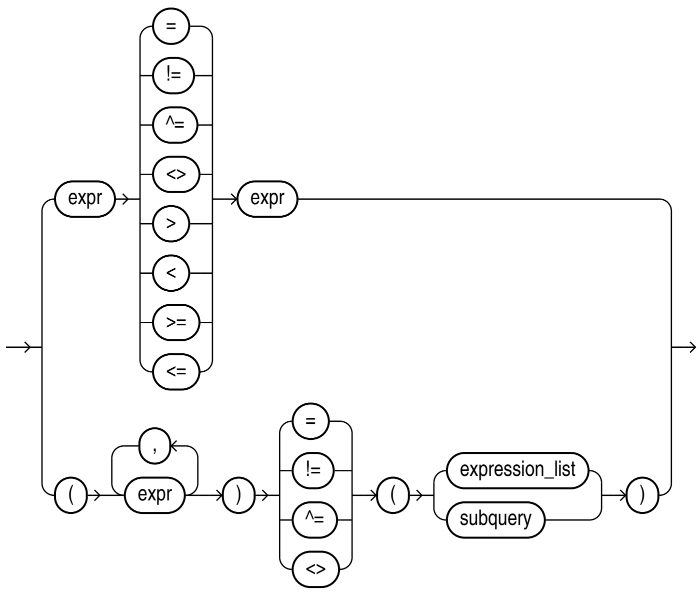
`op ALL`	"op" must be one of =, !=, >, <, <=, or >=.
	
	`op ALL` compares a value on the left side either to each value in a list, or to each value returned by a subquery, whichever is specified on the right side, using the condition `op`.
	
	If any of these comparisons returns `FALSE`, `op ANY` returns `FALSE`.
	
	If all of these comparisons return `TRUE`, or the subquery on the right side returns no rows, `op ALL` returns `TRUE` . Otherwise, the return value is `UNKNOWN`.
		`SELECT * FROM employees`
		`  WHERE salary >=`
		`  ALL (1400, 3000)`
		`  ORDER BY employee_id;`

Note 1: *`logical_negation_symbol`* is the ASCII character with decimal value 170.

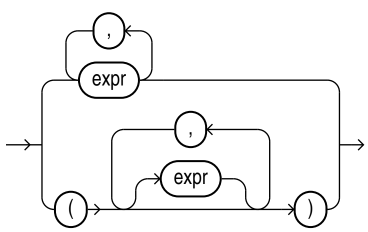
Simple Comparison Conditions

A simple comparison condition specifies a comparison with expressions or subquery results.

***simple_comparison_condition*::=**



***expression_list*::=**



If you use the lower form of this condition with a single expression to the left of the operator, then you can use the upper or lower form of `expression_list`. If you use the lower form of this condition with multiple expressions to the left of the operator, then you must use the lower form of `expression_list`. In either case, the expressions in `expression_list` must match in number and data type the expressions to the left of the operator. If you specify *`subquery`*, then the values returned by the subquery must match in number and data type the expressions to the left of the operator.

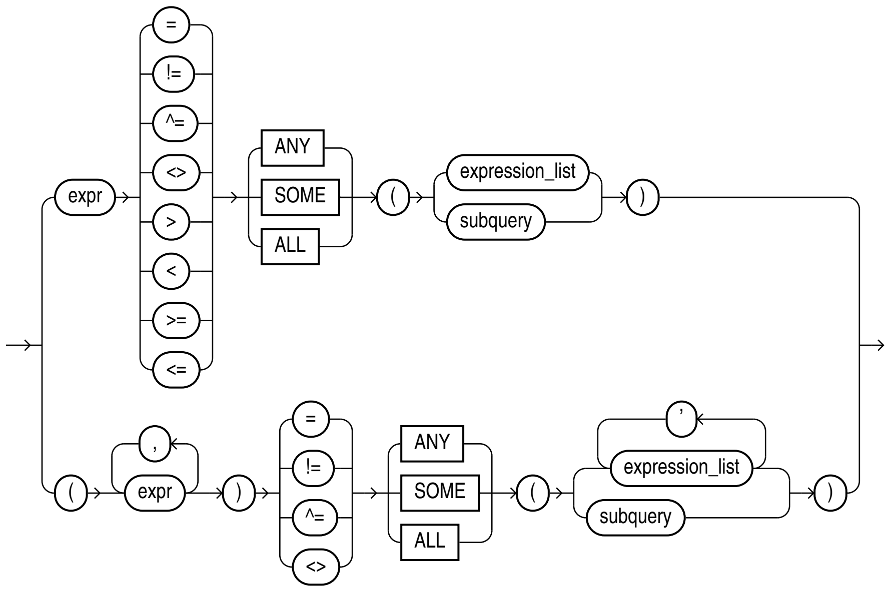
See Also:

Expression Lists for more information about combining expressions and SELECT for information about subqueries

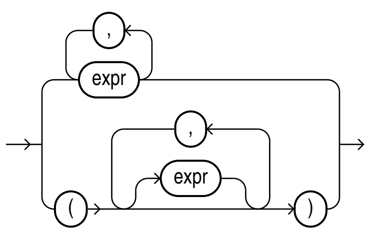
Group Comparison Conditions

A group comparison condition specifies a comparison with any or all members in a list or subquery.

***group_comparison_condition*::=**



***expression_list*::=**



If you use the upper form of this condition (with a single expression to the left of the operator), then you must use the upper form of `expression_list`. If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of `expression_list`, and the expressions in each `expression_list` must match in number and data type the expressions to the left of the operator. If you specify *`subquery`*, then the values returned by the subquery must match in number and data type the expressions to the left of the operator.

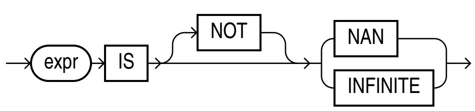
See Also:

- Expression Lists
- SELECT
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for comparison conditions

Floating-Point Conditions

The floating-point conditions let you determine whether an expression is infinite or is the undefined result of an operation (is not a number or NaN).

***floating_point_condition*::=**



In both forms of floating-point condition, *expr* must resolve to a numeric data type or to any data type that can be implicitly converted to a numeric data type. Table 6-3 describes the floating-point conditions.

Table 6-3 Floating-Point Conditions

Type of Condition	Operation	Example
IS [NOT] NAN	Returns TRUE if *expr* is the special value NaN when NOT is not specified. Returns TRUE if *expr* is not the special value NaN when NOT is specified.	`SELECT COUNT(*) FROM employees WHERE commission_pct IS NOT NAN;`
IS [NOT] INFINITE	Returns TRUE if *expr* is the special value +INF or -INF when NOT is not specified. Returns TRUE if *expr* is neither +INF nor -INF when NOT is specified.	`SELECT last_name FROM employees WHERE salary IS NOT INFINITE;`

See Also:

- Floating-Point Numbers for more information on the Oracle implementation of floating-point numbers
- Implicit Data Conversion for more information on how Oracle converts floating-point data types

Logical Conditions

A logical condition combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 6-4 lists logical conditions.

Table 6-4 Logical Conditions

Type of Condition	Operation	Examples
`NOT`	Returns `TRUE` if the following condition is `FALSE`. Returns `FALSE` if it is `TRUE`. If it is `UNKNOWN`, then it remains `UNKNOWN`.	`SELECT * FROM employees WHERE NOT (job_id IS NULL) ORDER BY employee_id;` `SELECT * FROM employees WHERE NOT (salary BETWEEN 1000 AND 2000) ORDER BY employee_id;`
`AND`	Returns `TRUE` if both component conditions are `TRUE`. Returns `FALSE` if either is `FALSE`. Otherwise returns `UNKNOWN`.	`SELECT * FROM employees WHERE job_id = 'PU_CLERK' AND department_id = 30 ORDER BY employee_id;`
`OR`	Returns `TRUE` if either component condition is `TRUE`. Returns `FALSE` if both are `FALSE`. Otherwise returns `UNKNOWN`.	`SELECT * FROM employees WHERE job_id = 'PU_CLERK' OR department_id = 10 ORDER BY employee_id;`

Table 6-5 shows the result of applying the `NOT` condition to an expression.

Table 6-5 NOT Truth Table

--	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

Table 6-6 shows the results of combining the `AND` condition to two expressions.

Table 6-6 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

For example, in the `WHERE` clause of the following `SELECT` statement, the `AND` logical condition is used to ensure that only those hired before 2004 and earning more than $2500 a month are returned:

```
SELECT * FROM employees
WHERE hire_date < TO_DATE('01-JAN-2004', 'DD-MON-YYYY')
  AND salary > 2500
  ORDER BY employee_id;
```

Table 6-7 shows the results of applying `OR` to two expressions.

Table 6-7 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

For example, the following query returns employees who have a 40% commission rate or a salary greater than $20,000:

```
SELECT employee_id FROM employees
   WHERE commission_pct = .4 OR salary > 20000
   ORDER BY employee_id;
```

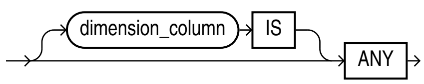
Model Conditions

Model conditions can be used only in the `MODEL` clause of a `SELECT` statement.

IS ANY Condition

The `IS ANY` condition can be used only in the *`model_clause`* of a `SELECT` statement. Use this condition to qualify all values of a dimension column, including `NULL`.

***is_any_condition*::=**



The condition always returns a Boolean value of `TRUE` in order to qualify all values of the column.

See Also:

model_clause and Model Expressions for information

Example

The following example sets sales for each product for year 2000 to 0:

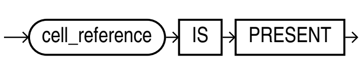
```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[ANY, 2000] = 0
    )
  ORDER BY country, prod, year;

COUNTRY        PROD                                    YEAR          S
----------     -----------------------------------  --------- ---------
France         Mouse Pad                               1998   2509.42
France         Mouse Pad                               1999   3678.69
France         Mouse Pad                               2000         0
France         Mouse Pad                               2001   3269.09
France         Standard Mouse                          1998   2390.83
France         Standard Mouse                          1999   2280.45
France         Standard Mouse                          2000         0
France         Standard Mouse                          2001   2164.54
Germany        Mouse Pad                               1998   5827.87
Germany        Mouse Pad                               1999   8346.44
Germany        Mouse Pad                               2000         0
Germany        Mouse Pad                               2001   9535.08
Germany        Standard Mouse                          1998   7116.11
Germany        Standard Mouse                          1999   6263.14
Germany        Standard Mouse                          2000         0
Germany        Standard Mouse                          2001   6456.13

16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to The MODEL clause: Examples to create this view.

IS PRESENT Condition

***is_present_condition*::=**

The `IS PRESENT` condition can be used only in the *`model_clause`* of a `SELECT` statement. Use this condition to test whether the cell referenced is present prior to the execution of the *`model_clause`*.



The condition returns `TRUE` if the cell exists prior to the execution of the *`model_clause`* and `FALSE` if it does not.

See Also:

model_clause and Model Expressions for information

Example

In the following example, if sales of the Mouse Pad for year 1999 exist, then sales of the Mouse Pad for year 2000 is set to sales of the Mouse Pad for year 1999. Otherwise, sales of the Mouse Pad for year 2000 is set to 0.

```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s['Mouse Pad', 2000] =
        CASE WHEN s['Mouse Pad', 1999] IS PRESENT
             THEN s['Mouse Pad', 1999]
             ELSE 0
        END
    )
  ORDER BY country, prod, year;
```

```
COUNTRY       PROD                                     YEAR          S
----------    -----------------------------------  --------  ---------
France        Mouse Pad                                1998    2509.42
France        Mouse Pad                                1999    3678.69
France        Mouse Pad                                2000    3678.69
France        Mouse Pad                                2001    3269.09
France        Standard Mouse                           1998    2390.83
France        Standard Mouse                           1999    2280.45
France        Standard Mouse                           2000    1274.31
France        Standard Mouse                           2001    2164.54
Germany       Mouse Pad                                1998    5827.87
Germany       Mouse Pad                                1999    8346.44
Germany       Mouse Pad                                2000    8346.44
Germany       Mouse Pad                                2001    9535.08
Germany       Standard Mouse                           1998    7116.11
Germany       Standard Mouse                           1999    6263.14
Germany       Standard Mouse                           2000    2637.31
Germany       Standard Mouse                           2001    6456.13
16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to The MODEL clause: Examples to create this view.

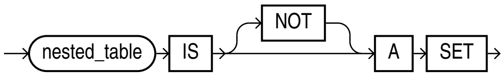
Multiset Conditions

Multiset conditions test various aspects of nested tables.

IS A SET Condition

Use `IS A SET` conditions to test whether a specified nested table is composed of unique elements. The condition returns `UNKNOWN` if the nested table is `NULL`. Otherwise, it returns `TRUE` if the nested table is a set, even if it is a nested table of length zero, and `FALSE` otherwise.

***is_a_set_condition*::=**



Example

The following example selects from the table `customers_demo` those rows in which the `cust_address_ntab` nested table column contains unique elements:

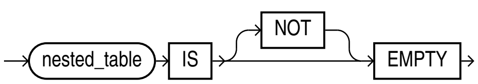
```
SELECT customer_id, cust_address_ntab
  FROM customers_demo
  WHERE cust_address_ntab IS A SET
  ORDER BY customer_id;

CUSTOMER_ID CUST_ADDRESS_NTAB(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
-----------------------------------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'))
        102 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'))
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
        104 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'))
        105 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
```

The preceding example requires the table `customers_demo` and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table column.

IS EMPTY Condition

Use the `IS [NOT] EMPTY` conditions to test whether a specified nested table is empty. A nested table that consists of a single value, a `NULL`, is not considered an empty nested table.

***is_empty_condition*::=**



The condition returns a Boolean value: `TRUE` for an `IS EMPTY` condition if the collection is empty, and `TRUE` for an `IS NOT EMPTY` condition if the collection is not empty. If you specify `NULL` for the nested table or varray, then the result is `NULL`.

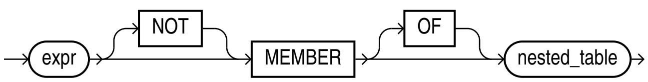
Example

The following example selects from the sample table `pm.print_media` those rows in which the `ad_textdocs_ntab` nested table column is not empty:

```
SELECT product_id, TO_CHAR(ad_finaltext) AS text
   FROM print_media
   WHERE ad_textdocs_ntab IS NOT EMPTY
   ORDER BY product_id, text;
```

MEMBER Condition

***member_condition*::=**



A *member_condition* is a membership condition that tests whether an element is a member of a nested table. The return value is TRUE if *expr* is equal to a member of the specified nested table or varray. The return value is NULL if *expr* is null or if the nested table is empty.

- *expr* must be of the same type as the element type of the nested table.
- The OF keyword is optional and does not change the behavior of the condition.
- The NOT keyword reverses the Boolean output: Oracle returns FALSE if *expr* is a member of the specified nested table.
- The element types of the nested table must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example selects from the table customers_demo those rows in which the cust_address_ntab nested table column contains the values specified in the WHERE clause:

```
SELECT customer_id, cust_address_ntab
  FROM customers_demo
  WHERE cust_address_typ('8768 N State Rd 37', 47404,
                         'Bloomington', 'IN', 'US')
  MEMBER OF cust_address_ntab
  ORDER BY customer_id;

CUSTOMER_ID CUST_ADDRESS_NTAB(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
------------ ---------------------------------------------------------------------------------
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
```

The preceding example requires the table customers_demo and a nested table column containing data. Refer to Multiset Operators to create this table and nested table column.

SUBMULTISET Condition

The SUBMULTISET condition tests whether a specified nested table is a submultiset of another specified nested table.

The operator returns a Boolean value. TRUE is returned when *nested_table1* is a submultiset of *nested_table2*. *nested_table1* is a submultiset of *nested_table2* when one of the following conditions occur:

- *nested_table1* is not null and contains no rows. TRUE is returned even if *nested_table2* is null since an empty multiset is a submultiset of any non-null replacement for *nested_table2*.

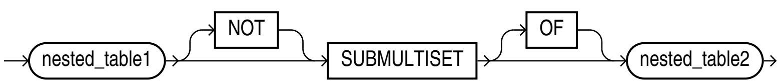
- *nested_table1* and *nested_table2* are not null, *nested_table1* does not contain a null element, and there is a one-to-one mapping of each element in *nested_table1* to an equal element in *nested_table2*.

NULL is returned when one of the following conditions occurs:

- *nested_table1* is null.
- *nested_table2* is null, and *nested_table1* is not null and not empty.
- *nested_table1* is a submultiset of *nested_table2* after modifying each null element of *nested_table1* and *nested_table2* to some non-null value, enabling a one-to-one mapping of each element in *nested_table1* to an equal element in *nested_table2*.

If none of the above conditions occur, then FALSE is returned.

***submultiset_condition*::=**



- The OF keyword is optional and does not change the behavior of the operator.
- The NOT keyword reverses the Boolean output: Oracle returns FALSE if *nested_table1* is a subset of *nested_table2*.
- The element types of the nested table must be comparable. Refer to Comparison Conditions for information on the comparability of nonscalar types.

Example

The following example selects from the customers_demo table those rows in which the cust_address_ntab nested table is a submultiset of the cust_address2_ntab nested table:

```
SELECT customer_id, cust_address_ntab
  FROM customers_demo
  WHERE cust_address_ntab SUBMULTISET OF cust_address2_ntab
  ORDER BY customer_id;
```

The preceding example requires the table customers_demo and two nested table columns containing data. Refer to Multiset Operators to create this table and nested table columns.

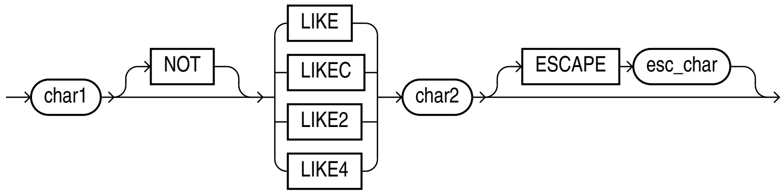
Pattern-matching Conditions

The pattern-matching conditions compare character data.

LIKE Condition

The LIKE conditions specify a test involving pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the LIKE conditions match a portion of one character value to another by searching the first value for the pattern specified by the second. LIKE calculates strings using characters as defined by the input character set. LIKEC uses Unicode complete characters. LIKE2 uses UCS2 code points. LIKE4 uses UCS4 code points.

***like_condition*::=**



In this syntax:

- *char1* is a character expression, such as a character column, called the **search value**.
- *char2* is a character expression, usually a literal, called the **pattern**.
- *esc_char* is a character expression, usually a literal, called the **escape character**.

The LIKE condition is the best choice in almost all situations. Use the following guidelines to determine whether any of the variations would be helpful in your environment:

- Use LIKE2 to process strings using UCS-2 semantics. LIKE2 treats a Unicode supplementary character as two characters.
- Use LIKE4 to process strings using UCS-4 semantics. LIKE4 treats a Unicode supplementary character as one character.
- Use LIKEC to process strings using Unicode complete character semantics. LIKEC treats a composite character as one character.

If *esc_char* is not specified, then there is no default escape character. If any of *char1*, *char2*, or *esc_char* is null, then the result is unknown. Otherwise, the escape character, if specified, must be a character string of length 1.

All of the character expressions (*char1*, *char2*, and *esc_char*) can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. If they differ, then Oracle converts all of them to the data type of *char1*.

The pattern can contain special pattern-matching characters:

- An underscore (_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
- A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. The pattern '%' cannot match a null.

You can include the actual characters % or _ in the pattern by using the ESCAPE clause, which identifies the escape character. If the escape character precedes the character % or _ in the pattern, then Oracle interprets this character literally in the pattern rather than as a special pattern-matching character. You can also search for the escape character itself by repeating it. For example, if @ is the escape character, then you can use @@ to search for @.

> **Note:**
>
> Only ASCII-equivalent underscore (_) and percent (%) characters are recognized as pattern-matching characters. Their full-width variants, present in East Asian character sets and in Unicode, are treated as normal characters.

Table 6-8 describes the `LIKE` conditions.

Table 6-8 LIKE Condition

Type of Condition	Operation	Example
`x [NOT] LIKE y [ESCAPE 'z']`	`TRUE` if *x* does [not] match the pattern *y*. Within *y*, the character `%` matches any string of zero or more characters except null. The character _ matches any single character. Any character can follow `ESCAPE` except percent (%) and underbar (_). A wildcard character is treated as a literal if preceded by the escape character.	`SELECT last_name FROM employees WHERE last_name LIKE '%A\_B%' ESCAPE '\' ORDER BY last_name;`

To process the `LIKE` conditions, Oracle divides the pattern into subpatterns consisting of one or two characters each. The two-character subpatterns begin with the escape character and the other character is %, or _, or the escape character.

Let P_1, P_2, ..., P_n be these subpatterns. The like condition is true if there is a way to partition the search value into substrings S_1, S_2, ..., S_n so that for all *i* between 1 and *n*:

- If P_i is _, then S_i is a single character.
- If P_i is %, then S_i is any string.
- If P_i is two characters beginning with an escape character, then S_i is the second character of P_i.
- Otherwise, $P_i = S_i$.

With the `LIKE` conditions, you can compare a value to a pattern rather than to a constant. The pattern must appear after the `LIKE` keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with `R`:

```
SELECT salary
    FROM employees
    WHERE last_name LIKE 'R%'
    ORDER BY salary;
```

The following query uses the = operator, rather than the `LIKE` condition, to find the salaries of all employees with the name 'R%':

```
SELECT salary
    FROM employees
    WHERE last_name = 'R%'
    ORDER BY salary;
```

The following query finds the salaries of all employees with the name 'SM%'. Oracle interprets 'SM%' as a text literal, rather than as a pattern, because it precedes the `LIKE` keyword:

```
SELECT salary
    FROM employees
    WHERE 'SM%' LIKE last_name
    ORDER BY salary;
```

Collation and Case Sensitivity

The `LIKE` condition is collation-sensitive. Oracle Database compares the subpattern Pi to the substring Si in the processing algorithm above using the collation determined from the derived collations of *`char1`* and *`char2`*. If this collation is case-insensitive, the pattern-matching is case-insensitive as well.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on case- and accent-insensitive collations and on collation determination rules for the `LIKE` condition

Pattern Matching on Indexed Columns

When you use `LIKE` to search an indexed column for a pattern, Oracle can use the index to improve performance of a query if the leading character in the pattern is not `%` or _. In this case, Oracle can scan the index by this leading character. If the first character in the pattern is `%` or _, then the index cannot improve performance because Oracle cannot scan the index.

LIKE Condition: General Examples

This condition is true for all `last_name` values beginning with `Ma`:

```
last_name LIKE 'Ma%'
```

All of these `last_name` values make the condition true:

```
Mallin, Markle, Marlow, Marvins, Mavris, Matos
```

Case is significant, so `last_name` values beginning with `MA`, `ma`, and `mA` make the condition false.

Consider this condition:

```
last_name LIKE 'SMITH_'
```

This condition is true for these `last_name` values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for `SMITH` because the special underscore character (_) must match exactly one character of the `last_name` value.

ESCAPE Clause Example

The following example searches for employees with the pattern `A_B` in their name:

```
SELECT last_name
    FROM employees
    WHERE last_name LIKE '%A\_B%' ESCAPE '\'
    ORDER BY last_name;
```

The `ESCAPE` clause identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

Patterns Without % Example

If a pattern does not contain the `%` character, then the condition can be true only if both operands have the same length. Consider the definition of this table and the values inserted into it:

```
CREATE TABLE ducks (f CHAR(6), v VARCHAR2(6));
INSERT INTO ducks VALUES ('DUCK', 'DUCK');
SELECT '*'||f||'*' "char",
   '*'||v||'*' "varchar"
   FROM ducks;

char     varchar
-------- --------
*DUCK  * *DUCK*
```

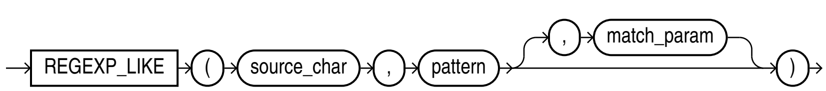
Because Oracle blank-pads `CHAR` values, the value of `f` is blank-padded to 6 bytes. `v` is not blank-padded and has length 4.

REGEXP_LIKE Condition

`REGEXP_LIKE` is similar to the `LIKE` condition, except `REGEXP_LIKE` performs regular expression matching instead of the simple pattern matching performed by `LIKE`. This condition evaluates strings using characters as defined by the input character set.

This condition complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

***regexp_like_condition*::=**



- *`source_char`* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.
- *`pattern`* is the **regular expression**. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *`pattern`* is different from the data type of *`source_char`*, Oracle converts

pattern to the data type of *source_char*. For a listing of the operators you can specify in *pattern*, refer to Oracle Regular Expression Support.

- *match_param* is a character expression of the data type VARCHAR2 or CHAR that lets you change the default matching behavior of the condition.

  The value of *match_param* can include one or more of the following characters:

  – 'i' specifies case-insensitive matching, even if the determined collation of the condition is case-sensitive.

  – 'c' specifies case-sensitive and accent-sensitive matching, even if the determined collation of the condition is case-insensitive or accent-insensitive.

  – 'n' allows the period (.), which is the match-any-character wildcard character, to match the newline character. If you omit this parameter, then the period does not match the newline character.

  – 'm' treats the source string as multiple lines. Oracle interprets ^ and $ as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, then Oracle treats the source string as a single line.

  – 'x' ignores whitespace characters. By default, whitespace characters match themselves.

  If the value of *match_param* contains multiple contradictory characters, then Oracle uses the last character. For example, if you specify 'ic', then Oracle uses case-sensitive and accent-sensitive matching. If the value contains a character other than those shown above, then Oracle returns an error.

  If you omit *match_param*, then:

  – The default case and accent sensitivity are determined by the determined collation of the REGEXP_LIKE condition.

  – A period (.) does not match the newline character.

  – The source string is treated as a single line.

Similar to the LIKE condition, the REGEXP_LIKE condition is collation-sensitive.

See Also:

- LIKE Condition
- REGEXP_INSTR , REGEXP_REPLACE , and REGEXP_SUBSTR for functions that provide regular expression support
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the REGEXP_LIKE condition

Examples

The following query returns the first and last names for those employees with a first name of Steven or Stephen (where first_name begins with Ste and ends with en and in between is either v or ph):

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$')
ORDER BY first_name, last_name;

FIRST_NAME           LAST_NAME
-------------------- -------------------------
Steven               King
Steven               Markle
Stephen              Stiles
```

The following query returns the last name for those employees with a double vowel in their last name (where `last_name` contains two adjacent occurrences of either `a`, `e`, `i`, `o`, or `u`, regardless of case):

```
SELECT last_name
FROM employees
WHERE REGEXP_LIKE (last_name, '([aeiou])\1', 'i')
ORDER BY last_name;

LAST_NAME
-------------------------
De Haan
Greenberg
Khoo
Gee
Greene
Lee
Bloom
Feeney
```

Null Conditions

A `NULL` condition tests for nulls. This is the only condition that you should use to test for nulls.

***null_condition*::=**

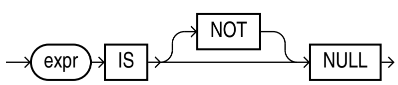


Table 6-9 lists the null conditions.

Table 6-9 Null Condition

Type of Condition	Operation	Example
`IS [NOT] NULL`	Tests for nulls. **See Also:** Nulls	`SELECT last_name FROM employees WHERE commission_pct IS NULL ORDER BY last_name;`

XML Conditions

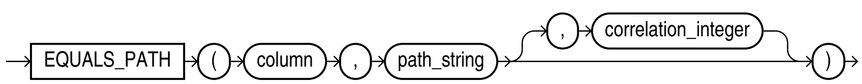
XML conditions determine whether a specified XML resource can be found in a specified path.

EQUALS_PATH Condition

The EQUALS_PATH condition determines whether a resource in the Oracle XML database can be found in the database at a specified path.

Use this condition in queries to RESOURCE_VIEW and PATH_VIEW. These public views provide a mechanism for SQL access to data stored in the XML database repository. RESOURCE_VIEW contains one row for each resource in the repository, and PATH_VIEW contains one row for each unique path in the repository.

equals_path_condition::=



This condition applies only to the path as specified. It is similar to but more restrictive than UNDER_PATH.

For *path_string*, specify the (absolute) path name to resolve. This can contain components that are hard or weak resource links.

The optional *correlation_integer* argument correlates the EQUALS_PATH condition with its ancillary functions DEPTH and PATH.

> **See Also:**
>
> UNDER_PATH Condition , DEPTH , and PATH

Example

The view RESOURCE_VIEW computes the paths (in the any_path column) that lead to all XML resources (in the res column) in the database repository. The following example queries the RESOURCE_VIEW view to find the paths to the resources in the sample schema oe. The EQUALS_PATH condition causes the query to return only the specified path:

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE EQUALS_PATH(res, '/sys/schemas/OE/www.example.com')=1;

ANY_PATH
----------------------------------------------
/sys/schemas/OE/www.example.com
```

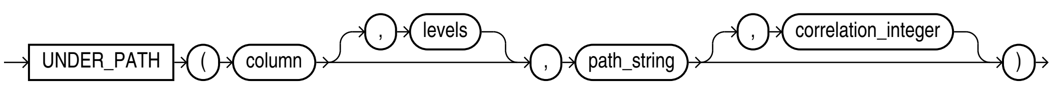
Compare this example with that for UNDER_PATH Condition .

UNDER_PATH Condition

The UNDER_PATH condition determines whether resources specified in a column can be found under a particular path specified by *path_string* in the Oracle XML database repository. The path information is computed by the RESOURCE_VIEW view, which you query to use this condition.

Use this condition in queries to RESOURCE_VIEW and PATH_VIEW. These public views provide a mechanism for SQL access to data stored in the XML database repository. RESOURCE_VIEW contains one row for each resource in the repository, and PATH_VIEW contains one row for each unique path in the repository.

under_path_condition::=



The optional *levels* argument indicates the number of levels down from *path_string* Oracle should search. For *levels*, specify any nonnegative integer.

The optional *correlation_integer* argument correlates the UNDER_PATH condition with its ancillary functions PATH and DEPTH.

> **See Also:**
>
> The related condition EQUALS_PATH Condition and the ancillary functions DEPTH and PATH

Example

The view RESOURCE_VIEW computes the paths (in the any_path column) that lead to all XML resources (in the res column) in the database repository. The following example queries the RESOURCE_VIEW view to find the paths to the resources in the sample schema oe. The query returns the path of the XML schema that was created in XMLType Table Examples:

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE UNDER_PATH(res, '/sys/schemas/OE/www.example.com')=1;

ANY_PATH
----------------------------------------------
/sys/schemas/OE/www.example.com/xwarehouses.xsd
```

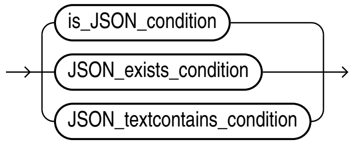
SQL/JSON Conditions

SQL/JSON conditions allow you to test JavaScript Object Notation (JSON) data as follows:

- IS JSON Condition lets you test whether an expression is syntactically correct JSON data
- JSON_EXISTS Condition lets you test whether a specified JSON value exists in JSON data

- JSON_TEXTCONTAINS Condition lets you test whether a specified character string exists in JSON property values.

***JSON_condition*::=**

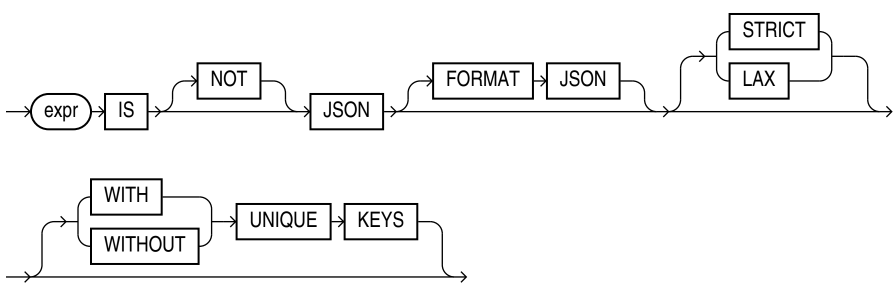


IS JSON Condition

Use this SQL/JSON condition to test whether an expression is syntactically correct, or well-formed, JSON data.

- If you specify `IS JSON`, then this condition returns `TRUE` if the expression is well-formed JSON data and `FALSE` if the expression is not well-formed JSON data.
- If you specify `IS NOT JSON`, then this condition returns `TRUE` if the expression is not well-formed JSON data and `FALSE` if the expression is well-formed JSON data.

***is_JSON_condition*::=**



- Use *`expr`* to specify the JSON data to be evaluated. Specify an expression that evaluates to a text literal. If *`expr`* is a column, then the column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. If *`expr`* evaluates to null or a text literal of length zero, then this condition returns `UNKNOWN`.
- You must specify `FORMAT JSON` if *`expr`* is a column of data type `BLOB`.
- If you specify `STRICT`, then this condition considers only strict JSON syntax to be well-formed JSON data. If you specify `LAX`, then this condition also considers lax JSON syntax to be well-formed JSON data. The default is `LAX`. Refer to *Oracle Database JSON Developer's Guide* for more information on strict and lax JSON syntax.
- If you specify `WITH UNIQUE KEYS`, then this condition considers JSON data to be well-formed only if key names are unique within each object. If you specify `WITHOUT UNIQUE KEYS`, then this condition considers JSON data to be well-formed even if duplicate key names occur within an object. A `WITHOUT UNIQUE KEYS` test performs faster than a `WITH UNIQUE KEYS` test. The default is `WITHOUT UNIQUE KEYS`.

Examples

Testing for STRICT or LAX JSON Syntax: Example

The following statement creates table `t` with column `col1`:

```
CREATE TABLE t (col1 VARCHAR2(100));
```

The following statements insert values into column `col1` of table `t`:

```
INSERT INTO t VALUES ( '[ "LIT192", "CS141", "HIS160" ]' );
INSERT INTO t VALUES ( '{ "Name": "John" }' );
INSERT INTO t VALUES ( '{ "Grade Values" : { A : 4.0, B : 3.0, C : 2.0 } }');
INSERT INTO t VALUES ( '{ "isEnrolled" : true }' );
INSERT INTO t VALUES ( '{ "isMatriculated" : False }' );
INSERT INTO t VALUES (NULL);
INSERT INTO t VALUES ('This is not well-formed JSON data');
```

The following statement queries table `t` and returns `col1` values that are well-formed JSON data. Because neither the `STRICT` nor `LAX` keyword is specified, this example uses the default `LAX` setting. Therefore, this query returns values that use strict or lax JSON syntax.

```
SELECT col1
  FROM t
  WHERE col1 IS JSON;

COL1
--------------------------------------------------
[ "LIT192", "CS141", "HIS160" ]
{ "Name": "John" }
{ "Grade Values" : { A : 4.0, B : 3.0, C : 2.0 } }
{ "isEnrolled" : true }
{ "isMatriculated" : False }
```

The following statement queries table `t` and returns `col1` values that are well-formed JSON data. This example specifies the `STRICT` setting. Therefore, this query returns only values that use strict JSON syntax.

```
SELECT col1
  FROM t
  WHERE col1 IS JSON STRICT;

COL1
--------------------------------------------------
[ "LIT192", "CS141", "HIS160" ]
{ "Name": "John" }
{ "isEnrolled" : true }
```

The following statement queries table `t` and returns `col1` values that use lax JSON syntax, but omits `col1` values that use strict JSON syntax. Therefore, this query returns only values that contain the exceptions allowed in lax JSON syntax.

```
SELECT col1
  FROM t
  WHERE col1 IS NOT JSON STRICT AND col1 IS JSON LAX;

COL1
--------------------------------------------------
```

```
{ "Grade Values" : { A : 4.0, B : 3.0, C : 2.0 } }
{ "isMatriculated" : False }
```

Testing for Unique Keys: Example

The following statement creates table `t` with column `col1`:

```
CREATE TABLE t (col1 VARCHAR2(100));
```

The following statements insert values into column `col1` of table `t`:

```
INSERT INTO t VALUES ('{a:100, b:200, c:300}');
INSERT INTO t VALUES ('{a:100, a:200, b:300}');
INSERT INTO t VALUES ('{a:100, b : {a:100, c:300}}');
```

The following statement queries table t and returns `col1` values that are well-formed JSON data with unique key names within each object:

```
SELECT col1 FROM t
  WHERE col1 IS JSON WITH UNIQUE KEYS;

COL1
---------------------------
{a:100, b:200, c:300}
{a:100, b : {a:100, c:300}}
```

The second row is returned because, while the key name `a` appears twice, it is in two different objects.

The following statement queries table `t` and returns `col1` values that are well-formed JSON data, regardless of whether there are unique key names within each object:

```
SELECT col1 FROM t
  WHERE col1 IS JSON WITHOUT UNIQUE KEYS;

COL1
---------------------------
{a:100, b:200, c:300}
{a:100, a:200, b:300}
{a:100, b : {a:100, c:300}}
```

Using IS JSON as a Check Constraint: Example

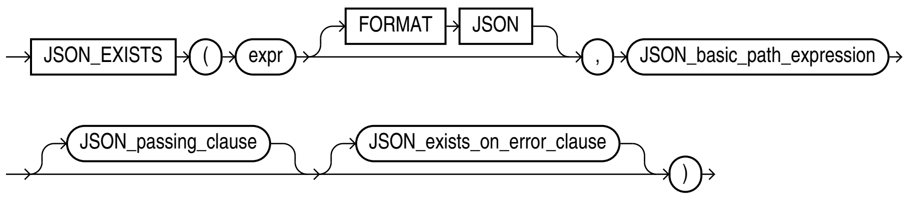
The following statement creates table `j_purchaseorder`, which will store JSON data in column `po_document`. The statement uses the `IS JSON` condition as a check constraint to ensure that only well-formed JSON is stored in column `po_document`.

```
CREATE TABLE j_purchaseorder
  (id RAW (16) NOT NULL,
   date_loaded TIMESTAMP(6) WITH TIME ZONE,
   po_document CLOB CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

JSON_EXISTS Condition

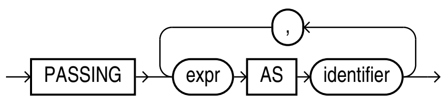
Use the SQL/JSON condition `JSON_EXISTS` to test whether a specified JSON value exists in JSON data. This condition returns `TRUE` if the JSON value exists and `FALSE` if the JSON value does not exist.

***JSON_exists_condition*::=**

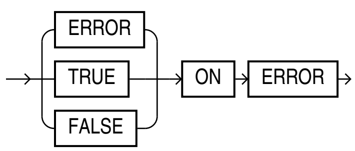


(*`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

***JSON_passing_clause*::=**



***JSON_exists_on_error_clause*::=**



expr

Use this clause to specify the JSON data to be evaluated. For *`expr`*, specify an expression that evaluates to a text literal. If *`expr`* is a column, then the column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. If *`expr`* evaluates to null or a text literal of length zero, then the condition returns `UNKNOWN`.

If *`expr`* is not a text literal of well-formed JSON data using strict or lax syntax, then the condition returns `FALSE` by default. You can use the *`JSON_exists_on_error_clause`* to override this default behavior. Refer to the *JSON_exists_on_error_clause*.

FORMAT JSON

You must specify `FORMAT JSON` if *`expr`* is a column of data type `BLOB`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The condition uses the path expression to evaluate *`expr`* and determine if a JSON value that matches, or satisfies, the path expression exists. The path expression must be a text literal, but it can contain variables whose values are passed to the path expression by the *`JSON_passing_clause`*. See *Oracle Database JSON Developer's Guide* for the full semantics of *`JSON_basic_path_expression`*.

JSON_passing_clause

Use this clause to pass values to the path expression. For *`expr`*, specify a value of data type `VARCHAR2`, `NUMBER`, `BINARY_DOUBLE`, `DATE`, `TIMESTAMP`, or `TIMESTAMP WITH TIME ZONE`. The result of evaluating *`expr`* is bound to the corresponding identifier in the *`JSON_basic_path_expression`*.

JSON_exists_on_error_clause

Use this clause to specify the value returned by this condition when *`expr`* is not well-formed JSON data.

You can specify the following clauses:

- `ERROR ON ERROR` - Returns the appropriate Oracle error when *`expr`* is not well-formed JSON data.
- `TRUE ON ERROR` - Returns `TRUE` when *`expr`* is not well-formed JSON data.
- `FALSE ON ERROR` - Returns `FALSE` when *`expr`* is not well-formed JSON data. This is the default.

Examples

The following statement creates table `t` with column `name`:

```
CREATE TABLE t (name VARCHAR2(100));
```

The following statements insert values into column `name` of table `t`:

```
INSERT INTO t VALUES ('[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]');
INSERT INTO t VALUES ('[{first:"Mary"}, {last:"Jones"}]');
INSERT INTO t VALUES ('[{first:"Jeff"}, {last:"Williams"}]');
INSERT INTO t VALUES ('[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]');
INSERT INTO t VALUES (NULL);
INSERT INTO t VALUES ('This is not well-formed JSON data');
```

The following statement queries column `name` in table `t` and returns JSON data that consists of an array whose first element is an object with property name `first`. The `ON ERROR` clause is not specified. Therefore, the `JSON_EXISTS` condition returns `FALSE` for values that are not well-formed JSON data.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[0].first');

NAME
--------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Mary"}, {last:"Jones"}]
[{first:"Jeff"}, {last:"Williams"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

The following statement queries column `name` in table `t` and returns JSON data that consists of an array whose second element is an object with property name `middle`. The `ON ERROR` clause is not specified. Therefore, the `JSON_EXISTS` condition returns `FALSE` for values that are not well-formed JSON data.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[1].middle');
```

```
NAME
--------------------------------------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

The following statement is similar to the previous statement, except that the `TRUE ON ERROR` clause is specified. Therefore, the `JSON_EXISTS` condition returns `TRUE` for values that are not well-formed JSON data.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[1].middle' TRUE ON ERROR);

NAME
--------------------------------------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
This is not well-formed JSON data
```

The following statement queries column `name` in table `t` and returns JSON data that consists of an array that contains an element that is an object with property name `last`. The wildcard symbol (`*`) is specified for the array index. Therefore, the query returns arrays that contain such an object, regardless of its index number in the array.

```
SELECT name FROM t
  WHERE JSON_EXISTS(name, '$[*].last');

NAME
--------------------------------------------------
[{first:"John"}, {middle:"Mark"}, {last:"Smith"}]
[{first:"Mary"}, {last:"Jones"}]
[{first:"Jeff"}, {last:"Williams"}]
[{first:"Jean"}, {middle:"Anne"}, {last:"Brown"}]
```

JSON_TEXTCONTAINS Condition

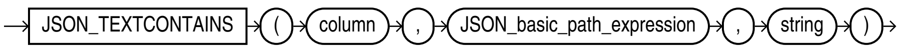
Use the SQL/JSON condition `JSON_TEXTCONTAINS` to test whether a specified character string exists in JSON property values. You can use this condition to filter JSON data on a specific word or number.

This condition takes the following arguments:

- A table or view column that contains JSON data. A JSON search index, which is an Oracle Text index designed specifically for use with JSON data, must be defined on the column. Each row of JSON data in the column is referred to as a **JSON document**.
- A SQL/JSON path expression. The path expression is applied to each JSON document in an attempt to match a specific JSON object within the document. The path expression can contain only JSON object steps; it cannot contain JSON array steps.
- A character string. The condition searches for the character string in all of the string and numeric property values in the matched JSON object, including array values. The string must exist as a separate word in the property value. For example, if you search for 'beth', then a match will be found for string property value "beth smith", but not for "elizabeth smith". If you search for '10', then a match will be found for numeric property value 10 or string property value "10 main street", but a match will not be found for numeric property value 110 or string property value "102 main street".

This condition returns `TRUE` if a match is found, and `FALSE` if a match is not found.

***JSON_textcontains_condition*::=**



(*`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

column

Specify the name of the table or view column containing the JSON data to be tested. The column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. A JSON search index, which is an Oracle Text index designed specifically for use with JSON data, must be defined on the column. If a column value is a null or a text literal of length zero, then the condition returns `UNKNOWN`.

If a column value is not a text literal of well-formed JSON data using strict or lax syntax, then the condition returns `FALSE`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The condition uses the path expression to evaluate *`column`* and determine if a JSON value that matches, or satisfies, the path expression exists. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of *`JSON_basic_path_expression`*.

string

The condition searches for the character string specified by *`string`*. The string must be enclosed in single quotation marks.

Examples

The following statement creates table `families` with column `family_doc`:

```
CREATE TABLE families (family_doc VARCHAR2(200));
```

The following statement creates a JSON search index on column `family_doc`:

```
CREATE INDEX ix
  ON families(family_doc)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS ('SECTION GROUP CTXSYS.JSON_SECTION_GROUP SYNC (ON COMMIT)');
```

The following statements insert JSON documents that describe families into column `family_doc`:

```
INSERT INTO families
VALUES ('{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}');

INSERT INTO families
VALUES ('{family : {id:11, ages:[42,40,10,5], address : {street : "200 East Street", apt :
20}}}');

INSERT INTO families
VALUES ('{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}');
```

The following statement commits the transaction:

```
COMMIT;
```

The following query returns the JSON documents that contain `10` in any property value in the document:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}
{family : {id:11, ages:[42,40,10,5], address : {street : "200 East Street", apt : 20}}}
{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}
```

The following query returns the JSON documents that contain 10 in the `id` property value:

```
SELECT family_doc FROM families
  where json_textcontains(family_doc, '$.family.id', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}
```

The following query returns the JSON documents that have a 10 in the array of values for the `ages` property:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$.family.ages', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:11, ages:[42,40,10,5], address : {street : "200 East Street", apt : 20}}}
```

The following query returns the JSON documents that have a 10 in the `address` property value:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$.family.address', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:10, ages:[40,38,12], address : {street : "10 Main Street"}}}
{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}
```

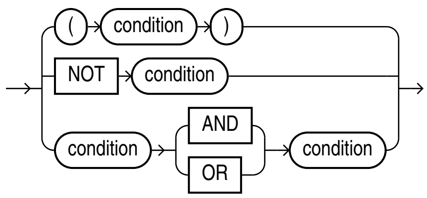
The following query returns the JSON documents that have a 10 in the `apt` property value:

```
SELECT family_doc FROM families
  WHERE JSON_TEXTCONTAINS(family_doc, '$.family.address.apt', '10');

FAMILY_DOC
--------------------------------------------------------------------------------
{family : {id:12, ages:[25,23], address : {street : "300 Oak Street", apt : 10}}}
```

Compound Conditions

A compound condition specifies a combination of other conditions.

***compound_condition*::=**



See Also:

Logical Conditions for more information about `NOT`, `AND`, and `OR` conditions

BETWEEN Condition

A `BETWEEN` condition determines whether the value of one expression is in an interval defined by two other expressions.

***between_condition*::=**

All three expressions must be numeric, character, or datetime expressions. In SQL, it is possible that *`expr1`* will be evaluated more than once. If the `BETWEEN` expression appears in PL/SQL, *`expr1`* is guaranteed to be evaluated only once. If the expressions are not all the same data type, then Oracle Database implicitly converts the expressions to a common data type. If it cannot do so, then it returns an error.

See Also:

Implicit Data Conversion for more information on SQL data type conversion

The value of

```
expr1 NOT BETWEEN expr2 AND expr3
```

is the value of the expression

```
NOT (expr1 BETWEEN expr2 AND expr3)
```

And the value of

```
expr1 BETWEEN expr2 AND expr3
```

is the value of the boolean expression:

```
expr2 <= expr1 AND expr1 <= expr3
```

If *expr3* < *expr2*, then the interval is empty. If *expr1* is `NULL`, then the result is `NULL`. If *expr1* is not `NULL`, then the value is `FALSE` in the ordinary case and `TRUE` when the keyword `NOT` is used.

The boolean operator `AND` may produce unexpected results. Specifically, in the expression `x AND y`, the condition `x IS NULL` is not sufficient to determine the value of the expression. The second operand still must be evaluated. The result is `FALSE` if the second operand has the value `FALSE` and `NULL` otherwise. See Logical Conditions for more information on `AND`.

Table 6-10 BETWEEN Condition

Type of Condition	Operation	Example
`[NOT] BETWEEN x AND y`	`[NOT]` (*expr2* less than or equal to *expr1* `AND` *expr1* less than or equal to *expr3*)	`SELECT * FROM employees WHERE salary BETWEEN 2000 AND 3000 ORDER BY employee_id;`

EXISTS Condition

An `EXISTS` condition tests for existence of rows in a subquery.

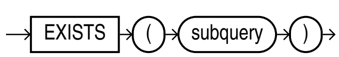


Table 6-11 shows the `EXISTS` condition.

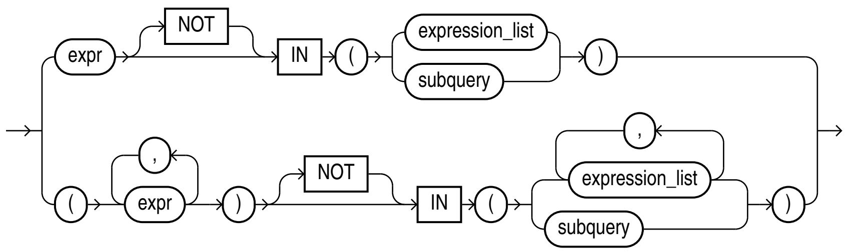
Table 6-11 EXISTS Condition

Type of Condition	Operation	Example
`EXISTS`	`TRUE` if a subquery returns at least one row.	`SELECT department_id FROM departments d WHERE EXISTS (SELECT * FROM employees e WHERE d.department_id = e.department_id) ORDER BY department_id;`

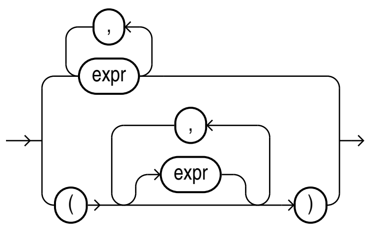
IN Condition

An *`in_condition`* is a membership condition. It tests a value for membership in a list of values or subquery

in_condition::=



expression_list::=



If you use the upper form of the `in_condition` condition (with a single expression to the left of the operator), then you must use the upper form of `expression_list`. If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of `expression_list`, and the expressions in each `expression_list` must match in number and data type the expressions to the left of the operator. You can specify up to 1000 expressions in `expression_list`.

Oracle Database does not always evaluate the expressions in an `expression_list` in the order in which they appear in the `IN` list. However, expressions in the select list of a subquery are evaluated in their specified order.

> **See Also:**
>
> Expression Lists

Table 6-12 lists the form of `IN` condition.

Table 6-12 IN Condition

Type of Condition	Operation	Example
`IN`	Equal-to-any-member-of test. Equivalent to `=ANY`.	`SELECT * FROM employees WHERE job_id IN ('PU_CLERK','SH_CLERK') ORDER BY employee_id;` `SELECT * FROM employees WHERE salary IN (SELECT salary FROM employees WHERE department_id =30) ORDER BY employee_id;`
`NOT IN`	Equivalent to `!=ALL`. Evaluates to `FALSE` if any member of the set is `NULL`.	`SELECT * FROM employees WHERE salary NOT IN (SELECT salary FROM employees WHERE department_id = 30) ORDER BY employee_id;` `SELECT * FROM employees WHERE job_id NOT IN ('PU_CLERK', 'SH_CLERK') ORDER BY employee_id;`

If any item in the list following a `NOT IN` operation evaluates to null, then all rows evaluate to `FALSE` or `UNKNOWN`, and no rows are returned. For example, the following statement returns the string '`True`' for each row:

```
SELECT 'True' FROM employees
   WHERE department_id NOT IN (10, 20);
```

However, the following statement returns no rows:

```
SELECT 'True' FROM employees
    WHERE department_id NOT IN (10, 20, NULL);
```

The preceding example returns no rows because the `WHERE` clause condition evaluates to:

```
department_id != 10 AND department_id != 20 AND department_id != null
```

Because the third condition compares `department_id` with a null, it results in an `UNKNOWN`, so the entire expression results in `FALSE` (for rows with `department_id` equal to 10 or 20). This behavior can easily be overlooked, especially when the `NOT IN` operator references a subquery.

Moreover, if a `NOT IN` condition references a subquery that returns no rows at all, then all rows will be returned, as shown in the following example:

```
SELECT 'True' FROM employees
   WHERE department_id NOT IN (SELECT 0 FROM DUAL WHERE 1=2);
```

For character arguments, the `IN` condition is collation-sensitive. The collation determination rules determine the collation to use.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `IN` condition

Restriction on LEVEL in WHERE Clauses

In a [`NOT`] `IN` condition in a `WHERE` clause, if the right-hand side of the condition is a subquery, you cannot use `LEVEL` on the left-hand side of the condition. However, you can specify `LEVEL` in a subquery of the `FROM` clause to achieve the same result. For example, the following statement is not valid:

```
SELECT employee_id, last_name FROM employees
   WHERE (employee_id, LEVEL)
      IN (SELECT employee_id, 2 FROM employees)
   START WITH employee_id = 2
   CONNECT BY PRIOR employee_id = manager_id;
```

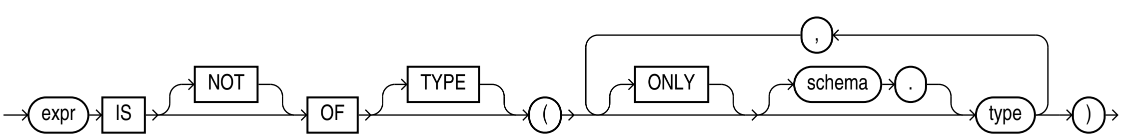
But the following statement is valid because it encapsulates the query containing the `LEVEL` information in the `FROM` clause:

```
SELECT v.employee_id, v.last_name, v.lev FROM
      (SELECT employee_id, last_name, LEVEL lev
      FROM employees v
      START WITH employee_id = 100
      CONNECT BY PRIOR employee_id = manager_id) v
   WHERE (v.employee_id, v.lev) IN
      (SELECT employee_id, 2 FROM employees);
```

IS OF *type* Condition

Use the `IS OF` *`type`* condition to test object instances based on their specific type information.

***is_of_type_condition*::=**



You must have `EXECUTE` privilege on all types referenced by *`type`*, and all *`type`*s must belong to the same type family.

This condition evaluates to null if *`expr`* is null. If *`expr`* is not null, then the condition evaluates to true (or false if you specify the `NOT` keyword) under either of these circumstances:

- The most specific type of *`expr`* is the subtype of one of the types specified in the *`type`* list and you have not specified `ONLY` for the type, or
- The most specific type of *`expr`* is explicitly specified in the *`type`* list.

The *`expr`* frequently takes the form of the `VALUE` function with a correlation variable.

The following example uses the sample table `oe.persons`, which is built on a type hierarchy in Substitutable Table and Column Examples. The example uses the `IS OF` *type* condition to restrict the query to specific subtypes:

```
SELECT * FROM persons p
   WHERE VALUE(p) IS OF TYPE (employee_t);

NAME                         SSN
----------------------------
Joe                        32456
Tim                         5678

SELECT * FROM persons p
   WHERE VALUE(p) IS OF (ONLY part_time_emp_t);

NAME                         SSN
----------------------------
Tim                         5678
```

7
Functions

Functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

A function without any arguments is similar to a pseudocolumn (refer to Pseudocolumns). However, a pseudocolumn typically returns a different value for each row in the result set, whereas a function without any arguments typically returns the same value for each row.

This chapter contains these sections:

- About SQL Functions
- Single-Row Functions
  - Numeric Functions
  - Character Functions Returning Character Values
  - Character Functions Returning Number Values
  - Character Set Functions
  - Collation Functions
  - Datetime Functions
  - General Comparison Functions
  - Conversion Functions
  - Large Object Functions
  - Collection Functions
  - Hierarchical Functions
  - Data Mining Functions
  - XML Functions
  - JSON Functions
  - Encoding and Decoding Functions
  - NULL-Related Functions
  - Environment and Identifier Functions
- Aggregate Functions
- Analytic Functions
- Object Reference Functions
- Model Functions
- OLAP Functions
- Data Cartridge Functions

- About User-Defined Functions

About SQL Functions

SQL functions are built into Oracle Database and are available for use in various appropriate SQL statements. Do not confuse SQL functions with user-defined functions written in PL/SQL.

If you call a SQL function with an argument of a data type other than the data type expected by the SQL function, then Oracle attempts to convert the argument to the expected data type before performing the SQL function.

> **See Also:**
>
> About User-Defined Functions for information on user functions and Data Conversion for implicit conversion of data types

Nulls in SQL Functions

Most scalar functions return null when given a null argument. You can use the `NVL` function to return a value when a null occurs. For example, the expression `NVL(commission_pct,0)` returns 0 if `commission_pct` is null or the value of `commission_pct` if it is not null.

For information on how aggregate functions handle nulls, see Aggregate Functions .

Syntax for SQL Functions

In the syntax diagrams for SQL functions, arguments are indicated by their data types. When the parameter *`function`* appears in SQL syntax, replace it with one of the functions described in this section. Functions are grouped by the data types of their arguments and their return values.

> **Note:**
>
> When you apply SQL functions to LOB columns, Oracle Database creates temporary LOBs during SQL and PL/SQL processing. You should ensure that temporary tablespace quota is sufficient for storing these temporary LOBs for your application.

A SQL function may be collation-sensitive, which means that character value comparison or matching that it performs is controlled by a collation. The particular collation to use by the function is determined from the collations of the function's arguments.

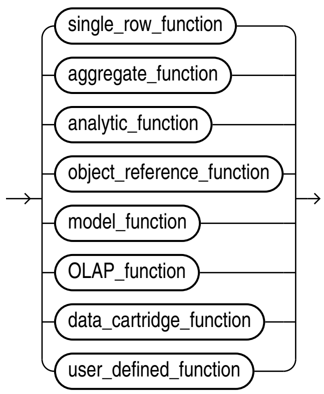
If the result of a SQL function has a character data type, the collation derivation rules define the collation to associate with the result.

See Also:

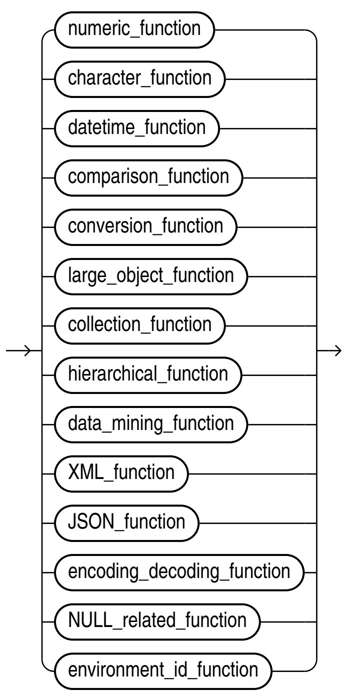
Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation and determination rules for SQL functions

The syntax showing the categories of functions follows:

***function*::=**



***single_row_function*::=**



The sections that follow list the built-in SQL functions in each of the groups illustrated in the preceding diagrams except user-defined functions. All of the built-in SQL functions are then described in alphabetical order.

> **See Also:**
>
> About User-Defined Functions and CREATE FUNCTION

Single-Row Functions

Single-row functions return a single result row for every row of a queried table or view. These functions can appear in select lists, `WHERE` clauses, `START WITH` and `CONNECT BY` clauses, and `HAVING` clauses.

Numeric Functions

Numeric functions accept numeric input and return numeric values. Most numeric functions return `NUMBER` values that are accurate to 38 decimal digits. The transcendental functions `COS`, `COSH`, `EXP`, `LN`, `LOG`, `SIN`, `SINH`, `SQRT`, `TAN`, and `TANH` are accurate to 36 decimal digits. The transcendental functions `ACOS`, `ASIN`, `ATAN`, and `ATAN2` are accurate to 30 decimal digits. The numeric functions are:

- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- BITAND
- CEIL
- COS
- COSH
- EXP
- FLOOR
- LN
- LOG
- MOD
- NANVL
- POWER
- REMAINDER
- ROUND (number)
- SIGN
- SIN
- SINH
- SQRT
- TAN
- TANH
- TRUNC (number)
- WIDTH_BUCKET

Character Functions Returning Character Values

Character functions that return character values return values of the following data types unless otherwise documented:

- If the input argument is `CHAR` or `VARCHAR2`, then the value returned is `VARCHAR2`.
- If the input argument is `NCHAR` or `NVARCHAR2`, then the value returned is `NVARCHAR2`.

The length of the value returned by the function is limited by the maximum length of the data type returned.

- For functions that return `CHAR` or `VARCHAR2`, if the length of the return value exceeds the limit, then Oracle Database truncates it and returns the result without an error message.
- For functions that return `CLOB` values, if the length of the return values exceeds the limit, then Oracle raises an error and returns no data.

The character functions that return character values are:

CHR
CONCAT
INITCAP
LOWER
LPAD
LTRIM
NCHR
NLS_INITCAP
NLS_LOWER
NLS_UPPER
NLSSORT
REGEXP_REPLACE
REGEXP_SUBSTR
REPLACE
RPAD
RTRIM
SOUNDEX
SUBSTR
TRANSLATE
TRANSLATE ... USING
TRIM
UPPER

Character Functions Returning Number Values

Character functions that return number values can take as their argument any character data type. The character functions that return number values are:

ASCII
INSTR
LENGTH
REGEXP_COUNT
REGEXP_INSTR

Character Set Functions

The character set functions return information about the character set. The character set functions are:

NLS_CHARSET_DECL_LEN
NLS_CHARSET_ID
NLS_CHARSET_NAME

Collation Functions

The collation functions return information about collation settings. The collation functions are:

COLLATION
NLS_COLLATION_ID
NLS_COLLATION_NAME

Datetime Functions

Datetime functions operate on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE), and interval (INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH) values.

Some of the datetime functions were designed for the Oracle DATE data type (ADD_MONTHS, CURRENT_DATE, LAST_DAY, NEW_TIME, and NEXT_DAY). If you provide a timestamp value as their argument, then Oracle Database internally converts the input type to a DATE value and returns a DATE value. The exceptions are the MONTHS_BETWEEN function, which returns a number, and the ROUND and TRUNC functions, which do not accept timestamp or interval values at all.

The remaining datetime functions were designed to accept any of the three types of data (date, timestamp, and interval) and to return a value of one of these types.

All of the datetime functions that return current system datetime information, such as SYSDATE, SYSTIMESTAMP, CURRENT_TIMESTAMP, and so forth, are evaluated once for each SQL statement, regardless how many times they are referenced in that statement.

The datetime functions are:

ADD_MONTHS
CURRENT_DATE
CURRENT_TIMESTAMP
DBTIMEZONE
EXTRACT (datetime)
FROM_TZ
LAST_DAY
LOCALTIMESTAMP
MONTHS_BETWEEN
NEW_TIME
NEXT_DAY
NUMTODSINTERVAL

NUMTOYMINTERVAL
ORA_DST_AFFECTED
ORA_DST_CONVERT
ORA_DST_ERROR
ROUND (date)
SESSIONTIMEZONE
SYS_EXTRACT_UTC
SYSDATE
SYSTIMESTAMP
TO_CHAR (datetime)
TO_DSINTERVAL
TO_TIMESTAMP
TO_TIMESTAMP_TZ
TO_YMINTERVAL
TRUNC (date)
TZ_OFFSET

General Comparison Functions

The general comparison functions determine the greatest and or least value from a set of values. The general comparison functions are:

GREATEST
LEAST

Conversion Functions

Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention `datatype TO datatype`. The first data type is the input data type. The second data type is the output data type. The SQL conversion functions are:

ASCIISTR
BIN_TO_NUM
CAST
CHARTOROWID
COMPOSE
CONVERT
DECOMPOSE
HEXTORAW
NUMTODSINTERVAL
NUMTOYMINTERVAL
RAWTOHEX
RAWTONHEX
ROWIDTOCHAR
ROWIDTONCHAR
SCN_TO_TIMESTAMP
TIMESTAMP_TO_SCN
TO_BINARY_DOUBLE
TO_BINARY_FLOAT

TO_BLOB (bfile)
TO_BLOB (raw)
TO_CHAR (bfile|blob)
TO_CHAR (character)
TO_CHAR (datetime)
TO_CHAR (number)
TO_CLOB (bfile|blob)
TO_CLOB (character)
TO_DATE
TO_DSINTERVAL
TO_LOB
TO_MULTI_BYTE
TO_NCHAR (character)
TO_NCHAR (datetime)
TO_NCHAR (number)
TO_NCLOB
TO_NUMBER
TO_SINGLE_BYTE
TO_TIMESTAMP
TO_TIMESTAMP_TZ
TO_YMINTERVAL
TREAT
UNISTR
VALIDATE_CONVERSION

Large Object Functions

The large object functions operate on LOBs. The large object functions are:

BFILENAME
EMPTY_BLOB, EMPTY_CLOB

Collection Functions

The collection functions operate on nested tables and varrays. The SQL collection functions are:

CARDINALITY
COLLECT
POWERMULTISET
POWERMULTISET_BY_CARDINALITY
SET

Hierarchical Functions

Hierarchical functions applies hierarchical path information to a result set. The hierarchical function is:

SYS_CONNECT_BY_PATH

Data Mining Functions

The data mining functions use Oracle Advanced Analytics to score data. The functions can apply a mining model schema object to the data, or they can dynamically mine the data by executing an analytic clause. The data mining functions can be applied to models built using the native algorithms of Oracle, as well as those built using R through the extensibility mechanism of Oracle Advanced Analytics.

The data mining functions are:

CLUSTER_DETAILS
CLUSTER_DISTANCE
CLUSTER_ID
CLUSTER_PROBABILITY
CLUSTER_SET
FEATURE_COMPARE
FEATURE_DETAILS
FEATURE_ID
FEATURE_SET
FEATURE_VALUE
ORA_DM_PARTITION_NAME
PREDICTION
PREDICTION_BOUNDS
PREDICTION_COST
PREDICTION_DETAILS
PREDICTION_PROBABILITY
PREDICTION_SET

See Also:

- *Oracle Data Mining Concepts* to learn about Oracle Data Mining
- *Oracle Data Mining User's Guide* for information about scoring

XML Functions

The XML functions operate on or return XML documents or fragments. These functions use arguments that are not defined as part of the ANSI/ISO/IEC SQL Standard but are defined as part of the World Wide Web Consortium (W3C) standards. The processing and operations that the functions perform are defined by the relevant W3C standards. The table below provides a link to the appropriate section of the W3C standard for the rules and guidelines that apply to each of these XML-related arguments. A SQL statement that uses one of these XML functions, where any of the arguments does not conform to the relevant W3C syntax, will result in an error. Of special note is the fact that not every character that is allowed in the value of a database column is considered legal in XML.

Syntax Element	W3C Standard URL
value_expr	http://www.w3.org/TR/2006/REC-xml-20060816

Syntax Element	W3C Standard URL
Xpath_string	`http://www.w3.org/TR/1999/REC-xpath-19991116`
XQuery_string	`http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/` `http://www.w3.org/TR/xquery-update-10/`
namespace_string	`http://www.w3.org/TR/2006/REC-xml-names-20060816/`
identifier	`http://www.w3.org/TR/2006/REC-xml-20060816/#NT-Nmtoken`

For more information about selecting and querying XML data using these functions, including information on formatting output, refer to *Oracle XML DB Developer's Guide*

The SQL XML functions are:

DEPTH
EXISTSNODE
EXTRACT (XML)
EXTRACTVALUE
PATH
SYS_DBURIGEN
SYS_XMLAGG
SYS_XMLGEN
XMLAGG
XMLCAST
XMLCDATA
XMLCOLATTVAL
XMLCOMMENT
XMLCONCAT
XMLDIFF
XMLELEMENT
XMLEXISTS
XMLFOREST
XMLISVALID
XMLPARSE
XMLPATCH
XMLPI
XMLQUERY
XMLROOT
XMLSEQUENCE
XMLSERIALIZE
XMLTABLE
XMLTRANSFORM

JSON Functions

JavaScript Object Notation (JSON) functions allow you to query and generate JSON data.

The following SQL/JSON functions allow you to query JSON data:

JSON_QUERY

JSON_TABLE
JSON_VALUE

The following SQL/JSON functions allow you to generate JSON data:

JSON_ARRAY
JSON_ARRAYAGG
JSON_OBJECT
JSON_OBJECTAGG

The following Oracle SQL function creates a JSON data guide:

JSON_DATAGUIDE

Encoding and Decoding Functions

The encoding and decoding functions let you inspect and decode data in the database. The encoding and decoding functions are:

DECODE
DUMP
ORA_HASH
STANDARD_HASH
VSIZE

NULL-Related Functions

The `NULL`-related functions facilitate null handling. The `NULL`-related functions are:

COALESCE
LNNVL
NANVL
NULLIF
NVL
NVL2

Environment and Identifier Functions

The environment and identifier functions provide information about the instance and session. The environment and identifier functions are:

CON_DBID_TO_ID
CON_GUID_TO_ID
CON_NAME_TO_ID
CON_UID_TO_ID
ORA_INVOKING_USER
ORA_INVOKING_USERID
SYS_CONTEXT
SYS_GUID
SYS_TYPEID
UID
USER
USERENV

Aggregate Functions

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in `ORDER BY` and `HAVING` clauses. They are commonly used with the `GROUP BY` clause in a `SELECT` statement, where Oracle Database divides the rows of a queried table or view into groups. In a query containing a `GROUP BY` clause, the elements of the select list can be aggregate functions, `GROUP BY` expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

If you omit the `GROUP BY` clause, then Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the `HAVING` clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

See Also:

- Using the GROUP BY Clause: Examples and the HAVING Clause for more information on the `GROUP BY` clause and `HAVING` clauses in queries and subqueries
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for expressions in the `ORDER BY` clause of an aggregate function

Many (but not all) aggregate functions that take a single argument accept these clauses:

- `DISTINCT` and `UNIQUE`, which are synonymous, cause an aggregate function to consider only distinct values of the argument expression. The syntax diagrams for aggregate functions in this chapter use the keyword `DISTINCT` for simplicity.
- `ALL` causes an aggregate function to consider all values, including all duplicates.

For example, the `DISTINCT` average of 1, 1, 1, and 3 is 2. The `ALL` average is 1.5. If you specify neither, then the default is `ALL`.

Some aggregate functions allow the *`windowing_clause`*, which is part of the syntax of analytic functions. Refer to *windowing_clause* for information about this clause. In the listing of aggregate functions at the end of this section, the functions that allow the *`windowing_clause`* are followed by an asterisk (*)

All aggregate functions except `COUNT`(*), `GROUPING`, and `GROUPING_ID` ignore nulls. You can use the `NVL` function in the argument to an aggregate function to substitute a value for a null. `COUNT` and `REGR_COUNT` never return null, but return either a number or zero. For all the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null.

The aggregate functions MIN, MAX, SUM, AVG, COUNT, VARIANCE, and STDDEV, when followed by the KEEP keyword, can be used in conjunction with the FIRST or LAST function to operate on a set of values from a set of rows that rank as the FIRST or LAST with respect to a given sorting specification. Refer to FIRST for more information.

You can nest aggregate functions. For example, the following example calculates the average of the maximum salaries of all the departments in the sample schema hr:

```
SELECT AVG(MAX(salary))
  FROM employees
  GROUP BY department_id;

AVG(MAX(SALARY))
----------------
      10926.3333
```

This calculation evaluates the inner aggregate (MAX(salary)) for each group defined by the GROUP BY clause (department_id), and aggregates the results again.

In the list of aggregate functions that follows, functions followed by an asterisk (*) allow the *windowing_clause*.

- APPROX_COUNT_DISTINCT
- APPROX_COUNT_DISTINCT_AGG
- APPROX_COUNT_DISTINCT_DETAIL
- APPROX_MEDIAN
- APPROX_PERCENTILE
- APPROX_PERCENTILE_AGG
- APPROX_PERCENTILE_DETAIL
- AVG
- COLLECT
- CORR
- CORR_*
- COUNT
- COVAR_POP
- COVAR_SAMP
- CUME_DIST
- DENSE_RANK
- FIRST
- GROUP_ID
- GROUPING
- GROUPING_ID
- JSON_ARRAYAGG
- JSON_OBJECTAGG
- LAST
- LISTAGG
- MAX
- MEDIAN
- MIN
- PERCENT_RANK
- PERCENTILE_CONT
- PERCENTILE_DISC
- RANK

- REGR_ (Linear Regression) Functions
- STATS_BINOMIAL_TEST
- STATS_CROSSTAB
- STATS_F_TEST
- STATS_KS_TEST
- STATS_MODE
- STATS_MW_TEST
- STATS_ONE_WAY_ANOVA
- STATS_T_TEST_*
- STATS_WSR_TEST
- STDDEV
- STDDEV_POP
- STDDEV_SAMP
- SUM
- SYS_OP_ZONE_ID
- SYS_XMLAGG
- TO_APPROX_COUNT_DISTINCT
- TO_APPROX_PERCENTILE
- VAR_POP
- VAR_SAMP
- VARIANCE
- XMLAGG

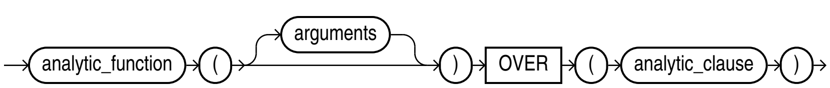
Analytic Functions

Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. The group of rows is called a **window** and is defined by the `analytic_clause`. For each row, a sliding window of rows is defined. The window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time.

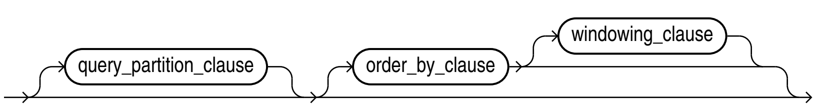
Analytic functions are the last set of operations performed in a query except for the final `ORDER BY` clause. All joins and all `WHERE`, `GROUP BY`, and `HAVING` clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or `ORDER BY` clause.

Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

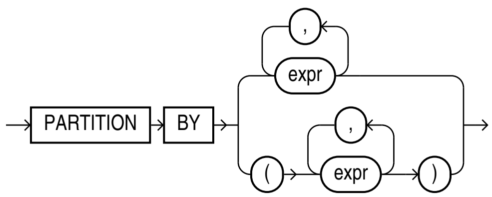
***analytic_function*::=**



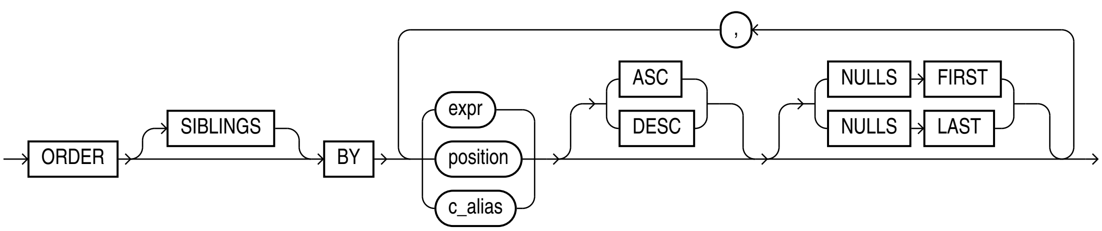
***analytic_clause*::=**



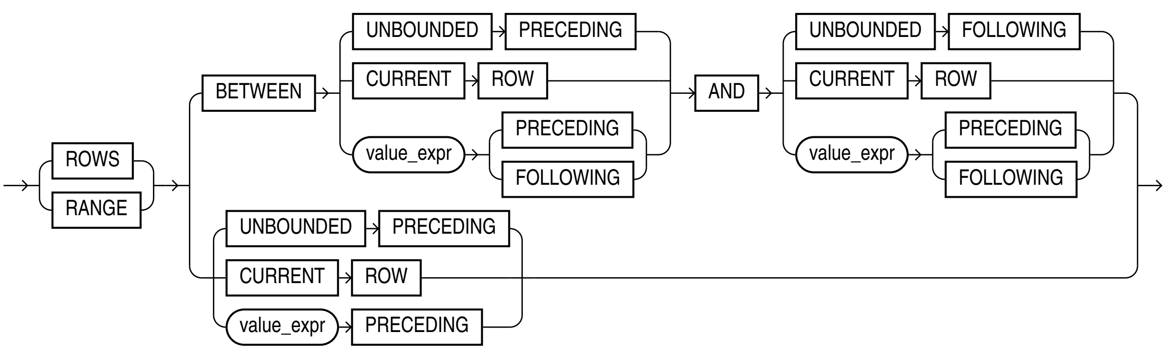
***query_partition_clause*::=**



***order_by_clause*::=**



***windowing_clause*::=**



The semantics of this syntax are discussed in the sections that follow.

analytic_function

Specify the name of an analytic function (see the listing of analytic functions following this discussion of semantics).

arguments

Analytic functions take 0 to 3 arguments. The arguments can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence and implicitly converts the remaining arguments to that data type. The return type is also that data type, unless otherwise noted for an individual function.

> **See Also:**
>
> Numeric Precedence for information on numeric precedence and Table 2-8 for more information on implicit conversion

analytic_clause

Use `OVER` *`analytic_clause`* to indicate that the function operates on a query result set. This clause is computed after the `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses. You can specify analytic functions with this clause in the select list or `ORDER BY` clause. To filter the results of a query based on an analytic function, nest these functions within the parent query, and then filter the results of the nested subquery.

Notes on the *analytic_clause*:

The following notes apply to the *`analytic_clause`*:

- You cannot nest analytic functions by specifying any analytic function in any part of the *`analytic_clause`*. However, you can specify an analytic function in a subquery and compute another analytic function over it.
- You can specify `OVER` *`analytic_clause`* with user-defined analytic functions as well as built-in analytic functions. See CREATE FUNCTION .
- The `PARTITION BY` and `ORDER BY` clauses in the *`analytic_clause`* are collation-sensitive.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `OVER (PARTITION BY` ... `ORDER BY` ... `)` clause of an analytic function

query_partition_clause

Use the `PARTITION BY` clause to partition the query result set into groups based on one or more *`value_expr`*. If you omit this clause, then the function treats all rows of the query result set as a single group.

To use the *`query_partition_clause`* in an analytic function, use the upper branch of the syntax (without parentheses). To use this clause in a model query (in the *`model_column_clauses`*) or a partitioned outer join (in the *`outer_join_clause`*), use the lower branch of the syntax (with parentheses).

You can specify multiple analytic functions in the same query, each with the same or different `PARTITION BY` keys.

If the objects being queried have the parallel attribute, and if you specify an analytic function with the *`query_partition_clause`*, then the function computations are parallelized as well.

Valid values of *`value_expr`* are constants, columns, nonanalytic functions, function expressions, or expressions involving any of these.

order_by_clause

Use the *order_by_clause* to specify how data is ordered within a partition. For all analytic functions you can order the values in a partition on multiple keys, each defined by a *value_expr* and each qualified by an ordering sequence.

Within each function, you can specify multiple ordering expressions. Doing so is especially useful when using functions that rank values, because the second expression can resolve ties between identical values for the first expression.

Whenever the *order_by_clause* results in identical values for multiple rows, the function behaves as follows:

- CUME_DIST, DENSE_RANK, NTILE, PERCENT_RANK, and RANK return the same result for each of the rows.
- ROW_NUMBER assigns each row a distinct value even if there is a tie based on the *order_by_clause*. The value is based on the order in which the row is processed, which may be nondeterministic if the ORDER BY does not guarantee a total ordering.
- For all other analytic functions, the result depends on the window specification. If you specify a logical window with the RANGE keyword, then the function returns the same result for each of the rows. If you specify a physical window with the ROWS keyword, then the result is nondeterministic.

Restrictions on the ORDER BY Clause

The following restrictions apply to the ORDER BY clause:

- When used in an analytic function, the *order_by_clause* must take an expression (*expr*). The SIBLINGS keyword is not valid (it is relevant only in hierarchical queries). Position (*position*) and column aliases (*c_alias*) are also invalid. Otherwise this *order_by_clause* is the same as that used to order the overall query or subquery.
- An analytic function that uses the RANGE keyword can use multiple sort keys in its ORDER BY clause if it specifies any of the following windows:
  - RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. The short form of this is RANGE UNBOUNDED PRECEDING.
  - RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
  - RANGE BETWEEN CURRENT ROW AND CURRENT ROW
  - RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

  Window boundaries other than these four can have only one sort key in the ORDER BY clause of the analytic function. This restriction does not apply to window boundaries specified by the ROW keyword.

ASC | DESC

Specify the ordering sequence (ascending or descending). ASC is the default.

NULLS FIRST | NULLS LAST

Specify whether returned rows containing nulls should appear first or last in the ordering sequence.

`NULLS LAST` is the default for ascending order, and `NULLS FIRST` is the default for descending order.

Analytic functions always operate on rows in the order specified in the *order_by_clause* of the function. However, the *order_by_clause* of the function does not guarantee the order of the result. Use the *order_by_clause* of the query to guarantee the final result ordering.

> **See Also:**
>
> *order_by_clause* of SELECT for more information on this clause

windowing_clause

Some analytic functions allow the *windowing_clause*. In the listing of analytic functions at the end of this section, the functions that allow the *windowing_clause* are followed by an asterisk (*).

ROWS | RANGE

These keywords define for each row a window (a physical or logical set of rows) used for calculating the function result. The function is then applied to all the rows in the window. The window moves through the query result set or partition from top to bottom.

- `ROWS` specifies the window in physical units (rows).
- `RANGE` specifies the window as a logical offset.

You cannot specify this clause unless you have specified the *order_by_clause*. Some window boundaries defined by the `RANGE` clause let you specify only one expression in the *order_by_clause*. Refer to Restrictions on the ORDER BY Clause.

The value returned by an analytic function with a logical offset is always deterministic. However, the value returned by an analytic function with a physical offset may produce nondeterministic results unless the ordering expression results in a unique ordering. You may have to specify multiple columns in the *order_by_clause* to achieve this unique ordering.

BETWEEN ... AND

Use the `BETWEEN` ... `AND` clause to specify a start point and end point for the window. The first expression (before `AND`) defines the start point and the second expression (after `AND`) defines the end point.

If you omit `BETWEEN` and specify only one end point, then Oracle considers it the start point, and the end point defaults to the current row.

UNBOUNDED PRECEDING

Specify `UNBOUNDED PRECEDING` to indicate that the window starts at the first row of the partition. This is the start point specification and cannot be used as an end point specification.

UNBOUNDED FOLLOWING

Specify `UNBOUNDED FOLLOWING` to indicate that the window ends at the last row of the partition. This is the end point specification and cannot be used as a start point specification.

CURRENT ROW

As a start point, `CURRENT ROW` specifies that the window begins at the current row or value (depending on whether you have specified `ROW` or `RANGE`, respectively). In this case the end point cannot be *`value_expr`* `PRECEDING`.

As an end point, `CURRENT ROW` specifies that the window ends at the current row or value (depending on whether you have specified `ROW` or `RANGE`, respectively). In this case the start point cannot be *`value_expr`* `FOLLOWING`.

value_expr PRECEDING or *value_expr* FOLLOWING

For `RANGE` or `ROW`:

- If *`value_expr`* `FOLLOWING` is the start point, then the end point must be *`value_expr`* `FOLLOWING`.
- If *`value_expr`* `PRECEDING` is the end point, then the start point must be *`value_expr`* `PRECEDING`.

If you are defining a logical window defined by an interval of time in numeric format, then you may need to use conversion functions.

> **See Also:**
>
> NUMTOYMINTERVAL and NUMTODSINTERVAL for information on converting numeric times into intervals

If you specified `ROWS`:

- *`value_expr`* is a physical offset. It must be a constant or expression and must evaluate to a positive numeric value.
- If *`value_expr`* is part of the start point, then it must evaluate to a row before the end point.

If you specified `RANGE`:

- *`value_expr`* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an interval literal. Refer to Literals for information on interval literals.
- You can specify only one expression in the *`order_by_clause`*.
- If *`value_expr`* evaluates to a numeric value, then the `ORDER BY` *`expr`* must be a numeric or `DATE` data type.
- If *`value_expr`* evaluates to an interval value, then the `ORDER BY` *`expr`* must be a `DATE` data type.

If you omit the *`windowing_clause`* entirely, then the default is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Analytic functions are commonly used in data warehousing environments. In the list of analytic functions that follows, functions followed by an asterisk (*) allow the full syntax, including the *windowing_clause*.

AVG *
CLUSTER_DETAILS
CLUSTER_DISTANCE
CLUSTER_ID
CLUSTER_PROBABILITY
CLUSTER_SET
CORR *
COUNT *
COVAR_POP *
COVAR_SAMP *
CUME_DIST
DENSE_RANK
FEATURE_DETAILS
FEATURE_ID
FEATURE_SET
FEATURE_VALUE
FIRST
FIRST_VALUE *
LAG
LAST
LAST_VALUE *
LEAD
LISTAGG
MAX *
MIN *
NTH_VALUE *
NTILE
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DISC
PREDICTION
PREDICTION_COST
PREDICTION_DETAILS
PREDICTION_PROBABILITY
PREDICTION_SET
RANK
RATIO_TO_REPORT
REGR_ (Linear Regression) Functions *
ROW_NUMBER
STDDEV *
STDDEV_POP *
STDDEV_SAMP *
SUM *
VAR_POP *
VAR_SAMP *
VARIANCE *

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on these functions and for scenarios illustrating their use

Object Reference Functions

Object reference functions manipulate `REF` values, which are references to objects of specified object types. The object reference functions are:

DEREF
MAKE_REF
REF
REFTOHEX
VALUE

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information about `REF` data types

Model Functions

Model functions can be used only in the *`model_clause`* of the `SELECT` statement. The model functions are:

CV
ITERATION_NUMBER
PRESENTNNV
PRESENTV
PREVIOUS

OLAP Functions

OLAP functions returns data from a dimensional object in two-dimension relational format. The OLAP function is:

CUBE_TABLE

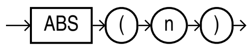
Data Cartridge Functions

Data Cartridge functions are useful for Data Cartridge developers. The Data Cartridge functions are:

DATAOBJ_TO_MAT_PARTITION
DATAOBJ_TO_PARTITION

ABS

Syntax



Purpose

`ABS` returns the absolute value of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

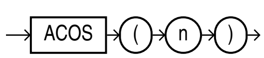
Examples

The following example returns the absolute value of -15:

```
SELECT ABS(-15) "Absolute"
  FROM DUAL;

  Absolute
----------
        15
```

ACOS

Syntax



Purpose

`ACOS` returns the arc cosine of *n*. The argument *n* must be in the range of -1 to 1, and the function returns a value in the range of 0 to *pi*, expressed in radians.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

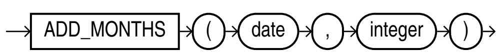
Examples

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3)"Arc_Cosine"
  FROM DUAL;

Arc_Cosine
----------
1.26610367
```

ADD_MONTHS

Syntax



Purpose

`ADD_MONTHS` returns the date *`date`* plus *`integer`* months. A month is defined by the session parameter `NLS_CALENDAR`. The date argument can be a datetime value or any value that can be implicitly converted to `DATE`. The *`integer`* argument can be an integer or any value that can be implicitly converted to an integer. The return type is always `DATE`, regardless of the data type of *`date`*. If *`date`* is the last day of the month or if the resulting month has fewer days than the day component of *`date`*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *`date`*.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

Examples

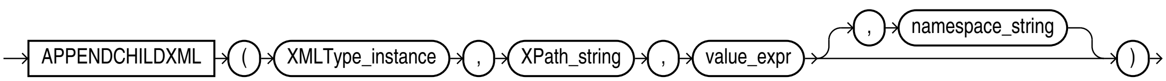
The following example returns the month after the *`hire_date`* in the sample table `employees`:

```
SELECT TO_CHAR(ADD_MONTHS(hire_date, 1), 'DD-MON-YYYY') "Next month"
  FROM employees
  WHERE last_name = 'Baer';

Next Month
-----------
07-JUL-2002
```

APPENDCHILDXML

> **Note:**
>
> The `APPENDCHILDXML` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`APPENDCHILDXML` appends a user-supplied value onto the target XML as the child of the node indicated by an XPath expression.

- *`XMLType_instance`* is an instance of `XMLType`.
- *`XPath_string`* is an Xpath expression indicating one or more nodes onto which one or more child nodes are to be appended. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node.
- *`value_expr`* specifies one or more nodes of `XMLType`. It must resolve to a string.
- The optional *`namespace_string`* provides namespace information for the *`XPath_string`*. This parameter must be of type `VARCHAR2`.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

Examples

The following example adds an `/Owner` node to the `/Warehouse/Building` node of `warehouse_spec` in the `oe.warehouses` table if the value of the `/Building` node is "Rented":

```
UPDATE warehouses
  SET warehouse_spec = APPENDCHILDXML(warehouse_spec, 'Warehouse/Building',
    XMLType('<Owner>Grandco</Owner>'))
  WHERE EXTRACTVALUE(warehouse_spec, '/Warehouse/Building') = 'Rented';

SELECT warehouse_id,
       warehouse_name,
       EXTRACTVALUE(warehouse_spec, '/Warehouse/Building/Owner') "Prop.Owner"
```

```
  FROM warehouses
  WHERE EXISTSNODE(warehouse_spec, '/Warehouse/Building/Owner') = 1;

WAREHOUSE_ID WAREHOUSE_NAME  Prop.Owner
------------ --------------- ----------
           2 San Francisco   Grandco
           3 New Jersey      Grandco
```

APPROX_COUNT_DISTINCT

Syntax

Purpose

`APPROX_COUNT_DISTINCT` returns the approximate number of rows that contain a distinct value for *`expr`*.

This function provides an alternative to the `COUNT (DISTINCT` *`expr`*`)` function, which returns the exact number of rows that contain distinct values of *`expr`*. `APPROX_COUNT_DISTINCT` processes large amounts of data significantly faster than `COUNT`, with negligible deviation from the exact result.

For *`expr`*, you can specify a column of any scalar data type other than `BFILE`, `BLOB`, `CLOB`, `LONG`, `LONG RAW`, or `NCLOB`.

`APPROX_COUNT_DISTINCT` ignores rows that contain a null value for *`expr`*. This function returns a `NUMBER`.

> **See Also:**
>
> - COUNT for more information on the `COUNT (DISTINCT` *`expr`*`)` function
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `APPROX_COUNT_DISTINCT` uses to compare character values for *`expr`*

Examples

The following statement returns the approximate number of rows with distinct values for `manager_id`:

```
SELECT APPROX_COUNT_DISTINCT(manager_id) AS "Active Managers"
  FROM employees;

Active Managers
---------------
             18
```

The following statement returns the approximate number of distinct customers for each product:

```
SELECT prod_id, APPROX_COUNT_DISTINCT(cust_id) AS "Number of Customers"
  FROM sales
  GROUP BY prod_id
  ORDER BY prod_id;

   PROD_ID Number of Customers
---------- -------------------
        13                2516
        14                2030
        15                2105
        16                2367
        17                2093
        18                2975
        19                2630
        20                3791
. . .
```

APPROX_COUNT_DISTINCT_AGG

Syntax

Purpose

APPROX_COUNT_DISTINCT_AGG takes as its input a column of details containing information about approximate distinct value counts, and enables you to perform aggregations of those counts.

For *detail*, specify a column of details created by the APPROX_COUNT_DISTINCT_DETAIL function or the APPROX_COUNT_DISTINCT_AGG function. This column is of data type BLOB.

You can specify this function in a SELECT statement with a GROUP BY clause to aggregate the information contained in the details within each group of rows and return a single detail for each group.

This function returns a BLOB value, called a detail, which contains information about the count aggregations in a special format. You can store details returned by this function in a table or materialized view, and then again use the APPROX_COUNT_DISTINCT_AGG function to further aggregate those details, or use the TO_APPROX_COUNT_DISTINCT function to convert the detail values to human-readable NUMBER values.

See Also:

- APPROX_COUNT_DISTINCT_DETAIL
- TO_APPROX_COUNT_DISTINCT

Examples

Refer to APPROX_COUNT_DISTINCT_AGG: Examples for examples of using the `APPROX_COUNT_DISTINCT_AGG` function in conjunction with the `APPROX_COUNT_DISTINCT_DETAIL` and `TO_APPROX_COUNT_DISTINCT` functions.

APPROX_COUNT_DISTINCT_DETAIL

Syntax

Purpose

`APPROX_COUNT_DISTINCT_DETAIL` calculates information about the approximate number of rows that contain a distinct value for expr and returns a BLOB value, called a detail, which contains that information in a special format.

For *`expr`*, you can specify a column of any scalar data type other than `BFILE`, `BLOB`, `CLOB`, `LONG`, `LONG RAW`, or `NCLOB`. This function ignores rows for which the value of *`expr`* is null.

This function is commonly used with the `GROUP BY` clause in a `SELECT` statement. When used in this way, it calculates approximate distinct value count information for *`expr`* within each group of rows and returns a single detail for each group.

The details returned by `APPROX_COUNT_DISTINCT_DETAIL` can be used as input to the `APPROX_COUNT_DISTINCT_AGG` function, which enables you to perform aggregations of the details, or the `TO_APPROX_COUNT_DISTINCT` function, which converts a detail to a human-readable distinct count value. You can use these three functions together to perform resource-intensive approximate count calculations once, store the resulting details, and then perform efficient aggregations and queries on those details. For example:

1. Use the `APPROX_COUNT_DISTINCT_DETAIL` function to calculate approximate distinct value count information and store the resulting details in a table or materialized view. These could be highly-granular details, such as city demographic counts or daily sales counts.
2. Use the `APPROX_COUNT_DISTINCT_AGG` function to aggregate the details obtained in the previous step and store the resulting details in a table or materialized view. These could be details of lower granularity, such as state demographic counts or monthly sales counts.
3. Use the `TO_APPROX_COUNT_DISTINCT` function to convert the stored detail values to human-readable `NUMBER` values. You can use the `TO_APPROX_COUNT_DISTINCT` function to query detail values created by the `APPROX_COUNT_DISTINCT_DETAIL` function or the `APPROX_COUNT_DISTNCT_AGG` function.

See Also:

- APPROX_COUNT_DISTINCT_AGG
- TO_APPROX_COUNT_DISTINCT

Examples

The examples in this section demonstrate how to use the `APPROX_COUNT_DISTINCT_DETAIL`, `APPROX_COUNT_DISTINCT_AGG`, and `TO_APPROX_COUNT_DISTINCT` functions together to perform resource-intensive approximate count calculations once, store the resulting details, and then perform efficient aggregations and queries on those details.

APPROX_COUNT_DISTINCT_DETAIL: Example

The following statement queries the tables `sh.times` and `sh.sales` for the approximate number of distinct products sold each day. The `APPROX_COUNT_DISTINCT_DETAIL` function returns the information in a detail, called `daily_detail`, for each day that products were sold. The returned details are stored in a materialized view called `daily_prod_count_mv`.

```
CREATE MATERIALIZED VIEW daily_prod_count_mv AS
  SELECT t.calendar_year year,
         t.calendar_month_number month,
         t.day_number_in_month day,
         APPROX_COUNT_DISTINCT_DETAIL(s.prod_id) daily_detail
  FROM times t, sales s
  WHERE t.time_id = s.time_id
  GROUP BY t.calendar_year, t.calendar_month_number, t.day_number_in_month;
```

APPROX_COUNT_DISTINCT_AGG: Examples

The following statement uses the `APPROX_COUNT_DISTINCT_AGG` function to read the daily details stored in `daily_prod_count_mv` and create aggregated details that contain the approximate number of distinct products sold each month. These aggregated details are stored in a materialized view called `monthly_prod_count_mv`.

```
CREATE MATERIALIZED VIEW monthly_prod_count_mv AS
  SELECT year,
         month,
         APPROX_COUNT_DISTINCT_AGG(daily_detail) monthly_detail
  FROM daily_prod_count_mv
  GROUP BY year, month;
```

The following statement is similar to the previous statement, except it creates aggregated details that contain the approximate number of distinct products sold each year. These aggregated details are stored in a materialized view called `annual_prod_count_mv`.

```
CREATE MATERIALIZED VIEW annual_prod_count_mv AS
  SELECT year,
         APPROX_COUNT_DISTINCT_AGG(daily_detail) annual_detail
  FROM daily_prod_count_mv
  GROUP BY year;
```

TO_APPROX_COUNT_DISTINCT: Examples

The following statement uses the `TO_APPROX_COUNT_DISTINCT` function to query the daily detail information stored in `daily_prod_count_mv` and return the approximate number of distinct products sold each day:

```
SELECT year,
       month,
```

```
       day,
       TO_APPROX_COUNT_DISTINCT(daily_detail) "NUM PRODUCTS"
  FROM daily_prod_count_mv
  ORDER BY year, month, day;

      YEAR      MONTH        DAY NUM PRODUCTS
---------- ---------- ---------- ------------
      1998          1          1           24
      1998          1          2           25
      1998          1          3           11
      1998          1          4           34
      1998          1          5           10
      1998          1          6            8
      1998          1          7           37
      1998          1          8           26
      1998          1          9           25
      1998          1         10           38
. . .
```

The following statement uses the `TO_APPROX_COUNT_DISTINCT` function to query the monthly detail information stored in `monthly_prod_count_mv` and return the approximate number of distinct products sold each month:

```
SELECT year,
       month,
       TO_APPROX_COUNT_DISTINCT(monthly_detail) "NUM PRODUCTS"
  FROM monthly_prod_count_mv
  ORDER BY year, month;

      YEAR      MONTH NUM PRODUCTS
---------- ---------- ------------
      1998          1           57
      1998          2           56
      1998          3           55
      1998          4           49
      1998          5           49
      1998          6           48
      1998          7           54
      1998          8           56
      1998          9           55
      1998         10           57
. . .
```

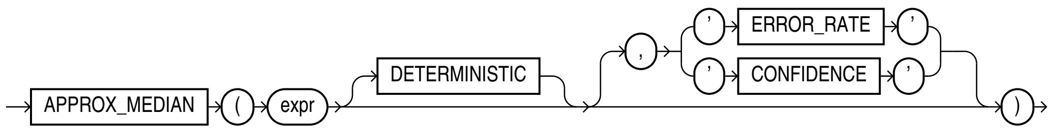
The following statement uses the `TO_APPROX_COUNT_DISTINCT` function to query the annual detail information stored in `annual_prod_count_mv` and return the approximate number of distinct products sold each year:

```
SELECT year,
       TO_APPROX_COUNT_DISTINCT(annual_detail) "NUM PRODUCTS"
  FROM annual_prod_count_mv
  ORDER BY year;

      YEAR NUM PRODUCTS
```

```
---------- ------------
      1998           60
      1999           72
      2000           72
      2001           71
```

APPROX_MEDIAN

Syntax



Purpose

`APPROX_MEDIAN` is an approximate inverse distribution function that assumes a continuous distribution model. It takes a numeric or datetime value and returns an approximate middle value or an approximate interpolated value that would be the middle value once the values are sorted. Nulls are ignored in the calculation.

This function provides an alternative to the `MEDIAN` function, which returns the exact middle value or interpolated value. `APPROX_MEDIAN` processes large amounts of data significantly faster than `MEDIAN`, with negligible deviation from the exact result.

For *`expr`*, specify the expression for which the approximate median value is being calculated. The acceptable data types for *`expr`*, and the return value data type for this function, depend on the algorithm that you specify with the `DETERMINISTIC` clause.

DETERMINISTIC

This clause lets you specify the type of algorithm this function uses to calculate the approximate median value.

- If you specify `DETERMINISTIC`, then this function calculates a deterministic approximate median value. In this case, *`expr`* must evaluate to a numeric value, or to a value that can be implicitly converted to a numeric value. The function returns the same data type as the numeric data type of its argument.
- If you omit `DETERMINSTIC`, then this function calculates a nondeterministic approximate median value. In this case, *`expr`* must evaluate to a numeric or datetime value, or to a value that can be implicitly converted to a numeric or datetime value. The function returns the same data type as the numeric or datetime data type of its argument.

ERROR_RATE | CONFIDENCE

These clauses let you determine the accuracy of the value calculated by this function. If you specify one of these clauses, then instead of returning the approximate median value for *`expr`*, the function returns a decimal value from 0 to 1, inclusive, which represents one of the following values:

- If you specify `ERROR_RATE`, then the return value represents the error rate for the approximate median value calculation for *`expr`*.

- If you specify CONFIDENCE, then the return value represents the confidence level for the error rate that is returned when you specify ERROR_RATE.

See Also:

- MEDIAN
- APPROX_PERCENTILE which returns, for a given percentile, the approximate value that corresponds to that percentile by way of interpolation. APPROX_MEDIAN is the specific case of APPROX_PERCENTILE where the percentile value is 0.5.

Examples

The following query returns the deterministic approximate median salary for each department in the hr.employees table:

```
SELECT department_id "Department",
       APPROX_MEDIAN(salary DETERMINISTIC) "Median Salary"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Median Salary
---------- -------------
        10          4400
        20          6000
        30          2765
        40          6500
        50          3100
        60          4800
        70         10000
        80          9003
        90         17000
       100          7739
       110          8300
                    7000
```

The following query returns the error rates for the approximate median salaries that were returned by the previous query:

```
SELECT department_id "Department",
       APPROX_MEDIAN(salary DETERMINISTIC, 'ERROR_RATE') "Error Rate"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Error Rate
---------- ----------
        10 .002718282
        20 .021746255
        30 .021746255
        40 .002718282
        50 .019027973
        60 .019027973
        70 .002718282
        80 .021746255
        90 .021746255
```

```
       100 .019027973
       110 .019027973
           .002718282
```

The following query returns the confidence levels for the error rates that were returned by the previous query:

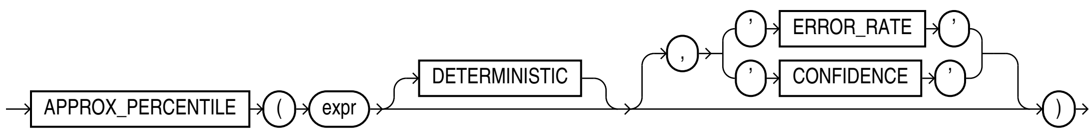
```
SELECT department_id "Department",
       APPROX_MEDIAN(salary DETERMINISTIC, 'CONFIDENCE') "Confidence Level"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Confidence Level
---------- ----------------
        10       .997281718
        20       .999660215
        30       .999660215
        40       .997281718
        50       .999611674
        60       .999611674
        70       .997281718
        80       .999660215
        90       .999660215
       100       .999611674
       110       .999611674
                 .997281718
```

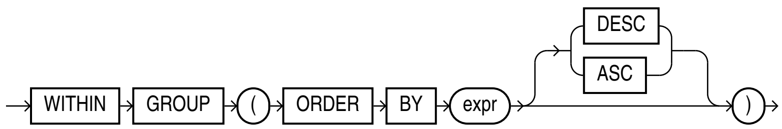
The following query returns the nondeterministic approximate median hire date for each department in the `hr.employees` table:

```
SELECT department_id "Department",
       APPROX_MEDIAN(hire_date) "Median Hire Date"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Median Hire Date
---------- ----------------
        10        17-SEP-03
        20        17-FEB-04
        30        24-JUL-05
        40        07-JUN-02
        50        15-MAR-06
        60        05-FEB-06
        70        07-JUN-02
        80        23-MAR-06
        90        17-JUN-03
       100        28-SEP-05
       110        07-JUN-02
                  24-MAY-07
```

APPROX_PERCENTILE

Syntax





Purpose

APPROX_PERCENTILE is an approximate inverse distribution function. It takes a percentile value and a sort specification, and returns the value that would fall into that percentile value with respect to the sort specification. Nulls are ignored in the calculation

This function provides an alternative to the PERCENTILE_CONT and PERCENTILE_DISC functions, which returns the exact results. APPROX_PERCENTILE processes large amounts of data significantly faster than PERCENTILE_CONT and PERCENTILE_DISC, with negligible deviation from the exact result.

The first *expr* is the percentile value, which must evaluate to a numeric value between 0 and 1.

The second *expr*, which is part of the ORDER BY clause, is a single expression over which this function calculates the result. The acceptable data types for *expr*, and the return value data type for this function, depend on the algorithm that you specify with the DETERMINISTIC clause.

DETERMINISTIC

This clause lets you specify the type of algorithm this function uses to calculate the return value.

- If you specify DETERMINISTIC, then this function calculates a deterministic result. In this case, the ORDER BY clause expression must evaluate to a numeric value, or to a value that can be implicitly converted to a numeric value, in the range -2,147,483,648 through 2,147,483,647. The function rounds numeric input to the closest integer. The function returns the same data type as the numeric data type of the ORDER BY clause expression. The return value is not necessarily one of the values of *expr*
- If you omit DETERMINSTIC, then this function calculates a nondeterministic result. In this case, the ORDER BY clause expression must evaluate to a numeric or datetime value, or to a value that can be implicitly converted to a numeric or datetime value. The function returns the same data type as the numeric or datetime data type of the ORDER BY clause expression. The return value is one of the values of *expr*.

ERROR_RATE | CONFIDENCE

These clauses let you determine the accuracy of the result calculated by this function. If you specify one of these clauses, then instead of returning the value that would fall into the specified percentile value for *expr*, the function returns a decimal value from 0 to 1, inclusive, which represents one of the following values:

- If you specify `ERROR_RATE`, then the return value represents the error rate for calculating the value that would fall into the specified percentile value for*expr*.
- If you specify `CONFIDENCE`, then the return value represents the confidence level for the error rate that is returned when you specify `ERROR_RATE`.

DESC | ASC

Specify the sort specification for the calculating the value that would fall into the specified percentile value. Specify `DESC` to sort the `ORDER BY` clause expression values in descending order, or `ASC` to sort the values in ascending order. `ASC` is the default.

> **See Also:**
>
> - PERCENTILE_CONT and PERCENTILE_DISC
> - APPROX_MEDIAN, which is the specific case of `APPROX_PERCENTILE` where the percentile value is 0.5

Examples

The following query returns the deterministic approximate 25th percentile, 50th percentile, and 75th percentile salaries for each department in the `hr.employees` table. The salaries are sorted in ascending order for the interpolation calculation.

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25 DETERMINISTIC)
         WITHIN GROUP (ORDER BY salary ASC) "25th Percentile Salary",
       APPROX_PERCENTILE(0.50 DETERMINISTIC)
         WITHIN GROUP (ORDER BY salary ASC) "50th Percentile Salary",
       APPROX_PERCENTILE(0.75 DETERMINISTIC)
         WITHIN GROUP (ORDER BY salary ASC) "75th Percentile Salary"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department 25th Percentile Salary 50th Percentile Salary 75th Percentile Salary
---------- ---------------------- ---------------------- ----------------------
        10                   4400                   4400                   4400
        20                   6000                   6000                  13000
        30                   2633                   2765                   3100
        40                   6500                   6500                   6500
        50                   2600                   3100                   3599
        60                   4800                   4800                   6000
        70                  10000                  10000                  10000
        80                   7400                   9003                  10291
        90                  17000                  17000                  24000
       100                   7698                   7739                   8976
       110                   8300                   8300                  12006
                             7000                   7000                   7000
```

The following query returns the error rates for the approximate 25th percentile salaries that were calculated in the previous query:

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25 DETERMINISTIC, 'ERROR_RATE')
         WITHIN GROUP (ORDER BY salary ASC) "Error Rate"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Error Rate
---------- ----------
        10 .002718282
        20 .021746255
        30 .021746255
        40 .002718282
        50 .019027973
        60 .019027973
        70 .002718282
        80 .021746255
        90 .021746255
       100 .019027973
       110 .019027973
           .002718282
```

The following query returns the confidence levels for the error rates that were calculated in the previous query:

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25 DETERMINISTIC, 'CONFIDENCE')
         WITHIN GROUP (ORDER BY salary ASC) "Confidence"
FROM employees
GROUP BY department_id
ORDER BY department_id;

Department Confidence
---------- ----------
        10 .997281718
        20 .999660215
        30 .999660215
        40 .997281718
        50 .999611674
        60 .999611674
        70 .997281718
        80 .999660215
        90 .999660215
       100 .999611674
       110 .999611674
           .997281718
```

The following query returns the nondeterministic approximate 25th percentile, 50th percentile, and 75th percentile salaries for each department in the `hr.employees` table. The salaries are sorted in ascending order for the interpolation calculation.

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25)
         WITHIN GROUP (ORDER BY salary ASC) "25th Percentile Salary",
       APPROX_PERCENTILE(0.50)
         WITHIN GROUP (ORDER BY salary ASC) "50th Percentile Salary",
       APPROX_PERCENTILE(0.75)
         WITHIN GROUP (ORDER BY salary ASC) "75th Percentile Salary"
```

```
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department 25th Percentile Salary 50th Percentile Salary 75th Percentile Salary
---------- ---------------------- ---------------------- ----------------------
        10                   4400                   4400                   4400
        20                   6000                   6000                  13000
        30                   2600                   2800                   3100
        40                   6500                   6500                   6500
        50                   2600                   3100                   3600
        60                   4800                   4800                   6000
        70                  10000                  10000                  10000
        80                   7300                   8800                  10000
        90                  17000                  17000                  24000
       100                   7700                   7800                   9000
       110                   8300                   8300                  12008
                             7000                   7000                   7000
```

APPROX_PERCENTILE_AGG

Syntax

Purpose

APPROX_PERCENTILE_AGG takes as its input a column of details containing approximate percentile information, and enables you to perform aggregations of that information.

For *detail*, specify a column of details created by the APPROX_PERCENT_DETAIL function or the APPROX_PERCENTILE_AGG function. This column is of data type BLOB.

You can specify this function in a SELECT statement with a GROUP BY clause to aggregate the information contained in the details within each group of rows and return a single detail for each group.

This function returns a BLOB value, called a detail, which contains approximate percentile information in a special format. You can store details returned by this function in a table or materialized view, and then again use the APPROX_PERCENTILE_AGG function to further aggregate those details, or use the TO_APPROX_PERCENTILE function to convert the details to specified percentile values.

See Also:

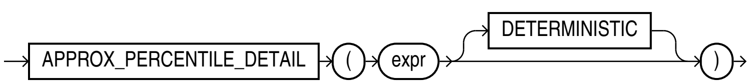
- APPROX_PERCENTILE_DETAIL
- TO_APPROX_PERCENTILE

Examples

Refer to APPROX_PERCENTILE_AGG: Examples for examples of using the `APPROX_PERCENTILE_AGG` function in conjunction with the `APPROX_PERCENTILE_DETAIL` and `TO_APPROX_PERCENTILE` functions.

APPROX_PERCENTILE_DETAIL

Syntax



Purpose

`APPROX_PERCENTILE_DETAIL` calculates approximate percentile information for the values of *`expr`* and returns a `BLOB` value, called a detail, which contains that information in a special format.

The acceptable data types for *`expr`* depend on the algorithm that you specify with the `DETERMINISTIC` clause. Refer to the DETERMINISTIC clause for more information.

This function is commonly used with the `GROUP BY` clause in a `SELECT` statement. It calculates approximate percentile information for *`expr`* within each group of rows and returns a single detail for each group.

The details returned by `APPROX_PERCENTILE_DETAIL` can be used as input to the `APPROX_PERCENTILE_AGG` function, which enables you to perform aggregations of the details, or the `TO_APPROX_PERCENTILE` function, which converts a detail to a specified percentile value. You can use these three functions together to perform resource-intensive approximate percentile calculations once, store the resulting details, and then perform efficient aggregations and queries on those details. For example:

1. Use the `APPROX_PERCENTILE_DETAIL` function to perform approximate percentile calculations and store the resulting details in a table or materialized view. These could be highly-granular percentile details, such as income percentile information for cities.
2. Use the `APPROX_PERCENTILE_AGG` function to aggregate the details obtained in the previous step and store the resulting details in a table or materialized view. These could be details of lower granularity, such as income percentile information for states.
3. Use the `TO_APPROX_PERCENTILE` function to convert the stored detail values to percentile values. You can use the `TO_APPROX_PERCENTILE` function to query detail values created by the `APPROX_PERCENTILE_DETAIL` function or the `APPROX_PERCENTILE_AGG` function.

DETERMINISTIC

This clause lets you control the type of algorithm used to calculate the approximate percentile values.

- If you specify `DETERMINISTIC`, then this function calculates deterministic approximate percentile information. In this case, *`expr`* must evaluate to a numeric value, or to a value that can be implicitly converted to a numeric value.

- If you omit `DETERMINSTIC`, then this function calculates nondeterministic approximate percentile information. In this case, *`expr`* must evaluate to a numeric or datetime value, or to a value that can be implicitly converted to a numeric or datetime value.

> **See Also:**
>
> - APPROX_PERCENTILE_AGG
> - TO_APPROX_PERCENTILE

Examples

The examples in this section demonstrate how to use the `APPROX_PERCENTILE_DETAIL`, `APPROX_PERCENTILE_AGG`, and `TO_APPROX_PERCENTILE` functions together to perform resource-intensive approximate percentile calculations once, store the resulting details, and then perform efficient aggregations and queries on those details.

APPROX_PERCENTILE_DETAIL: Example

The following statement queries the tables `sh.customers` and `sh.sales` for the monetary amounts for products sold to each customer. The `APPROX_PERCENTILE_DETAIL` function returns the information in a detail, called `city_detail`, for each city in which customers reside. The returned details are stored in a materialized view called `amt_sold_by_city_mv`.

```
CREATE MATERIALIZED VIEW amt_sold_by_city_mv
ENABLE QUERY REWRITE AS
SELECT c.country_id country,
       c.cust_state_province state,
       c.cust_city city,
       APPROX_PERCENTILE_DETAIL(s.amount_sold) city_detail
FROM customers c, sales s
WHERE c.cust_id = s.cust_id
GROUP BY c.country_id, c.cust_state_province, c.cust_city;
```

APPROX_PERCENTILE_AGG: Examples

The following statement uses the `APPROX_PERCENTILE_AGG` function to read the details stored in `amt_sold_by_city_mv` and create aggregated details that contain the monetary amounts for products sold to customers in each state. These aggregated details are stored in a materialized view called `amt_sold_by_state_mv`.

```
CREATE MATERIALIZED VIEW amt_sold_by_state_mv AS
SELECT country,
       state,
       APPROX_PERCENTILE_AGG(city_detail) state_detail
FROM amt_sold_by_city_mv
GROUP BY country, state;
```

The following statement is similar to the previous statement, except it creates aggregated details that contain the approximate monetary amounts for products sold to customers in each country. These aggregated details are stored in a materialized view called `amt_sold_by_country_mv`.

```
CREATE MATERIALIZED VIEW amt_sold_by_country_mv AS
  SELECT country,
         APPROX_PERCENTILE_AGG(city_detail) country_detail
  FROM amt_sold_by_city_mv
  GROUP BY country;
```

TO_APPROX_PERCENTILE: Examples

The following statement uses the `TO_APPROX_PERCENTILE` function to query the details stored in `amt_sold_by_city_mv` and return approximate 25th percentile, 50th percentile, and 75th percentile values for monetary amounts for products sold to customers in each city:

```
SELECT country,
       state,
       city,
       TO_APPROX_PERCENTILE(city_detail, .25, 'NUMBER') "25th Percentile",
       TO_APPROX_PERCENTILE(city_detail, .50, 'NUMBER') "50th Percentile",
       TO_APPROX_PERCENTILE(city_detail, .75, 'NUMBER') "75th Percentile"
FROM amt_sold_by_city_mv
ORDER BY country, state, city;

COUNTRY STATE        CITY           25th Percentile 50th Percentile 75th Percentile
------- ------------ -------------- --------------- --------------- ---------------
  52769 Kuala Lumpur Kuala Lumpur             19.29            38.1           53.84
  52769 Penang       Batu Ferringhi           21.51           42.09           57.26
  52769 Penang       Georgetown               19.15           33.25           56.12
  52769 Selangor     Klang                    18.08           32.06           51.29
  52769 Selangor     Petaling Jaya            19.29           35.43            60.2
. . .
```

The following statement uses the `TO_APPROX_PERCENTILE` function to query the details stored in `amt_sold_by_state_mv` and return approximate 25th percentile, 50th percentile, and 75th percentile values for monetary amounts for products sold to customers in each state:

```
SELECT country,
       state,
       TO_APPROX_PERCENTILE(state_detail, .25, 'NUMBER') "25th Percentile",
       TO_APPROX_PERCENTILE(state_detail, .50, 'NUMBER') "50th Percentile",
       TO_APPROX_PERCENTILE(state_detail, .75, 'NUMBER') "75th Percentile"
FROM amt_sold_by_state_mv
ORDER BY country, state;

COUNTRY STATE        25th Percentile 50th Percentile 75th Percentile
------- ------------ --------------- --------------- ---------------
  52769 Kuala Lumpur           19.29            38.1           53.84
  52769 Penang                 20.19           36.84           56.12
  52769 Selangor               16.97           32.41           52.69
  52770 Drenthe                16.76            31.7           53.89
  52770 Flevopolder            20.38           39.73           61.81
. . .
```

The following statement uses the `TO_APPROX_PERCENTILE` function to query the details stored in `amt_sold_by_country_mv` and return approximate 25th percentile, 50th percentile, and 75th percentile values for monetary amounts for products sold to customers in each country:

```
SELECT country,
       TO_APPROX_PERCENTILE(country_detail, .25, 'NUMBER') "25th Percentile",
       TO_APPROX_PERCENTILE(country_detail, .50, 'NUMBER') "50th Percentile",
       TO_APPROX_PERCENTILE(country_detail, .75, 'NUMBER') "75th Percentile"
FROM amt_sold_by_country_mv
ORDER BY country;
```

```
  COUNTRY 25th Percentile 50th Percentile 75th Percentile
--------- --------------- --------------- ---------------
    52769            19.1           35.43           52.78
    52770           19.29           38.99           59.58
    52771           11.99           44.99          561.47
    52772           18.08           33.72           54.16
    52773           15.67           29.61           50.65
. . .
```

`APPROX_PERCENTILE_AGG` takes as its input a column of details containing approximate percentile information, and enables you to perform aggregations of that information. The following statement demonstrates how approximate percentile details can interpreted by `APPROX_PERCENTILE_AGG` to provide an input to the `TO_APPROX_PERCENTILE` function. Like the previous example, this query returns approximate 25th percentile values for monetary amounts for products sold to customers in each country. Note that the results are identical to those returned for the 25th percentile in the previous example.

```
SELECT country,
       TO_APPROX_PERCENTILE(APPROX_PERCENTILE_AGG(city_detail), .25, 'NUMBER')
"25th Percentile"
FROM amt_sold_by_city_mv
GROUP BY country
ORDER BY country;

  COUNTRY 25th Percentile
---------- ---------------
     52769            19.1
     52770           19.29
     52771           11.99
     52772           18.08
     52773           15.67
. . .
```

Query Rewrite and Materialized Views Based on Approximate Queries: Example

In APPROX_PERCENTILE_DETAIL: Example, the `ENABLE QUERY REWRITE` clause is specified when creating the materialized view `amt_sold_by_city_mv`. This enables queries that contain approximation functions, such as `APPROX_MEDIAN` or `APPROX_PERCENTILE`, to be rewritten using the materialized view.

For example, ensure that query rewrite is enabled at either the database level or for the current session, and run the following query:

```
SELECT c.country_id country,
       APPROX_MEDIAN(s.amount_sold) amount_median
FROM customers c, sales s
WHERE c.cust_id = s.cust_id
GROUP BY c.country_id;
```

Explain the plan by querying `DBMS_XPLAN`:

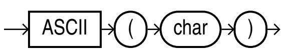
```
SET LINESIZE 300
SET PAGESIZE 0
COLUMN plan_table_output FORMAT A150

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format=>'BASIC'));
```

As shown in the following plan, the optimizer used the materialized view `amt_sold_by_city_mv` for the query:

```
EXPLAINED SQL STATEMENT:
------------------------
SELECT c.country_id country, APPROX_MEDIAN(s.amount_sold)
amount_median FROM customers c, sales s WHERE c.cust_id = s.cust_id
GROUP BY c.country_id

Plan hash value: 2232676046

-------------------------------------------------------------
| Id  | Operation                      | Name               |
-------------------------------------------------------------
|   0 | SELECT STATEMENT               |                    |
|   1 |  HASH GROUP BY APPROX          |                    |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| AMT_SOLD_BY_CITY_MV |
-------------------------------------------------------------
```

ASCII

Syntax



Purpose

`ASCII` returns the decimal representation in the database character set of the first character of *`char`*.

`char` can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is of data type `NUMBER`. If your database character set is 7-bit ASCII, then this function returns an ASCII value. If your database character set is EBCDIC Code, then this function returns an EBCDIC value. There is no corresponding EBCDIC character function.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> Data Type Comparison Rules for more information

Examples

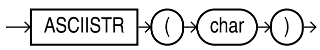
The following example returns employees whose last names begin with the letter L, whose ASCII equivalent is 76:

```
SELECT last_name
  FROM employees
  WHERE ASCII(SUBSTR(last_name, 1, 1)) = 76
  ORDER BY last_name;

LAST_NAME
```

```
-------------------------
Ladwig
Landry
Lee
Livingston
Lorentz
```

ASCIISTR

Syntax



Purpose

`ASCIISTR` takes as its argument a string, or an expression that resolves to a string, in any character set and returns an ASCII version of the string in the database character set. Non-ASCII characters are converted to the form `\xxxx`, where `xxxx` represents a UTF-16 code unit.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode character sets and character semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `ASCIISTR`

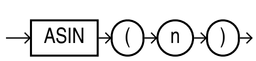
Examples

The following example returns the ASCII string equivalent of the text string "`ABÄCDE`":

```
SELECT ASCIISTR('ABÄCDE')
  FROM DUAL;

ASCIISTR('
----------
AB\00C4CDE
```

ASIN

Syntax



Purpose

`ASIN` returns the arc sine of *n*. The argument *n* must be in the range of -1 to 1, and the function returns a value in the range of -*pi*/2 to *pi*/2, expressed in radians.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

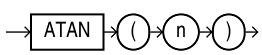
Examples

The following example returns the arc sine of .3:

```
SELECT ASIN(.3) "Arc_Sine"
  FROM DUAL;

 Arc_Sine
----------
.304692654
```

ATAN

Syntax



Purpose

`ATAN` returns the arc tangent of *n*. The argument *n* can be in an unbounded range and returns a value in the range of -*pi*/2 to *pi*/2, expressed in radians.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> ATAN2 for information about the `ATAN2` function and Table 2-8 for more information on implicit conversion

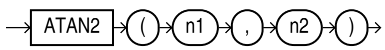
Examples

The following example returns the arc tangent of .3:

```
SELECT ATAN(.3) "Arc_Tangent"
  FROM DUAL;

Arc_Tangent
----------
.291456794
```

ATAN2

Syntax



Purpose

`ATAN2` returns the arc tangent of *`n1`* and *`n2`*. The argument *`n1`* can be in an unbounded range and returns a value in the range of -*pi* to *pi*, depending on the signs of *`n1`* and *`n2`*, expressed in radians.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If any argument is `BINARY_FLOAT` or `BINARY_DOUBLE`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns `NUMBER`.

> **See Also:**
>
> ATAN for information on the `ATAN` function and Table 2-8 for more information on implicit conversion

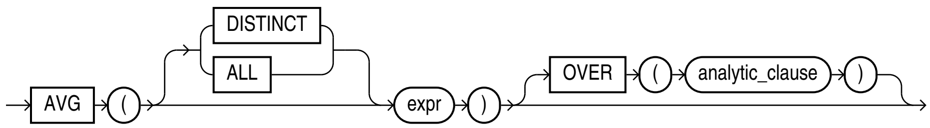
Examples

The following example returns the arc tangent of .3 and .2:

```
SELECT ATAN2(.3, .2) "Arc_Tangent2"
  FROM DUAL;

Arc_Tangent2
------------
  .982793723
```

AVG

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`AVG` returns average value of *`expr`*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

> **See Also:**
>
> About SQL Expressions for information on valid forms of *`expr`* and Aggregate Functions

Aggregate Example

The following example calculates the average salary of all employees in the `hr.employees` table:

```
SELECT AVG(salary) "Average"
  FROM employees;

       Average
--------------
    6461.83178
```

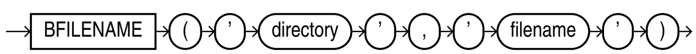
Analytic Example

The following example calculates, for each employee in the `employees` table, the average salary of the employees reporting to the same manager who were hired in the range just before through just after the employee:

```
SELECT manager_id, last_name, hire_date, salary,
       AVG(salary) OVER (PARTITION BY manager_id ORDER BY hire_date
  ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS c_mavg
  FROM employees
  ORDER BY manager_id, hire_date, salary;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     C_MAVG
---------- ------------------------- --------- ---------- ----------
       100 De Haan                   13-JAN-01      17000      14000
       100 Raphaely                  07-DEC-02      11000 11966.6667
       100 Kaufling                  01-MAY-03       7900 10633.3333
       100 Hartstein                 17-FEB-04      13000 9633.33333
       100 Weiss                     18-JUL-04       8000 11666.6667
       100 Russell                   01-OCT-04      14000 11833.3333
       100 Partners                  05-JAN-05      13500 13166.6667
       100 Errazuriz                 10-MAR-05      12000 11233.3333
. . .
```

BFILENAME

Syntax



Purpose

`BFILENAME` returns a `BFILE` locator that is associated with a physical LOB binary file on the server file system.

- '*`directory`*' is a database object that serves as an alias for a full path name on the server file system where the files are actually located.
- '*`filename`*' is the name of the file in the server file system.

You must create the directory object and associate a `BFILE` value with a physical file before you can use them as arguments to `BFILENAME` in a SQL or PL/SQL statement, `DBMS_LOB` package, or OCI operation.

You can use this function in two ways:

- In a DML statement to initialize a `BFILE` column
- In a programmatic interface to access `BFILE` data by assigning a value to the `BFILE` locator

The directory argument is case sensitive. You must ensure that you specify the directory object name exactly as it exists in the data dictionary. For example, if an `"Admin"` directory object was created using mixed case and a quoted identifier in the `CREATE DIRECTORY` statement, then when using the `BFILENAME` function you must refer to the directory object as 'Admin'. You must specify the filename argument according to the case and punctuation conventions for your operating system.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* and *Oracle Call Interface Programmer's Guide* for more information on LOBs and for examples of retrieving BFILE data
> - CREATE DIRECTORY

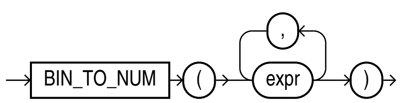
Examples

The following example inserts a row into the sample table pm.print_media. The example uses the BFILENAME function to identify a binary file on the server file system in the directory /demo/schema/product_media. The example shows how the directory database object media_dir was created in the pm schema.

```
CREATE DIRECTORY media_dir AS '/demo/schema/product_media';

INSERT INTO print_media (product_id, ad_id, ad_graphic)
  VALUES (3000, 31001, BFILENAME('MEDIA_DIR', 'modem_comp_ad.gif'));
```

BIN_TO_NUM

Syntax



Purpose

BIN_TO_NUM converts a bit vector to its equivalent number. Each argument to this function represents a bit in the bit vector. This function takes as arguments any numeric data type, or any nonnumeric data type that can be implicitly converted to NUMBER. Each *expr* must evaluate to 0 or 1. This function returns Oracle NUMBER.

BIN_TO_NUM is useful in data warehousing applications for selecting groups of interest from a materialized view using grouping sets.

> **See Also:**
>
> - *group_by_clause* for information on GROUPING SETS syntax
> - Table 2-8 for more information on implicit conversion
> - *Oracle Database Data Warehousing Guide* for information on data aggregation in general

Examples

The following example converts a binary value to a number:

```
SELECT BIN_TO_NUM(1,0,1,0)
  FROM DUAL;

BIN_TO_NUM(1,0,1,0)
-------------------
                 10
```

The next example converts three values into a single binary value and uses `BIN_TO_NUM` to convert that binary into a number. The example uses a PL/SQL declaration to specify the original values. These would normally be derived from actual data sources.

```
SELECT order_status
  FROM orders
  WHERE order_id = 2441;

ORDER_STATUS
------------
           5
DECLARE
  warehouse NUMBER := 1;
  ground    NUMBER := 1;
  insured   NUMBER := 1;
  result    NUMBER;
BEGIN
  SELECT BIN_TO_NUM(warehouse, ground, insured) INTO result FROM DUAL;
  UPDATE orders SET order_status = result WHERE order_id = 2441;
END;
/
PL/SQL procedure successfully completed.

SELECT order_status
  FROM orders
  WHERE order_id = 2441;

ORDER_STATUS
------------
           7
```

Refer to the examples for BITAND for information on reversing this process by extracting multiple values from a single column value.

BITAND

Syntax

Purpose

The `BITAND` function treats its inputs and its output as vectors of bits; the output is the bitwise `AND` of the inputs.

The types of *expr1* and *expr2* are NUMBER, and the result is of type NUMBER. If either argument to BITAND is NULL, the result is NULL.

The arguments must be in the range $-(2^{(n-1)})$ .. $((2^{(n-1)})-1)$. If an argument is out of this range, the result is undefined.

The result is computed in several steps. First, each argument A is replaced with the value SIGN(A)*FLOOR(ABS(A)). This conversion has the effect of truncating each argument towards zero. Next, each argument A (which must now be an integer value) is converted to an n-bit two's complement binary integer value. The two bit values are combined using a bitwise AND operation. Finally, the resulting n-bit two's complement value is converted back to NUMBER.

Notes on the BITAND Function

- The current implementation of BITAND defines *n* = 128.
- PL/SQL supports an overload of BITAND for which the types of the inputs and of the result are all BINARY_INTEGER and for which *n* = 32.

Examples

The following example performs an AND operation on the numbers 6 (binary 1,1,0) and 3 (binary 0,1,1):

```
SELECT BITAND(6,3)
  FROM DUAL;

BITAND(6,3)
-----------
          2
```

This is the same as the following example, which shows the binary values of 6 and 3. The BITAND function operates only on the significant digits of the binary values:

```
SELECT BITAND(
    BIN_TO_NUM(1,1,0),
    BIN_TO_NUM(0,1,1)) "Binary"
  FROM DUAL;

    Binary
----------
         2
```

Refer to the example for BIN_TO_NUM for information on encoding multiple values in a single column value.

The following example supposes that the *order_status* column of the sample table oe.orders encodes several choices as individual bits within a single numeric value. For example, an order still in the warehouse is represented by a binary value 001 (decimal 1). An order being sent by ground transportation is represented by a binary value 010 (decimal 2). An insured package is represented by a binary value 100 (decimal 4). The example uses the DECODE function to provide two values for each of the three bits in the *order_status* value, one value if the bit is turned on and one if it is turned off.

```
SELECT order_id, customer_id, order_status,
    DECODE(BITAND(order_status, 1), 1, 'Warehouse', 'PostOffice') "Location",
    DECODE(BITAND(order_status, 2), 2, 'Ground', 'Air') "Method",
    DECODE(BITAND(order_status, 4), 4, 'Insured', 'Certified') "Receipt"
  FROM orders
  WHERE sales_rep_id = 160
```

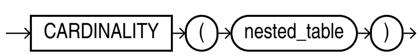
```
  ORDER BY order_id;

  ORDER_ID CUSTOMER_ID ORDER_STATUS Location   Method Receipt
---------- ----------- ------------ ---------- ------ ---------
      2416         104            6 PostOffice Ground Insured
      2419         107            3 Warehouse  Ground Certified
      2420         108            2 PostOffice Ground Certified
      2423         145            3 Warehouse  Ground Certified
      2441         106            5 Warehouse  Air    Insured
      2455         145            7 Warehouse  Ground Insured
```

For the `Location` column, `BITAND` first compares *`order_status`* with 1 (binary 001). Only significant bit values are compared, so any binary value with a 1 in its rightmost bit (any odd number) will evaluate positively and return 1. Even numbers will return 0. The `DECODE` function compares the value returned by `BITAND` with 1. If they are both 1, then the location is "Warehouse". If they are different, then the location is "PostOffice".

The `Method` and `Receipt` columns are calculated similarly. For `Method`, `BITAND` performs the `AND` operation on *`order_status`* and 2 (binary 010). For `Receipt`, `BITAND` performs the `AND` operation on *`order_status`* and 4 (binary 100).

CARDINALITY

Syntax



Purpose

`CARDINALITY` returns the number of elements in a nested table. The return type is `NUMBER`. If the nested table is empty, or is a null collection, then `CARDINALITY` returns `NULL`.

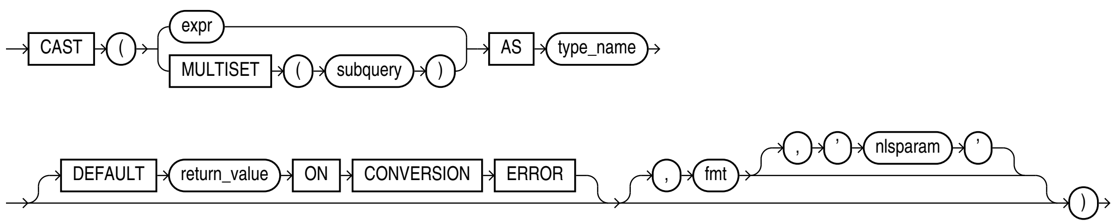
Examples

The following example shows the number of elements in the nested table column `ad_textdocs_ntab` of the sample table `pm.print_media`:

```
SELECT product_id, CARDINALITY(ad_textdocs_ntab) cardinality
  FROM print_media
  ORDER BY product_id;

PRODUCT_ID CARDINALITY
---------- -----------
      2056           3
      2268           3
      3060           3
      3106           3
```

CAST

Syntax



Purpose

CAST lets you convert built-in data types or collection-typed values of one type into another built-in data type or collection type. You can cast an unnamed operand (such as a date or the result set of a subquery) or a named collection (such as a varray or a nested table) into a type-compatible data type or named collection. The *type_name* must be the name of a built-in data type or collection type and the operand must be a built-in data type or must evaluate to a collection value.

For the operand, *expr* can be either a built-in data type, a collection type, or an instance of an ANYDATA type. If *expr* is an instance of an ANYDATA type, then CAST tries to extract the value of the ANYDATA instance and return it if it matches the cast target type, otherwise, null will be returned. MULTISET informs Oracle Database to take the result set of the subquery and return a collection value. Table 7-1 shows which built-in data types can be cast into which other built-in data types. (CAST does not support LONG, LONG RAW, or the Oracle-supplied types.)

CAST does not directly support any of the LOB data types. When you use CAST to convert a CLOB value into a character data type or a BLOB value into the RAW data type, the database implicitly converts the LOB value to character or raw data and then explicitly casts the resulting value into the target data type. If the resulting value is larger than the target type, then the database returns an error.

When you use CAST ... MULTISET to get a collection value, each select list item in the query passed to the CAST function is converted to the corresponding attribute type of the target collection element type.

Table 7-1 Casting Built-In Data Types

Destination Data Type	from BINARY_FLOAT, BINARY_DOUBLE	from CHAR, VARCHAR2	from NUMBER/ INTEGER	from DATETIME / INTERVAL (Note 1)	from RAW	from ROWID, UROWID (Note 2)	from NCHAR, NVARCHAR2
to BINARY_FLOAT, BINARY_DOUBLE	X (Note 3)	X (Note 3)	X (Note 3)	--	--	--	X (Note 3)

Table 7-1 (Cont.) Casting Built-In Data Types

Destination Data Type	from BINARY_FLOAT, BINARY_DOUBLE	from CHAR, VARCHAR2	from NUMBER/ INTEGER	from DATETIME / INTERVAL (Note 1)	from RAW	from ROWID, UROWID (Note 2)	from NCHAR, NVARCHAR2
to CHAR, VARCHAR2	X	X	X	X	X	X	--
to NUMBER/ INTEGER	X (Note 3)	X (Note 3)	X (Note 3)	--	--	--	X (Note 3)
to DATETIME/ INTERVAL	--	X (Note 3)	--	X (Note 3)	--	--	--
to RAW	--	X	--	--	X	--	--
to ROWID, UROWID	--	X	--	--	--	X	--
to NCHAR, NVARCHAR2	X	--	X	X	X	X	X

Note 1: Datetime/interval includes `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL DAY TO SECOND`, and `INTERVAL YEAR TO MONTH`.

Note 2: You cannot cast a `UROWID` to a `ROWID` if the `UROWID` contains the value of a `ROWID` of an index-organized table.

Note 3: You can specify the `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause for this type of conversion. You can specify the *`fmt`* and *`nlsparam`* clauses for this type of conversion with the following exceptions: you cannot specify fmt when converting to `INTERVAL DAY TO SECOND`, and you cannot specify *`fmt`* or *`nlsparam`* when converting to `INTERVAL YEAR TO MONTH`.

If you want to cast a named collection type into another named collection type, then the elements of both collections must be of the same type.

See Also:

- Implicit Data Conversion for information on how Oracle Database implicitly converts collection type data into character data and Security Considerations for Data Conversion
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `CAST` when it is a character value

MULTISET

If the result set of *`subquery`* can evaluate to multiple rows, then you must specify the `MULTISET` keyword. The rows resulting from the subquery form the elements of the collection value into which they are cast. Without the `MULTISET` keyword, the subquery is treated as a scalar subquery.

Restriction on MULTISET

If you specify the `MULTISET` keyword, then you cannot specify the `DEFAULT` *`return_value`* `ON CONVERSION ERROR`, *`fmt`*, or *`nlsparam`* clauses.

DEFAULT *return_value* ON CONVERSION ERROR

This clause allows you to specify the value returned by this function if an error occurs while converting *`expr`* to *`type_name`*. This clause has no effect if an error occurs while evaluating *`expr`*.

This clause is valid if *`expr`* evaluates to a character string of type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`, and *`type_name`* is `BINARY_DOUBLE`, `BINARY_FLOAT`, `DATE`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `NUMBER`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, or `TIMESTAMP WITH LOCAL TIME ZONE`.

The *`return_value`* can be a string literal, null, constant expression, or a bind variable, and must evaluate to null or a character string of type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. If *`return_value`* cannot be converted to *`type_name`*, then the function returns an error.

fmt and *nlsparam*

The *`fmt`* argument lets you specify a format model and the *`nlsparam`* argument lets you specify NLS parameters. If you specify these arguments, then they are applied when converting *`expr`* and *`return_value`*, if specified, to *`type_name`*.

You can specify *`fmt`* and *`nlsparam`* if *`type_name`* is one of the following data types:

- `BINARY_DOUBLE`

  If you specify `BINARY_DOUBLE`, then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_BINARY_DOUBLE` function. Refer to TO_BINARY_DOUBLE for more information.

- `BINARY_FLOAT`

  If you specify `BINARY_FLOAT`, then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_BINARY_FLOAT` function. Refer to TO_BINARY_FLOAT for more information.

- `DATE`

  If you specify `DATE`, then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_DATE` function. Refer to TO_DATE for more information.

- `NUMBER`

  If you specify `NUMBER`, then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_NUMBER` function. Refer to TO_NUMBER for more information.

- `TIMESTAMP`

  If you specify `TIMESTAMP`, then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_TIMESTAMP` function. If you omit *`fmt`*, then *`expr`* must be in the default format of the `TIMESTAMP` data type, which is determined explicitly by the `NLS_TIMESTAMP_FORMAT` parameter or implicitly by the `NLS_TERRITORY` parameter. Refer to TO_TIMESTAMP for more information.

- `TIMESTAMP WITH TIME ZONE`

If you specify `TIMESTAMP WITH TIME ZONE`, then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_TIMESTAMP_TZ` function. If you omit *`fmt`*, then *`expr`* must be in the default format of the `TIMESTAMP WITH TIME ZONE` data type, which is determined explicitly by the `NLS_TIMESTAMP_TZ_FORMAT` parameter or implicitly by the `NLS_TERRITORY` parameter. Refer to TO_TIMESTAMP_TZ for more information.

- `TIMESTAMP WITH LOCAL TIME ZONE`

  If you specify `TIMESTAMP WITH LOCAL TIME ZONE` then the optional *`fmt`* and *`nlsparam`* arguments serve the same purpose as for the `TO_TIMESTAMP` function. If you omit *`fmt`*, then *`expr`* must be in the default format of the `TIMESTAMP` data type, , which is determined explicitly by the `NLS_TIMESTAMP_FORMAT` parameter or implicitly by the `NLS_TERRITORY` parameter. Refer to TO_TIMESTAMP for more information.

Built-In Data Type Examples

The following examples use the `CAST` function with scalar data types. The first example converts text to a timestamp value by applying the format model provided in the session parameter `NLS_TIMESTAMP_FORMAT`. If you want to avoid dependency on this NLS parameter, then you can use the `TO_DATE` as shown in the second example.

```
SELECT CAST('22-OCT-1997'
       AS TIMESTAMP WITH LOCAL TIME ZONE)
  FROM DUAL;

SELECT CAST(TO_DATE('22-Oct-1997', 'DD-Mon-YYYY')
       AS TIMESTAMP WITH LOCAL TIME ZONE)
  FROM DUAL;
```

In the preceding example, `TO_DATE` converts from text to `DATE`, and `CAST` converts from `DATE` to `TIMESTAMP WITH LOCAL TIME ZONE`, interpreting the date in the session time zone (`SESSIONTIMEZONE`).

```
SELECT product_id, CAST(ad_sourcetext AS VARCHAR2(30)) text
  FROM print_media
  ORDER BY product_id;
```

The following examples return a default value if an error occurs while converting the specified value to the specified data type. In these examples, the conversions occurs without error.

```
SELECT CAST(200
       AS NUMBER
       DEFAULT 0 ON CONVERSION ERROR)
  FROM DUAL;

SELECT CAST('January 15, 1989, 11:00 A.M.'
       AS DATE
       DEFAULT NULL ON CONVERSION ERROR,
       'Month dd, YYYY, HH:MI A.M.')
  FROM DUAL;

SELECT CAST('1999-12-01 11:00:00 -8:00'
       AS TIMESTAMP WITH TIME ZONE
       DEFAULT '2000-01-01 01:00:00 -8:00' ON CONVERSION ERROR,
       'YYYY-MM-DD HH:MI:SS TZH:TZM',
       'NLS_DATE_LANGUAGE = American')
  FROM DUAL;
```

In the following example, an error occurs while converting 'N/A' to a NUMBER value. Therefore, the CAST function returns the default value of 0.

```
SELECT CAST('N/A'
       AS NUMBER
       DEFAULT '0' ON CONVERSION ERROR)
  FROM DUAL;
```

Collection Examples

The CAST examples that follow build on the cust_address_typ found in the sample order entry schema, oe.

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;
/
CREATE TYPE address_array_t AS VARRAY(3) OF cust_address_typ;
/
CREATE TABLE cust_address (
  custno            NUMBER,
  street_address    VARCHAR2(40),
  postal_code       VARCHAR2(10),
  city              VARCHAR2(30),
  state_province    VARCHAR2(10),
  country_id        CHAR(2));

CREATE TABLE cust_short (custno NUMBER, name VARCHAR2(31));

CREATE TABLE states (state_id NUMBER, addresses address_array_t);
```

This example casts a subquery:

```
SELECT s.custno, s.name,
       CAST(MULTISET(SELECT ca.street_address,
                            ca.postal_code,
                            ca.city,
                            ca.state_province,
                            ca.country_id
                       FROM cust_address ca
                       WHERE s.custno = ca.custno)
       AS address_book_t)
  FROM cust_short s
  ORDER BY s.custno;
```

CAST converts a varray type column into a nested table:

```
SELECT CAST(s.addresses AS address_book_t)
  FROM states s
  WHERE s.state_id = 111;
```

The following objects create the basis of the example that follows:

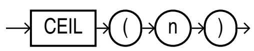
```
CREATE TABLE projects
  (employee_id NUMBER, project_name VARCHAR2(10));

CREATE TABLE emps_short
  (employee_id NUMBER, last_name VARCHAR2(10));

CREATE TYPE project_table_typ AS TABLE OF VARCHAR2(10);
/
```

The following example of a MULTISET expression uses these objects:

```
SELECT e.last_name,
       CAST(MULTISET(SELECT p.project_name
                       FROM projects p
                       WHERE p.employee_id = e.employee_id
                       ORDER BY p.project_name)
       AS project_table_typ)
  FROM emps_short e
  ORDER BY e.last_name;
```

CEIL

Syntax



Purpose

`CEIL` returns the smallest integer that is greater than or equal to n. The number n can always be written as the difference of an integer k and a positive fraction f such that $0 \le f < 1$ and $n = k - f$. The value of `CEIL` is the integer k. Thus, the value of `CEIL` is n itself if and only if n is precisely an integer.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and FLOOR

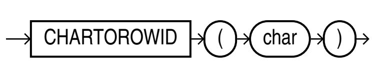
Examples

The following example returns the smallest integer greater than or equal to the order total of a specified order:

```
SELECT order_total, CEIL(order_total)
  FROM orders
  WHERE order_id = 2434;

ORDER_TOTAL CEIL(ORDER_TOTAL)
----------- -----------------
   268651.8            268652
```

CHARTOROWID

Syntax



Purpose

`CHARTOROWID` converts a value from `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to `ROWID` data type.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> Data Type Comparison Rules for more information.

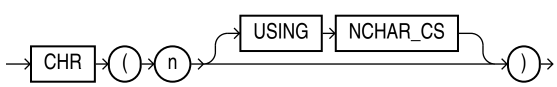
Examples

The following example converts a character rowid representation to a rowid. (The actual rowid is different for each database instance.)

```
SELECT last_name
  FROM employees
  WHERE ROWID = CHARTOROWID('AAAFd1AAFAAAABSAA/');

LAST_NAME
-------------------------
Greene
```

CHR

Syntax



Purpose

`CHR` returns the character having the binary equivalent to *n* as a `VARCHAR2` value in either the database character set or, if you specify `USING NCHAR_CS`, the national character set.

For single-byte character sets, if `n` > 256, then Oracle Database returns the binary equivalent of `n mod 256`. For multibyte character sets, `n` must resolve to one entire code point. Invalid code points are not validated, and the result of specifying invalid code points is indeterminate.

This function takes as an argument a `NUMBER` value, or any value that can be implicitly converted to `NUMBER`, and returns a character.

> **Note:**
>
> Use of the `CHR` function (either with or without the optional `USING NCHAR_CS` clause) results in code that is not portable between ASCII- and EBCDIC-based machine architectures.

> **See Also:**
>
> - NCHR and Table 2-8 for more information on implicit conversion
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `CHR`

Examples

The following example is run on an ASCII-based machine with the database character set defined as WE8ISO8859P1:

```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog"
  FROM DUAL;

Dog
---
CAT
```

To produce the same results on an EBCDIC-based machine with the WE8EBCDIC1047 character set, the preceding example would have to be modified as follows:

```
SELECT CHR(195)||CHR(193)||CHR(227) "Dog"
  FROM DUAL;

Dog
---
CAT
```

For multibyte character sets, this sort of concatenation gives different results. For example, given a multibyte character whose hexadecimal value is `a1a2` (`a1` representing the first byte and `a2` the second byte), you must specify for `n` the decimal equivalent of '`a1a2`', or 41378:

```
SELECT CHR(41378)
  FROM DUAL;
```

You cannot specify the decimal equivalent of a1 concatenated with the decimal equivalent of a2, as in the following example:

```
SELECT CHR(161)||CHR(162)
  FROM DUAL;
```

However, you can concatenate whole multibyte code points, as in the following example, which concatenates the multibyte characters whose hexadecimal values are `a1a2` and `a1a3`:

```
SELECT CHR(41378)||CHR(41379)
  FROM DUAL;
```

The following example assumes that the national character set is UTF16:

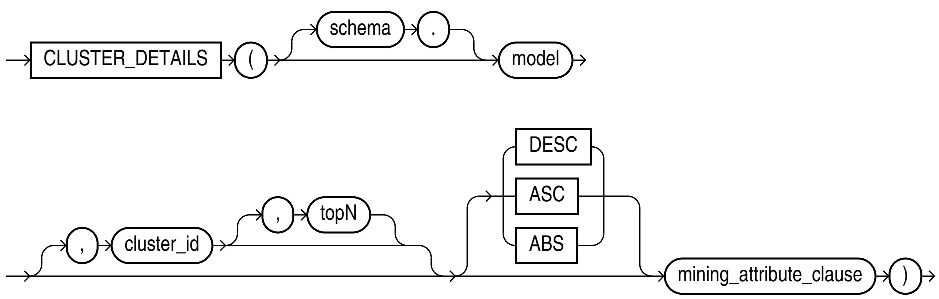
```
SELECT CHR (196 USING NCHAR_CS)
  FROM DUAL;
```

CH
--
Ä

CLUSTER_DETAILS

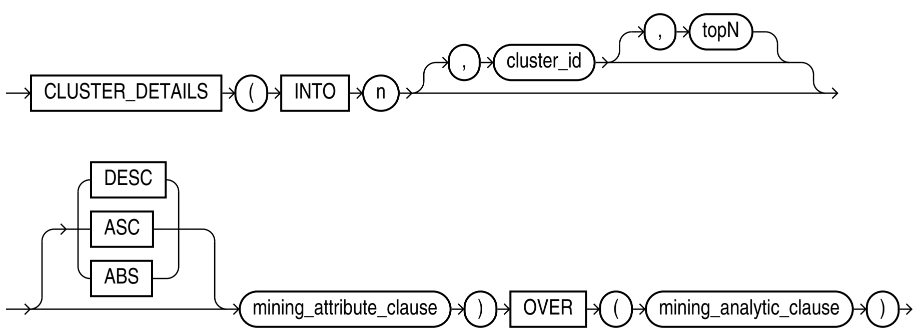
Syntax

***cluster_details*::=**

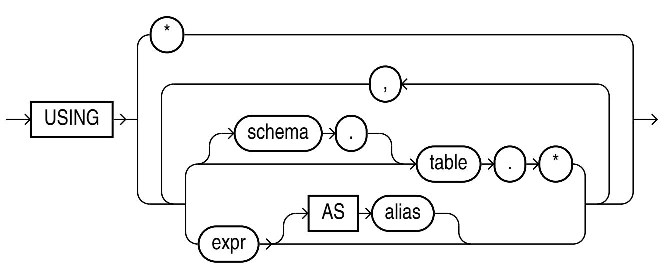


Analytic Syntax

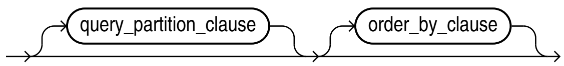
***cluster_details_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

CLUSTER_DETAILS returns cluster details for each row in the selection. The return value is an XML string that describes the attributes of the highest probability cluster or the specified *cluster_id*.

topN

If you specify a value for *topN*, the function returns the *N* attributes that most influence the cluster assignment (the score). If you do not specify *topN*, the function returns the 5 most influential attributes.

DESC, ASC, or ABS

The returned attributes are ordered by weight. The weight of an attribute expresses its positive or negative impact on cluster assignment. A positive weight indicates an increased likelihood of assignment. A negative weight indicates a decreased likelihood of assignment.

By default, CLUSTER_DETAILS returns the attributes with the highest positive weights (DESC). If you specify ASC, the attributes with the highest negative weights are returned. If you specify ABS, the attributes with the greatest weights, whether negative or positive, are returned. The results are ordered by absolute value from highest to lowest. Attributes with a zero weight are not included in the output.

Syntax Choice

CLUSTER_DETAILS can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the `CLUSTER_DETAILS` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about clustering.

> **Note:**
>
> The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

This example lists the attributes that have the greatest impact (more that 20% probability) on cluster assignment for customer ID 100955. The query invokes the `CLUSTER_DETAILS` and `CLUSTER_SET` functions, which apply the clustering model `em_sh_clus_sample`.

```
SELECT S.cluster_id, probability prob,
       CLUSTER_DETAILS(em_sh_clus_sample, S.cluster_id, 5 USING T.*) det
FROM
  (SELECT v.*, CLUSTER_SET(em_sh_clus_sample, NULL, 0.2 USING *) pset
    FROM mining_data_apply_v v
   WHERE cust_id = 100955) T,
  TABLE(T.pset) S
ORDER BY 2 DESC;

CLUSTER_ID  PROB DET
---------- ----- --------------------------------------------------------------------------------
        14 .6761 <Details algorithm="Expectation Maximization" cluster="14">
                 <Attribute name="AGE" actualValue="51" weight=".676" rank="1"/>
                 <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".557" rank="2"/>
                 <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".412" rank="3"/>
                 <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".171" rank="4"/>
                 <Attribute name="BOOKKEEPING_APPLICATION" actualValue="1" weight="-.003"rank="5"/>
                 </Details>

         3 .3227 <Details algorithm="Expectation Maximization" cluster="3">
                 <Attribute name="YRS_RESIDENCE" actualValue="3" weight=".323" rank="1"/>
                 <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".265" rank="2"/>
                 <Attribute name="EDUCATION" actualValue="HS-grad" weight=".172" rank="3"/>
                 <Attribute name="AFFINITY_CARD" actualValue="0" weight=".125" rank="4"/>
```

```
<Attribute name="OCCUPATION" actualValue="Crafts" weight=".055" rank="5"/>
</Details>
```

Analytic Example

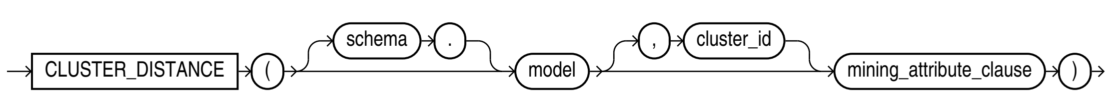
This example divides the customer database into four segments based on common characteristics. The clustering functions compute the clusters and return the score without a predefined clustering model.

```
SELECT * FROM (
     SELECT cust_id,
          CLUSTER_ID(INTO 4 USING *) OVER () cls,
          CLUSTER_DETAILS(INTO 4 USING *) OVER () cls_details
     FROM mining_data_apply_v)
WHERE cust_id <= 100003
ORDER BY 1;

CUST_ID CLS CLS_DETAILS
------- --- ------------------------------------------------------------------------------------
 100001   5 <Details algorithm="K-Means Clustering" cluster="5">
            <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".349" rank="1"/>
            <Attribute name="BULK_PACK_DISKETTES" actualValue="0" weight=".33" rank="2"/>
            <Attribute name="CUST_INCOME_LEVEL" actualValue="G: 130\,000 - 149\,999"
weight=".291"
             rank="3"/>
            <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".268" rank="4"/>
            <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".179" rank="5"/>
            </Details>

 100002   6 <Details algorithm="K-Means Clustering" cluster="6">
            <Attribute name="CUST_GENDER" actualValue="F" weight=".945" rank="1"/>
            <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".856" rank="2"/>
            <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".468" rank="3"/>
            <Attribute name="AFFINITY_CARD" actualValue="0" weight=".012" rank="4"/>
            <Attribute name="CUST_INCOME_LEVEL" actualValue="L: 300\,000 and above" weight=".009"
             rank="5"/>
            </Details>

 100003   7 <Details algorithm="K-Means Clustering" cluster="7">
            <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".862" rank="1"/>
            <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".423" rank="2"/>
            <Attribute name="HOME_THEATER_PACKAGE" actualValue="0" weight=".113" rank="3"/>
            <Attribute name="AFFINITY_CARD" actualValue="0" weight=".007" rank="4"/>
            <Attribute name="CUST_ID" actualValue="100003" weight=".006" rank="5"/>
            </Details>
```

CLUSTER_DISTANCE

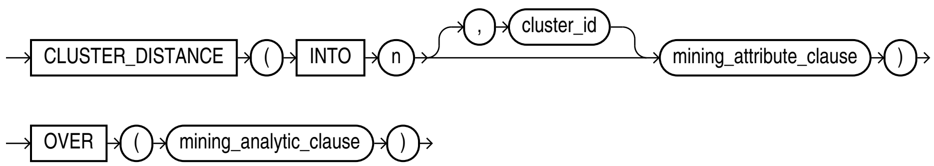
Syntax

***cluster_distance*::=**

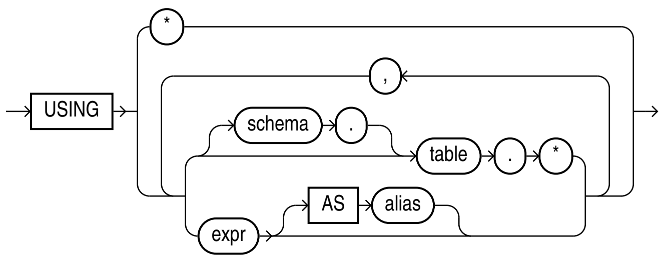


Analytic Syntax

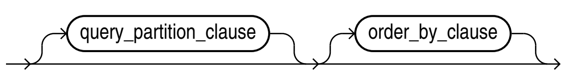
***cluster_distance_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of `mining_analytic_clause`

Purpose

`CLUSTER_DISTANCE` returns a cluster distance for each row in the selection. The cluster distance is the distance between the row and the centroid of the highest probability cluster or the specified `cluster_id`. The distance is returned as `BINARY_DOUBLE`.

Syntax Choice

`CLUSTER_DISTANCE` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.

- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the `CLUSTER_DISTANCE` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, this data is also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about clustering.

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

This example finds the 10 rows that are most anomalous as measured by their distance from their nearest cluster centroid.

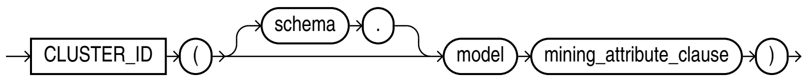
```
SELECT cust_id
  FROM (
    SELECT cust_id,
           rank() over
             (order by CLUSTER_DISTANCE(km_sh_clus_sample USING *) desc) rnk
      FROM mining_data_apply_v)
  WHERE rnk <= 11
  ORDER BY rnk;

   CUST_ID
----------
    100579
    100050
    100329
    100962
    101251
    100179
    100382
    100713
    100629
```

```
100787
101478
```

CLUSTER_ID

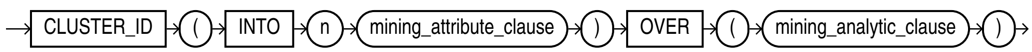
Syntax

***cluster_id*::=**

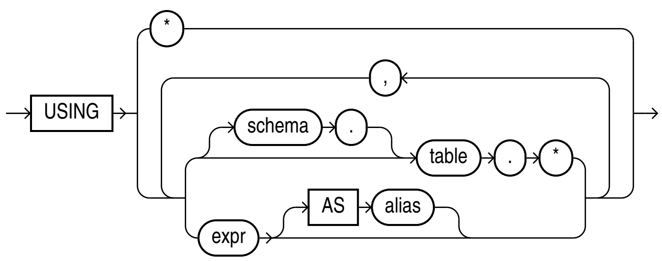


Analytic Syntax

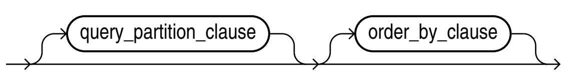
***cluster_id_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *`mining_analytic_clause`*

Purpose

`CLUSTER_ID` returns the identifier of the highest probability cluster for each row in the selection. The cluster identifier is returned as an Oracle `NUMBER`.

Syntax Choice

CLUSTER_ID can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the CLUSTER_ID function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See *mining_attribute_clause*::=.)

See Also:

- *Oracle Data Mining User's Guide* for information about scoring.
- *Oracle Data Mining Concepts* for information about clustering.

Note:

The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

The following example lists the clusters into which the customers in mining_data_apply_v have been grouped.

```
SELECT CLUSTER_ID(km_sh_clus_sample USING *) AS clus, COUNT(*) AS cnt
  FROM mining_data_apply_v
  GROUP BY CLUSTER_ID(km_sh_clus_sample USING *)
  ORDER BY cnt DESC;

      CLUS        CNT
---------- ----------
         2        580
        10        216
```

```
         6        186
         8        115
        19        110
        12        101
        18         81
        16         39
        17         38
        14         34
```

Analytic Example

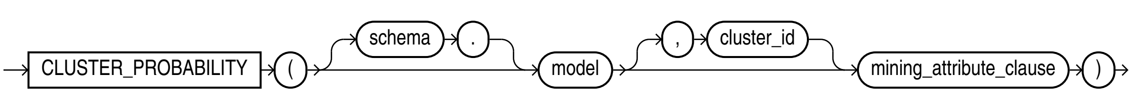
This example divides the customer database into four segments based on common characteristics. The clustering functions compute the clusters and return the score without a predefined clustering model.

```
SELECT * FROM (
     SELECT cust_id,
          CLUSTER_ID(INTO 4 USING *) OVER () cls,
          CLUSTER_DETAILS(INTO 4 USING *) OVER () cls_details
     FROM mining_data_apply_v)
WHERE cust_id <= 100003
ORDER BY 1;

CUST_ID CLS CLS_DETAILS
------- --- ------------------------------------------------------------------------------
 100001   5 <Details algorithm="K-Means Clustering" cluster="5">
            <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".349" rank="1"/>
            <Attribute name="BULK_PACK_DISKETTES" actualValue="0" weight=".33" rank="2"/>
            <Attribute name="CUST_INCOME_LEVEL" actualValue="G: 130\,000 - 149\,999"
               weight=".291" rank="3"/>
            <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".268" rank="4"/>
            <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".179" rank="5"/>
            </Details>

 100002   6 <Details algorithm="K-Means Clustering" cluster="6">
            <Attribute name="CUST_GENDER" actualValue="F" weight=".945" rank="1"/>
            <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".856" rank="2"/>
            <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".468" rank="3"/>
            <Attribute name="AFFINITY_CARD" actualValue="0" weight=".012" rank="4"/>
            <Attribute name="CUST_INCOME_LEVEL" actualValue="L: 300\,000 and above"
               weight=".009" rank="5"/>
            </Details>

 100003   7 <Details algorithm="K-Means Clustering" cluster="7">
            <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".862" rank="1"/>
            <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".423" rank="2"/>
            <Attribute name="HOME_THEATER_PACKAGE" actualValue="0" weight=".113" rank="3"/>
            <Attribute name="AFFINITY_CARD" actualValue="0" weight=".007" rank="4"/>
            <Attribute name="CUST_ID" actualValue="100003" weight=".006" rank="5"/>
            </Details>
```

CLUSTER_PROBABILITY

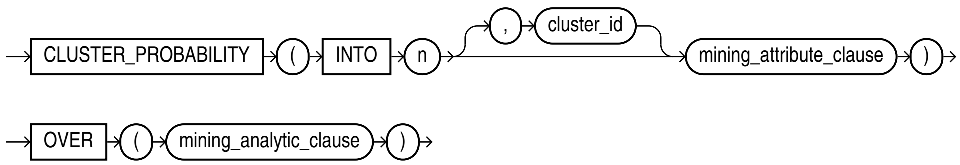
Syntax

***cluster_probability*::=**

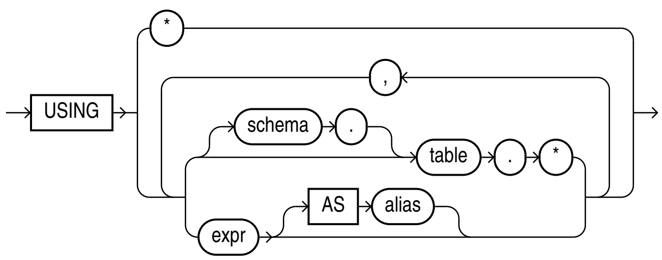


Analytic Syntax

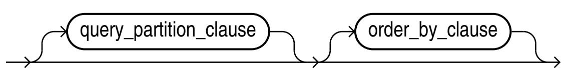
***cluster_prob_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`CLUSTER_PROBABILITY` returns a probability for each row in the selection. The probability refers to the highest probability cluster or to the specified *cluster_id*. The cluster probability is returned as `BINARY_DOUBLE`.

Syntax Choice

`CLUSTER_PROBABILITY` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.

- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *n*, where *n* is the number of clusters to compute, and mining_analytic_clause, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause::=*.)

The syntax of the `CLUSTER_PROBABILITY` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See *mining_attribute_clause::=*.)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about clustering.

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

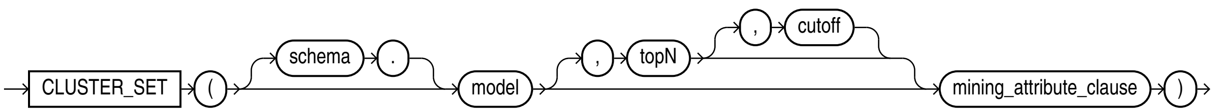
The following example lists the ten most representative customers, based on likelihood, of cluster 2.

```
SELECT cust_id
  FROM (SELECT cust_id, rank() OVER (ORDER BY prob DESC, cust_id) rnk_clus2
    FROM (SELECT cust_id, CLUSTER_PROBABILITY(km_sh_clus_sample, 2 USING *) prob
          FROM mining_data_apply_v))
WHERE rnk_clus2 <= 10
ORDER BY rnk_clus2;

   CUST_ID
----------
    100256
    100988
    100889
    101086
    101215
    100390
    100985
    101026
    100601
    100672
```

CLUSTER_SET

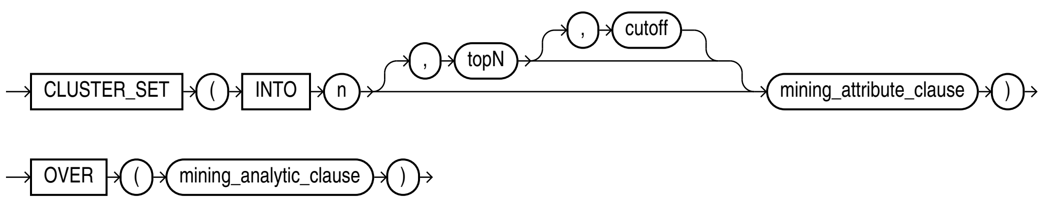
Syntax

***cluster_set*::=**

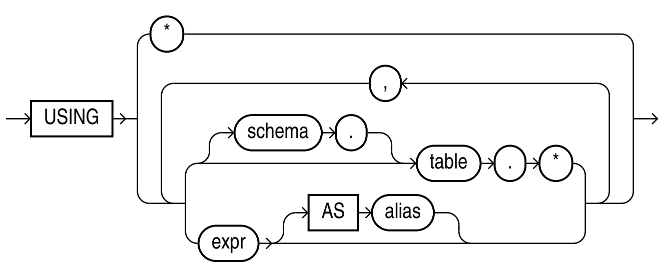


Analytic Syntax

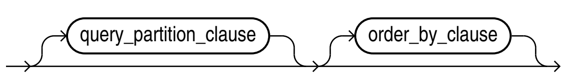
***cluster_set_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of `mining_analytic_clause`

Purpose

CLUSTER_SET returns a set of cluster ID and probability pairs for each row in the selection. The return value is a varray of objects with field names CLUSTER_ID and PROBABILITY. The cluster identifier is an Oracle NUMBER; the probability is BINARY_DOUBLE.

topN and cutoff

You can specify *topN* and *cutoff* to limit the number of clusters returned by the function. By default, both *topN* and *cutoff* are null and all clusters are returned.

- *topN* is the *N* most probable clusters. If multiple clusters share the *N*th probability, then the function chooses one of them.
- *cutoff* is a probability threshold. Only clusters with probability greater than or equal to *cutoff* are returned. To filter by *cutoff* only, specify NULL for *topN*.

To return up to the *N* most probable clusters that are greater than or equal to *cutoff*, specify both *topN* and *cutoff*.

Syntax Choice

CLUSTER_SET can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the CLUSTER_SET function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about clustering.

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

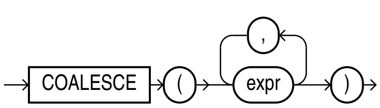
Example

This example lists the attributes that have the greatest impact (more that 20% probability) on cluster assignment for customer ID 100955. The query invokes the `CLUSTER_DETAILS` and `CLUSTER_SET` functions, which apply the clustering model `em_sh_clus_sample`.

```
SELECT S.cluster_id, probability prob,
       CLUSTER_DETAILS(em_sh_clus_sample, S.cluster_id, 5 USING T.*) det
FROM
  (SELECT v.*, CLUSTER_SET(em_sh_clus_sample, NULL, 0.2 USING *) pset
    FROM mining_data_apply_v v
   WHERE cust_id = 100955) T,
  TABLE(T.pset) S
ORDER BY 2 DESC;

CLUSTER_ID  PROB DET
---------- ----- ------------------------------------------------------------------------------
        14 .6761 <Details algorithm="Expectation Maximization" cluster="14">
                 <Attribute name="AGE" actualValue="51" weight=".676" rank="1"/>
                 <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".557" rank="2"/>
                 <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".412" rank="3"/>
                 <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".171" rank="4"/>
                 <Attribute name="BOOKKEEPING_APPLICATION" actualValue="1"
weight="-.003"rank="5"/>
                 </Details>

         3 .3227 <Details algorithm="Expectation Maximization" cluster="3">
                 <Attribute name="YRS_RESIDENCE" actualValue="3" weight=".323" rank="1"/>
                 <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".265" rank="2"/>
                 <Attribute name="EDUCATION" actualValue="HS-grad" weight=".172" rank="3"/>
                 <Attribute name="AFFINITY_CARD" actualValue="0" weight=".125" rank="4"/>
                 <Attribute name="OCCUPATION" actualValue="Crafts" weight=".055" rank="5"/>
                 </Details>
```

COALESCE

Syntax



Purpose

`COALESCE` returns the first non-null *`expr`* in the expression list. You must specify at least two expressions. If all occurrences of *`expr`* evaluate to null, then the function returns null.

Oracle Database uses **short-circuit evaluation**. The database evaluates each *expr* value and determines whether it is NULL, rather than evaluating all of the *expr* values before determining whether any of them is NULL.

If all occurrences of *expr* are numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type, then Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> - Table 2-8 for more information on implicit conversion and Numeric Precedence for information on numeric precedence
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of COALESCE when it is a character value

This function is a generalization of the NVL function.

You can also use COALESCE as a variety of the CASE expression. For example,

```
COALESCE(expr1, expr2)
```

is equivalent to:

```
CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END
```

Similarly,

```
COALESCE(expr1, expr2, ..., exprn)
```

where n >= 3, is equivalent to:

```
CASE WHEN expr1 IS NOT NULL THEN expr1
   ELSE COALESCE (expr2, ..., exprn) END
```

> **See Also:**
>
> NVL and CASE Expressions

Examples

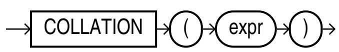
The following example uses the sample oe.product_information table to organize a clearance sale of products. It gives a 10% discount to all products with a list price. If there is no list price, then the sale price is the minimum price. If there is no minimum price, then the sale price is "5":

```
SELECT product_id, list_price, min_price,
       COALESCE(0.9*list_price, min_price, 5) "Sale"
  FROM product_information
  WHERE supplier_id = 102050
  ORDER BY product_id;
```

```
PRODUCT_ID LIST_PRICE  MIN_PRICE       Sale
---------- ---------- ---------- ----------
      1769         48                  43.2
      1770                    73         73
      2378        305        247      274.5
      2382        850        731        765
      3355                                5
```

COLLATION

Syntax



Purpose

`COLLATION` returns the name of the derived collation for *`expr`*. This function returns named collations and pseudo-collations. If the derived collation is a Unicode Collation Algorithm (UCA) collation, then the function returns the long form of its name. This function is evaluated during compilation of the SQL statement that contains it. If the derived collation is undefined due to a collation conflict while evaluating *`expr`*, then the function returns null.

`expr` must evaluate to a character string of type `CHAR`, `VARCHAR2`, `LONG`, `NCHAR`, or `NVARCHAR2`.

This function returns a `VARCHAR2` value.

> **Note:**
>
> The `COLLATION` function returns only the data-bound collation, and not the dynamic collation set by the `NLS_SORT` parameter. Thus, for a column declared as `COLLATE USING_NLS_SORT`, the function returns the character value `'USING_NLS_SORT'`, not the actual value of the session parameter `NLS_SORT`. You can use the built-in function `SYS_CONTEXT('USERENV','NLS_SORT')` to get the actual value of the session parameter `NLS_SORT`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `COLLATION`

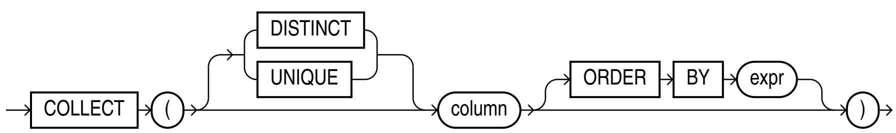
Examples

The following example returns the derived collation of columns `name` and `id` in table `id_table`:

```
CREATE TABLE id_table
  (name VARCHAR2(64) COLLATE BINARY_AI,
   id VARCHAR2(8) COLLATE BINARY_CI);

INSERT INTO id_table VALUES('Christopher', 'ABCD1234');

SELECT COLLATION(name), COLLATION(id)
  FROM id_table;

COLLATION COLLATION
--------- ---------
BINARY_AI BINARY_CI
```

COLLECT

Syntax



Purpose

`COLLECT` is an aggregate function that takes as its argument a column of any type and creates a nested table of the input type out of the rows selected. To get accurate results from this function you must use it within a `CAST` function.

If *`column`* is itself a collection, then the output of `COLLECT` is a nested table of collections. If *`column`* is of a user-defined type, then *`column`* must have a `MAP` or `ORDER` method defined on it in order for you to use the optional `DISTINCT`, `UNIQUE`, and `ORDER BY` clauses.

> **See Also:**
>
> - CAST and Aggregate Functions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `COLLECT` uses to compare character values for the `DISTINCT` and `ORDER BY` clauses

Examples

The following example creates a nested table from the varray column of phone numbers in the sample table `oe.customers`. The nested table includes only the phone numbers of customers with an income level of `L: 300,000 and above`.

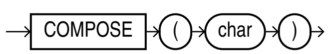
```
CREATE TYPE phone_book_t AS TABLE OF phone_list_typ;
/

SELECT CAST(COLLECT(phone_numbers) AS phone_book_t) "Income Level L Phone Book"
  FROM customers
  WHERE income_level = 'L: 300,000 and above';

Income Level L Phone Book
--------------------------------------------------------------------------------
PHONE_BOOK_T(PHONE_LIST_TYP('+1 414 123 4307'), PHONE_LIST_TYP('+1 608 123 4344'
), PHONE_LIST_TYP('+1 814 123 4696'), PHONE_LIST_TYP('+1 215 123 4721'), PHONE_L
IST_TYP('+1 814 123 4755'), PHONE_LIST_TYP('+91 11 012 4817', '+91 11 083 4817')
, PHONE_LIST_TYP('+91 172 012 4837'), PHONE_LIST_TYP('+41 31 012 3569', '+41 31
083 3569'))
```

The following example creates a nested table from the column of warehouse names in the sample table `oe.warehouses`. It uses `ORDER BY` to order the warehouse names.

```
CREATE TYPE warehouse_name_t AS TABLE OF VARCHAR2(35);
/

SELECT CAST(COLLECT(warehouse_name ORDER BY warehouse_name)
       AS warehouse_name_t) "Warehouses"
   FROM warehouses;

Warehouses
--------------------------------------------------------------------------------
WAREHOUSE_NAME_TYP('Beijing', 'Bombay', 'Mexico City', 'New Jersey', 'San Franci
sco', 'Seattle, Washington', 'Southlake, Texas', 'Sydney', 'Toronto')
```

COMPOSE

Syntax



Purpose

`COMPOSE` takes as its argument a string, or an expression that resolves to a string, in any data type, and returns a Unicode string in the same character set as the input. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. For example, an `o` code point qualified by an umlaut code point will be returned as the o-umlaut code point.

`COMPOSE` returns the string in NFC normal form. For a more exclusive setting, you can first call `DECOMPOSE` with the `CANONICAL` setting and then `COMPOSE`. This combination returns the string in NFKC normal form.

`CLOB` and `NCLOB` values are supported through implicit conversion. If *`char`* is a character LOB value, then it is converted to a `VARCHAR` value before the `COMPOSE` operation. The operation will fail if the size of the LOB value exceeds the supported length of the `VARCHAR` in the particular development environment.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode character sets and character semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `COMPOSE`
> - DECOMPOSE

Examples

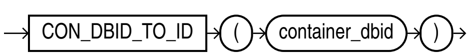
The following example returns the o-umlaut code point:

```
SELECT COMPOSE( 'o' || UNISTR('\0308') )
  FROM DUAL;

CO
--
ö
```

> **See Also:**
>
> UNISTR

CON_DBID_TO_ID

Syntax



Purpose

`CON_DBID_TO_ID` takes as its argument a container DBID and returns the container ID. For *`container_dbid`*, specify a `NUMBER` value or any value that can be implicitly converted to `NUMBER`. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and DBID for all containers in a CDB. The sample output shown is for the purpose of this example.

```
SELECT CON_ID, DBID
  FROM V$CONTAINERS;

    CON_ID       DBID
---------- ----------
```

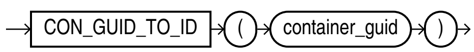
```
         1 1930093401
         2 4054529501
         4 2256797992
```

The following statement returns the ID for the container with DBID 2256797992:

```
SELECT CON_DBID_TO_ID(2256797992) "Container ID"
  FROM DUAL;

Container ID
------------
           4
```

CON_GUID_TO_ID

Syntax



Purpose

`CON_GUID_TO_ID` takes as its argument a container GUID (globally unique identifier) and returns the container ID. For *`container_guid`*, specify a raw value. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and GUID for all containers in a CDB. The GUID is stored as a 16-byte `RAW` value in the `V$CONTAINERS` view. The query returns the 32-character hexadecimal representation of the GUID. The sample output shown is for the purpose of this example.

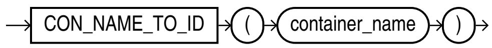
```
SELECT CON_ID, GUID
  FROM V$CONTAINERS;

    CON_ID GUID
---------- --------------------------------
         1 DB0A9F33DF99567FE04305B4F00A667D
         2 D990C280C309591EE04305B4F00A593E
         4 D990F4BD938865C1E04305B4F00ACA18
```

The following statement returns the ID for the container whose GUID is represented by the hexadecimal value `D990F4BD938865C1E04305B4F00ACA18`. The `HEXTORAW` function converts the GUID's hexadecimal representation to a raw value.

```
SELECT CON_GUID_TO_ID(HEXTORAW('D990F4BD938865C1E04305B4F00ACA18')) "Container
ID"
  FROM DUAL;

Container ID
------------
           4
```

CON_NAME_TO_ID

Syntax



Purpose

`CON_NAME_TO_ID` takes as its argument a container name and returns the container ID. For *container_name*, specify a string, or an expression that resolves to a string, in any data type. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and name for all containers in a CDB. The sample output shown is for the purpose of this example.

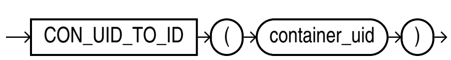
```
SELECT CON_ID, NAME
  FROM V$CONTAINERS;

    CON_ID NAME
---------- ----------
         1 CDB$ROOT
         2 PDB$SEED
         4 SALESPDB
```

The following statement returns the ID for the container named `SALESPDB`:

```
SELECT CON_NAME_TO_ID('SALESPDB') "Container ID"
  FROM DUAL;

Container ID
------------
           4
```

CON_UID_TO_ID

Syntax



Purpose

`CON_UID_TO_ID` takes as its argument a container UID (unique identifier) and returns the container ID. For *container_uid*, specify a `NUMBER` value or any value that can be implicitly converted to `NUMBER`. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and UID for all containers in a CDB. The sample output shown is for the purpose of this example.

```
SELECT CON_ID, CON_UID
  FROM V$CONTAINERS;

    CON_ID    CON_UID
---------- ----------
         1          1
         2 4054529501
         4 2256797992
```

The following query returns the ID for the container with UID 2256797992:

```
SELECT CON_UID_TO_ID(2256797992) "Container ID"
  FROM DUAL;

Container ID
------------
           4
```

CONCAT

Syntax

Purpose

`CONCAT` returns *`char1`* concatenated with *`char2`*. Both *`char1`* and *`char2`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is in the same character set as *`char1`*. Its data type depends on the data types of the arguments.

In concatenations of two different data types, Oracle Database returns the data type that results in a lossless conversion. Therefore, if one of the arguments is a LOB, then the returned value is a LOB. If one of the arguments is a national data type, then the returned value is a national data type. For example:

- `CONCAT`(`CLOB`, `NCLOB`) returns `NCLOB`
- `CONCAT`(`NCLOB`, `NCHAR`) returns `NCLOB`
- `CONCAT`(`NCLOB`, `CHAR`) returns `NCLOB`
- `CONCAT`(`NCHAR`, `CLOB`) returns `NCLOB`

This function is equivalent to the concatenation operator (||).

See Also:

- Concatenation Operator for information on the CONCAT operator
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of CONCAT

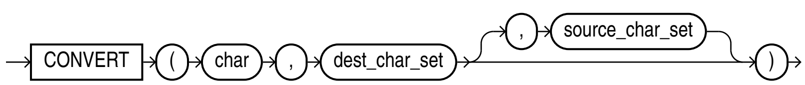
Examples

This example uses nesting to concatenate three character strings:

```
SELECT CONCAT(CONCAT(last_name, '''s job category is '), job_id) "Job"
  FROM employees
  WHERE employee_id = 152;

Job
------------------------------------------------------
Hall's job category is SA_REP
```

CONVERT

Syntax



Purpose

CONVERT converts a character string from one character set to another.

- The *char* argument is the value to be converted. It can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- The *dest_char_set* argument is the name of the character set to which *char* is converted.
- The *source_char_set* argument is the name of the character set in which *char* is stored in the database. The default value is the database character set.

The return value for CHAR and VARCHAR2 is VARCHAR2. For NCHAR and NVARCHAR2, it is NVARCHAR2. For CLOB, it is CLOB, and for NCLOB, it is NCLOB.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set.

For complete correspondence in character conversion, it is essential that the destination character set contains a representation of all the characters defined in the source character set. Where a character does not exist in the destination character set, a replacement character appears. Replacement characters can be defined as part of a character set definition.

Note:

Oracle discourages the use of the `CONVERT` function in the current Oracle Database release. The return value of `CONVERT` has a character data type, so it should be either in the database character set or in the national character set, depending on the data type. Any *`dest_char_set`* that is not one of these two character sets is unsupported. The *`char`* argument and the *`source_char_set`* have the same requirements. Therefore, the only practical use of the function is to correct data that has been stored in a wrong character set.

Values that are in neither the database nor the national character set should be processed and stored as `RAW` or `BLOB`. Procedures in the PL/SQL packages `UTL_RAW` and `UTL_I18N`—for example, `UTL_RAW.CONVERT`—allow limited processing of such values. Procedures accepting a `RAW` argument in the packages `UTL_FILE`, `UTL_TCP`, `UTL_HTTP`, and `UTL_SMTP` can be used to output the processed data.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `CONVERT`

Examples

The following example illustrates character set conversion by converting a Latin-1 string to ASCII. The result is the same as importing the same string from a WE8ISO8859P1 database to a US7ASCII database.

```
SELECT CONVERT('Ä Ê Í Õ Ø A B C D E ', 'US7ASCII', 'WE8ISO8859P1')
   FROM DUAL;

CONVERT('ÄÊÍÕØABCDE'
---------------------
A E I ? ? A B C D E ?
```

You can query the `V$NLS_VALID_VALUES` view to get a listing of valid character sets, as follows:

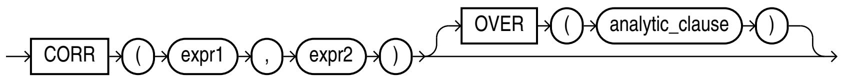
```
SELECT * FROM V$NLS_VALID_VALUES WHERE parameter = 'CHARACTERSET';
```

See Also:

Oracle Database Globalization Support Guide for the list of character sets that Oracle Database supports and *Oracle Database Reference* for information on the `V$NLS_VALID_VALUES` view

CORR

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

CORR returns the coefficient of correlation of a set of number pairs. You can use it as an aggregate or analytic function.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

Oracle Database applies the function to the set of (*expr1*, *expr2*) after eliminating the pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
COVAR_POP(expr1, expr2) / (STDDEV_POP(expr1) * STDDEV_POP(expr2))
```

The function returns a value of type NUMBER. If the function is applied to an empty set, then it returns null.

> **Note:**
>
> The CORR function calculates the Pearson's correlation coefficient, which requires numeric expressions as arguments. Oracle also provides the CORR_S (Spearman's rho coefficient) and CORR_K (Kendall's tau-b coefficient) functions to support nonparametric or rank correlation.

> **See Also:**
>
> Aggregate Functions , About SQL Expressions for information on valid forms of *expr*, and CORR_* for information on the CORR_S and CORR_K functions

Aggregate Example

The following example calculates the coefficient of correlation between the list prices and minimum prices of products by weight class in the sample table oe.product_information:

```
SELECT weight_class, CORR(list_price, min_price) "Correlation"
  FROM product_information
  GROUP BY weight_class
  ORDER BY weight_class, "Correlation";

WEIGHT_CLASS Correlation
------------ -----------
           1  .999149795
           2  .999022941
           3  .998484472
           4  .999359909
           5  .999536087
```

Analytic Example

The following example shows the correlation between duration at the company and salary by the employee's position. The result set shows the same correlation for each employee in a given job:

```
SELECT employee_id, job_id,
       TO_CHAR((SYSDATE - hire_date) YEAR TO MONTH ) "Yrs-Mns",     salary,
       CORR(SYSDATE-hire_date, salary)
       OVER(PARTITION BY job_id) AS "Correlation"
  FROM employees
  WHERE department_id in (50, 80)
  ORDER BY job_id, employee_id;

EMPLOYEE_ID JOB_ID     Yrs-Mns     SALARY Correlation
----------- ---------- ------- ---------- -----------
        145 SA_MAN     +04-09       14000  .912385598
        146 SA_MAN     +04-06       13500  .912385598
        147 SA_MAN     +04-04       12000  .912385598
        148 SA_MAN     +01-08       11000  .912385598
        149 SA_MAN     +01-05       10500  .912385598
        150 SA_REP     +04-05       10000   .80436755
        151 SA_REP     +04-03        9500   .80436755
        152 SA_REP     +03-10        9000   .80436755
        153 SA_REP     +03-03        8000   .80436755
        154 SA_REP     +02-07        7500   .80436755
        155 SA_REP     +01-07        7000   .80436755
. . .
```

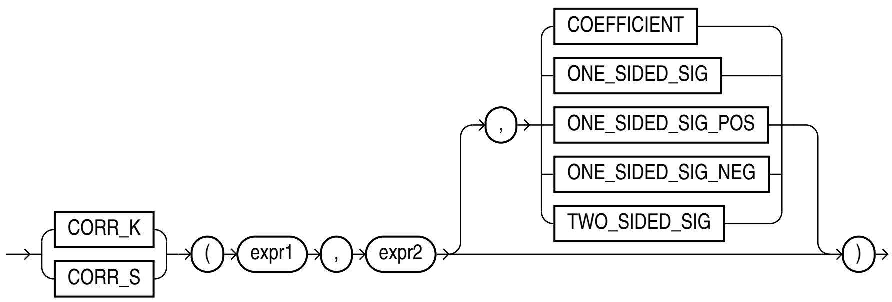
CORR_*

The CORR_* functions are:

- CORR_S
- CORR_K

Syntax

***correlation*::=**



Purpose

The CORR function (see CORR) calculates the Pearson's correlation coefficient and requires numeric expressions as input. The CORR_* functions support nonparametric or rank correlation. They let you find correlations between expressions that are ordinal scaled (where ranking of the values is possible). Correlation coefficients take on a value ranging from -1 to 1, where 1 indicates a perfect relationship, -1 a perfect inverse relationship (when one variable increases as the other decreases), and a value close to 0 means no relationship.

These functions takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, makes the calculation, and returns NUMBER.

> **See Also:**
>
> - Table 2-8 for more information on implicit conversion and Numeric Precedence for information on numeric precedence
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation CORR_K and CORR_S use to compare characters from *expr1* with characters from *expr2*

expr1 and *expr2* are the two variables being analyzed. The third argument is a return value of type VARCHAR2. If you omit the third argument, then the default is COEFFICIENT. The meaning of the return values is shown in the table that follows:

Table 7-2 CORR_* Return Values

Return Value	Meaning
COEFFICIENT	Coefficient of correlation

Table 7-2 (Cont.) CORR_* Return Values

Return Value	Meaning
ONE_SIDED_SIG	Positive one-tailed significance of the correlation
ONE_SIDED_SIG_POS	Same as ONE_SIDED_SIG
ONE_SIDED_SIG_NEG	Negative one-tailed significance of the correlation
TWO_SIDED_SIG	Two-tailed significance of the correlation

CORR_S

CORR_S calculates the Spearman's rho correlation coefficient. The input expressions should be a set of (x_i, y_i) pairs of observations. The function first replaces each value with a rank. Each value of x_i is replaced with its rank among all the other x_is in the sample, and each value of y_i is replaced with its rank among all the other y_is. Thus, each x_i and y_i take on a value from 1 to *n*, where *n* is the total number of pairs of values. Ties are assigned the average of the ranks they would have had if their values had been slightly different. Then the function calculates the linear correlation coefficient of the ranks.

CORR_S Example

Using Spearman's rho correlation coefficient, the following example derives a coefficient of correlation for each of two different comparisons -- salary and commission_pct, and salary and employee_id:

```
SELECT COUNT(*) count,
       CORR_S(salary, commission_pct) commission,
       CORR_S(salary, employee_id) empid
  FROM employees;

     COUNT COMMISSION      EMPID
---------- ---------- ----------
       107 .735837022 -.04473016
```

CORR_K

CORR_K calculates the Kendall's tau-b correlation coefficient. As for CORR_S, the input expressions are a set of (x_i, y_i) pairs of observations. To calculate the coefficient, the function counts the number of concordant and discordant pairs. A pair of observations is concordant if the observation with the larger x also has a larger value of y. A pair of observations is discordant if the observation with the larger x has a smaller y.

The significance of tau-b is the probability that the correlation indicated by tau-b was due to chance—a value of 0 to 1. A small value indicates a significant correlation for positive values of tau-b (or anticorrelation for negative values of tau-b).

CORR_K Example

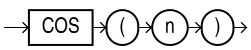
Using Kendall's tau-b correlation coefficient, the following example determines whether a correlation exists between an employee's salary and commission percent:

```
SELECT CORR_K(salary, commission_pct, 'COEFFICIENT') coefficient,
       CORR_K(salary, commission_pct, 'TWO_SIDED_SIG') two_sided_p_value
```

```
  FROM employees;

COEFFICIENT TWO_SIDED_P_VALUE
----------- -----------------
 .603079768        3.4702E-07
```

COS

Syntax



Purpose

`COS` returns the cosine of *n* (an angle expressed in radians).

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

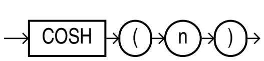
Examples

The following example returns the cosine of 180 degrees:

```
SELECT COS(180 * 3.14159265359/180) "Cosine of 180 degrees"
  FROM DUAL;

Cosine of 180 degrees
---------------------
                   -1
```

COSH

Syntax



Purpose

`COSH` returns the hyperbolic cosine of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns BINARY_DOUBLE. Otherwise the function returns the same numeric data type as the argument.

See Also:

Table 2-8 for more information on implicit conversion

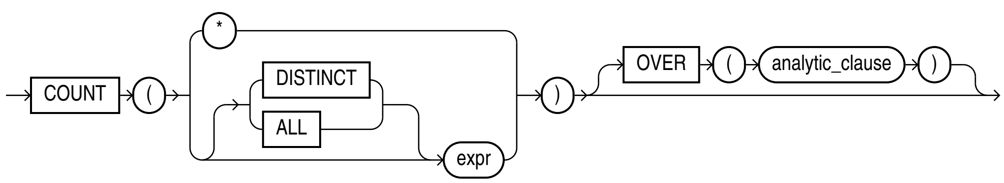
Examples

The following example returns the hyperbolic cosine of zero:

```
SELECT COSH(0) "Hyperbolic cosine of 0"
  FROM DUAL;

Hyperbolic cosine of 0
----------------------
                     1
```

COUNT

Syntax



See Also:

Analytic Functions for information on syntax, semantics, and restrictions

Purpose

COUNT returns the number of rows returned by the query. You can use it as an aggregate or analytic function.

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

If you specify *expr*, then COUNT returns the number of rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.

If you specify the asterisk (*), then this function returns all rows, including duplicates and nulls. COUNT never returns null.

> **Note:**
>
> Before performing a `COUNT (DISTINCT expr)` operation on a large amount of data, consider using one of the following methods to obtain approximate results more quickly than exact results:
>
> - Set the `APPROX_FOR_COUNT_DISTINCT` initialization parameter to true before using the `COUNT (DISTINCT expr)` function. Refer to *Oracle Database Reference* for more information on this parameter.
> - Use the `APPROX_COUNT_DISTINCT` function instead of the `COUNT (DISTINCT expr)` function. Refer to APPROX_COUNT_DISTINCT.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *expr* and Aggregate Functions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `COUNT` uses to compare character values for the `DISTINCT` clause

Aggregate Examples

The following examples use `COUNT` as an aggregate function:

```
SELECT COUNT(*) "Total"
  FROM employees;

     Total
----------
       107

SELECT COUNT(*) "Allstars"
  FROM employees
  WHERE commission_pct > 0;

 Allstars
---------
       35

SELECT COUNT(commission_pct) "Count"
  FROM employees;

     Count
----------
        35

SELECT COUNT(DISTINCT manager_id) "Managers"
  FROM employees;

  Managers
----------
        18
```

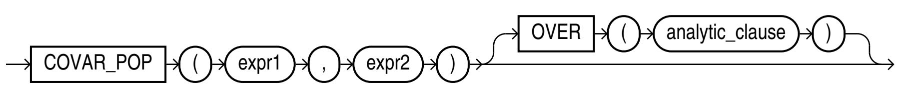
Analytic Example

The following example calculates, for each employee in the `employees` table, the moving count of employees earning salaries in the range 50 less than through 150 greater than the employee's salary.

```
SELECT last_name, salary,
       COUNT(*) OVER (ORDER BY salary RANGE BETWEEN 50 PRECEDING AND
                      150 FOLLOWING) AS mov_count
  FROM employees
  ORDER BY salary, last_name;

LAST_NAME                     SALARY  MOV_COUNT
------------------------- ---------- ----------
Olson                           2100          3
Markle                          2200          2
Philtanker                      2200          2
Gee                             2400          8
Landry                          2400          8
Colmenares                      2500         10
Marlow                          2500         10
Patel                           2500         10
. . .
```

COVAR_POP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`COVAR_POP` returns the population covariance of a set of number pairs. You can use it as an aggregate or analytic function.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

Oracle Database applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

where *n* is the number of (*expr1*, *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type `NUMBER`. If the function is applied to an empty set, then it returns null.

> **See Also:**
>
> About SQL Expressions for information on valid forms of *expr* and Aggregate Functions

Aggregate Example

The following example calculates the population covariance and sample covariance for time employed (`SYSDATE` - `hire_date`) and salary using the sample table `hr.employees`:

```
SELECT job_id,
       COVAR_POP(SYSDATE-hire_date, salary) AS covar_pop,
       COVAR_SAMP(SYSDATE-hire_date, salary) AS covar_samp
  FROM employees
  WHERE department_id in (50, 80)
  GROUP BY job_id
  ORDER BY job_id, covar_pop, covar_samp;

JOB_ID       COVAR_POP  COVAR_SAMP
---------- ----------- -----------
SA_MAN          660700      825875
SA_REP      579988.466   600702.34
SH_CLERK      212432.5  223613.158
ST_CLERK     176577.25  185870.789
ST_MAN          436092      545115
```

Analytic Example

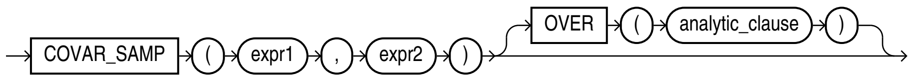
The following example calculates cumulative sample covariance of the list price and minimum price of the products in the sample schema `oe`:

```
SELECT product_id, supplier_id,
       COVAR_POP(list_price, min_price)
         OVER (ORDER BY product_id, supplier_id)
         AS CUM_COVP,
       COVAR_SAMP(list_price, min_price)
         OVER (ORDER BY product_id, supplier_id)
         AS CUM_COVS
  FROM product_information p
  WHERE category_id = 29
  ORDER BY product_id, supplier_id;

PRODUCT_ID SUPPLIER_ID   CUM_COVP   CUM_COVS
---------- ----------- ---------- ----------
      1774      103088          0
      1775      103087    1473.25     2946.5
      1794      103096 1702.77778 2554.16667
```

```
      1825       103093    1926.25 2568.33333
      2004       103086     1591.4    1989.25
      2005       103086     1512.5       1815
      2416       103088 1475.97959 1721.97619
. . .
```

COVAR_SAMP

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`COVAR_SAMP` returns the sample covariance of a set of number pairs. You can use it as an aggregate or analytic function.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

Oracle Database applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr1) * SUM(expr2) / n) / (n-1)
```

where *n* is the number of (*expr1*, *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type `NUMBER`. If the function is applied to an empty set, then it returns null.

> **See Also:**
>
> About SQL Expressions for information on valid forms of *expr* and Aggregate Functions

Aggregate Example

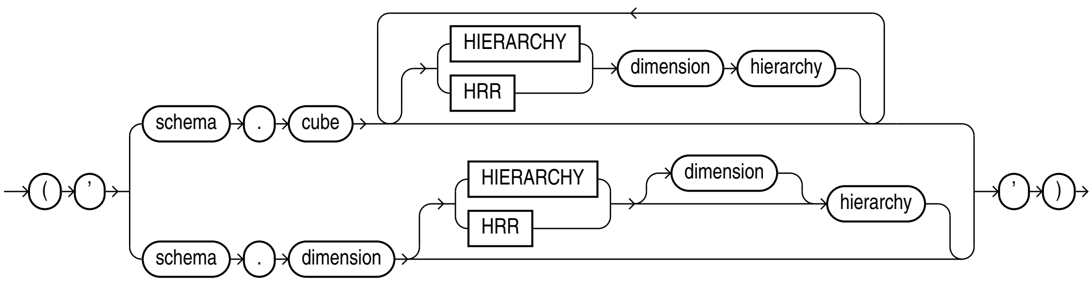
Refer to the aggregate example for COVAR_POP .

Analytic Example

Refer to the analytic example for COVAR_POP .

CUBE_TABLE

Syntax



Purpose

`CUBE_TABLE` extracts data from a cube or dimension and returns it in the two-dimensional format of a relational table, which can be used by SQL-based applications.

The function takes a single `VARCHAR2` argument. The optional hierarchy clause enables you to specify a dimension hierarchy. A cube can have multiple hierarchy clauses, one for each dimension.

You can generate these different types of tables:

- A cube table contains a key column for each dimension and a column for each measure and calculated measure in the cube. To create a cube table, you can specify the cube with or without a cube hierarchy clause. For a dimension with multiple hierarchies, this clause limits the return values to the dimension members and levels in the specified hierarchy. Without a hierarchy clause, all dimension members and all levels are included.
- A dimension table contains a key column, and a column for each level and each attribute. It also contains a `MEMBER_TYPE` column, which identifies each member with one of the following codes:
  - `L` - Loaded from a table, view, or synonym
  - `A` - Loaded member and the single root of all hierarchies in the dimension, that is, the "all" aggregate member
  - `C` - Calculated member

  All dimension members and all levels are included in the table. To create a dimension table, specify the dimension **without** a dimension hierarchy clause.

- A hierarchy table contains all the columns of a dimension table plus a column for the parent member and a column for each source level. It also contains a `MEMBER_TYPE` column, as described for dimension tables. Any dimension members and levels that are not part of the named hierarchy are excluded from the table. To create a hierarchy table, specify the dimension **with** a dimension hierarchy clause.

`CUBE_TABLE` is a table function and is always used in the context of a `SELECT` statement with this syntax:

```
SELECT ... FROM TABLE(CUBE_TABLE('arg'));
```

See Also:

- *Oracle OLAP User's Guide* for information about dimensional objects and about the tables generated by `CUBE_TABLE`.
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to each character data type column in the table generated by `CUBE_TABLE`

Examples

The following examples require Oracle Database with the OLAP option and the `GLOBAL` sample schema. Refer to *Oracle OLAP User's Guide* for information on downloading and installing the `GLOBAL` sample schema.

The following `SELECT` statement generates a dimension table of `CHANNEL` in the `GLOBAL` schema.

```
SELECT dim_key, level_name, long_description, channel_total_id tot_id,
       channel_channel_id chan_id, channel_long_description chan_desc,
       total_long_description tot_desc
  FROM TABLE(CUBE_TABLE('global.channel'));

DIM_KEY      LEVEL_NAME LONG_DESCRIPTION TOT_ID CHAN_ID CHAN_DESC    TOT_DESC
-----------  ---------- ---------------- ------ ------- ------------
-------------
CHANNEL_CAT  CHANNEL    Catalog          TOTAL  CAT     Catalog      Total
Channel
CHANNEL_DIR  CHANNEL    Direct Sales     TOTAL  DIR     Direct Sales Total
Channel
CHANNEL_INT  CHANNEL    Internet         TOTAL  INT     Internet     Total
Channel
TOTAL_TOTAL  TOTAL      Total Channel    TOTAL                       Total
Channel
```

The next statement generates a cube table of `UNITS_CUBE`. It restricts the table to the `MARKET` and `CALENDAR` hierarchies.

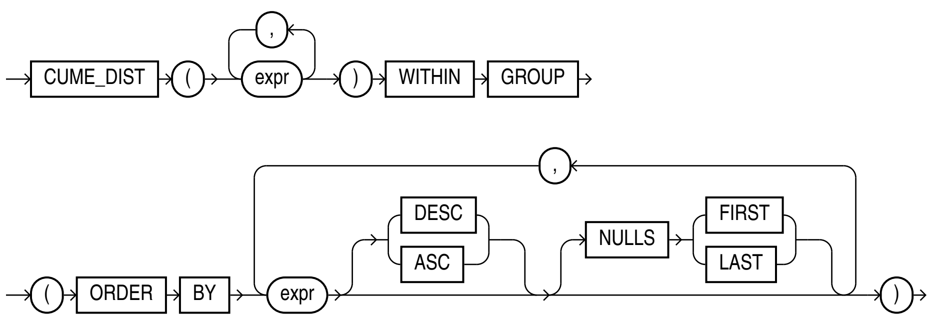
```
SELECT sales, units, cost, time, customer, product, channel
  FROM TABLE(CUBE_TABLE('global.units_cube HIERARCHY customer market HIERARCHY time calendar'))
  WHERE rownum < 20;

     SALES      UNITS       COST TIME                      CUSTOMER       PRODUCT     CHANNEL
---------- ---------- ---------- ------------------------- -------------- -----------
-----------
24538587.9      61109 22840853.7 CALENDAR_QUARTER_CY1998.Q1 TOTAL_TOTAL    TOTAL_TOTAL
```

```
TOTAL_TOTAL
24993273.3      61320   23147171 CALENDAR_QUARTER_CY1998.Q2 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
25080541.4      65265 23242535.4 CALENDAR_QUARTER_CY1998.Q3 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
  26258474      66122 24391020.6 CALENDAR_QUARTER_CY1998.Q4 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
  32785170      77589 30607218.1 CALENDAR_QUARTER_CY1999.Q1 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
. . .
```

CUME_DIST

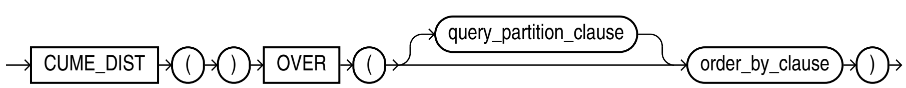
Aggregate Syntax

***cume_dist_aggregate*::=**



Analytic Syntax

***cume_dist_analytic*::=**



See Also:

Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`CUME_DIST` calculates the cumulative distribution of a value in a group of values. The range of values returned by `CUME_DIST` is >0 to <=1. Tie values always evaluate to the same cumulative distribution value.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, makes the calculation, and returns `NUMBER`.

See Also:

Table 2-8 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

- As an aggregate function, `CUME_DIST` calculates, for a hypothetical row *r* identified by the arguments of the function and a corresponding sort specification, the relative position of row *r* among the rows in the aggregation group. Oracle makes this calculation as if the hypothetical row *r* were inserted into the group of rows to be aggregated over. The arguments of the function identify a single hypothetical row within each aggregate group. Therefore, they must all evaluate to constant expressions within each aggregate group. The constant argument expressions and the expressions in the `ORDER BY` clause of the aggregate match by position. Therefore, the number of arguments must be the same and their types must be compatible.
- As an analytic function, `CUME_DIST` computes the relative position of a specified value in a group of values. For a row *r*, assuming ascending ordering, the `CUME_DIST` of *r* is the number of rows with values lower than or equal to the value of *r*, divided by the number of rows being evaluated (the entire query result set or a partition).

Aggregate Example

The following example calculates the cumulative distribution of a hypothetical employee with a salary of $15,500 and commission rate of 5% among the employees in the sample table `oe.employees`:

```
SELECT CUME_DIST(15500, .05) WITHIN GROUP
  (ORDER BY salary, commission_pct) "Cume-Dist of 15500"
  FROM employees;

Cume-Dist of 15500
------------------
        .972222222
```

Analytic Example

The following example calculates the salary percentile for each employee in the purchasing division. For example, 40% of clerks have salaries less than or equal to Himuro.

```
SELECT job_id, last_name, salary, CUME_DIST()
  OVER (PARTITION BY job_id ORDER BY salary) AS cume_dist
  FROM employees
  WHERE job_id LIKE 'PU%'
  ORDER BY job_id, last_name, salary, cume_dist;

JOB_ID     LAST_NAME                     SALARY  CUME_DIST
---------- ------------------------- ---------- ----------
PU_CLERK   Baida                           2900         .8
PU_CLERK   Colmenares                      2500         .2
PU_CLERK   Himuro                          2600         .4
PU_CLERK   Khoo                            3100          1
PU_CLERK   Tobias                          2800         .6
PU_MAN     Raphaely                       11000          1
```

CURRENT_DATE

Syntax

Purpose

`CURRENT_DATE` returns the current date in the session time zone, in a value in the Gregorian calendar of data type `DATE`.

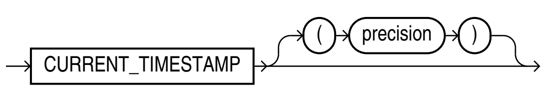
Examples

The following example illustrates that `CURRENT_DATE` is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-05:00          29-MAY-2000 13:14:03

ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-08:00          29-MAY-2000 10:14:33
```

CURRENT_TIMESTAMP

Syntax



Purpose

`CURRENT_TIMESTAMP` returns the current date and time in the session time zone, in a value of data type `TIMESTAMP WITH TIME ZONE`. The time zone offset reflects the current local time of the SQL session. If you omit precision, then the default is 6. The difference between this function and `LOCALTIMESTAMP` is that `CURRENT_TIMESTAMP` returns a `TIMESTAMP WITH TIME ZONE` value while `LOCALTIMESTAMP` returns a `TIMESTAMP` value.

In the optional argument, *`precision`* specifies the fractional second precision of the time value returned.

See Also:

LOCALTIMESTAMP

Examples

The following example illustrates that `CURRENT_TIMESTAMP` is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE CURRENT_TIMESTAMP
--------------- ---------------------------------------------------
-05:00          04-APR-00 01.17.56.917550 PM -05:00

ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE CURRENT_TIMESTAMP
--------------- ----------------------------------------------------
-08:00          04-APR-00 10.18.21.366065 AM -08:00
```

When you use the `CURRENT_TIMESTAMP` with a format mask, take care that the format mask matches the value returned by the function. For example, consider the following table:

```
CREATE TABLE current_test (col1 TIMESTAMP WITH TIME ZONE);
```

The following statement fails because the mask does not include the `TIME ZONE` portion of the type returned by the function:

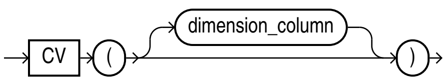
```
INSERT INTO current_test VALUES
  (TO_TIMESTAMP_TZ(CURRENT_TIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM'));
```

The following statement uses the correct format mask to match the return type of `CURRENT_TIMESTAMP`:

```
INSERT INTO current_test VALUES
  (TO_TIMESTAMP_TZ(CURRENT_TIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM TZH:TZM'));
```

CV

Syntax



Purpose

The `CV` function can be used only in the *`model_clause`* of a `SELECT` statement and then only on the right-hand side of a model rule. It returns the current value of a dimension column or a partitioning column carried from the left-hand side to the right-hand side of a rule. This function is used in the *model_clause* to provide relative indexing with respect to the dimension column. The return type is that of the data type of the dimension column. If you omit the argument, then it defaults to the dimension column associated with the relative position of the function within the cell reference.

The CV function can be used outside a cell reference. In this case, *dimension_column* is required.

> **See Also:**
>
> - *model_clause* and Model Expressions for the syntax and semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of CV when it is a character value

Examples

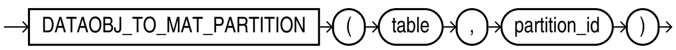
The following example assigns the sum of the sales of the product represented by the current value of the dimension column (Mouse Pad or Standard Mouse) for years 1999 and 2000 to the sales of that product for year 2001:

```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[FOR prod IN ('Mouse Pad', 'Standard Mouse'), 2001] =
        s[CV( ), 1999] + s[CV( ), 2000]
    )
  ORDER BY country, prod, year;

COUNTRY       PROD                                   YEAR          S
----------    -----------------------------------    --------   ---------
France        Mouse Pad                                  1998     2509.42
France        Mouse Pad                                  1999     3678.69
France        Mouse Pad                                  2000     3000.72
France        Mouse Pad                                  2001     6679.41
France        Standard Mouse                             1998     2390.83
France        Standard Mouse                             1999     2280.45
France        Standard Mouse                             2000     1274.31
France        Standard Mouse                             2001     3554.76
Germany       Mouse Pad                                  1998     5827.87
Germany       Mouse Pad                                  1999     8346.44
Germany       Mouse Pad                                  2000     7375.46
Germany       Mouse Pad                                  2001     15721.9
Germany       Standard Mouse                             1998     7116.11
Germany       Standard Mouse                             1999     6263.14
Germany       Standard Mouse                             2000     2637.31
Germany       Standard Mouse                             2001     8900.45

16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to The MODEL clause: Examples to create this view.

DATAOBJ_TO_MAT_PARTITION

Syntax



Purpose

`DATAOBJ_TO_MAT_PARTITION` is useful only to Data Cartridge developers who are performing data maintenance or query operations on system-partitioned tables that are used to store domain index data. The DML or query operations are triggered by corresponding operations on the base table of the domain index.

This function takes as arguments the name of the base table and the partition ID of the base table partition, both of which are passed to the function by the appropriate ODCIIndex method. The function returns the materialized partition number of the corresponding system-partitioned table, which can be used to perform the operation (DML or query) on that partition of the system-partitioned table.

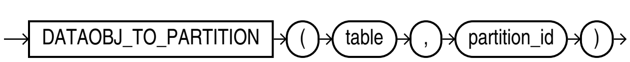
If the base table is interval partitioned, then Oracle recommends that you use this function instead of the `DATAOBJ_TO_PARTITION` function. The `DATAOBJ_TO_PARTITION` function determines the absolute partition number, given the physical partition identifier. However, if the base table is interval partitioned, then there might be holes in the partition numbers corresponding to unmaterialized partitions. Because the system partitioned table only has materialized partitions, `DATAOBJ_TO_PARTITION` numbers can cause a mis-match between the partitions of the base table and the partitions of the underlying system partitioned index storage tables. The `DATAOBJ_TO_MAT_PARTITION` function returns the materialized partition number (as opposed to the absolute partition number) and helps keep the two tables in sync. Indextypes planning to support local domain indexes on interval partitioned tables should migrate to the use of this function.

See Also:

- DATAOBJ_TO_PARTITION
- *Oracle Database Data Cartridge Developer's Guide* for information on the use of the `DATAOBJ_TO_MAT_PARTITION` function, including examples

DATAOBJ_TO_PARTITION

Syntax



Purpose

`DATAOBJ_TO_PARTITION` is useful only to Data Cartridge developers who are performing data maintenance or query operations on system-partitioned tables that are used to store domain index data. The DML or query operations are triggered by corresponding operations on the base table of the domain index.

This function takes as arguments the name of the base table and the partition ID of the base table partition, both of which are passed to the function by the appropriate ODCIIndex method. The function returns the absolute partition number of the corresponding system-partitioned table, which can be used to perform the operation (DML or query) on that partition of the system-partitioned table.

> **Note:**
>
> If the base table is interval partitioned, then Oracle recommends that you instead use the `DATAOBJ_TO_MAT_PARTITION` function. Refer to DATAOBJ_TO_MAT_PARTITION for more information.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information on the use of the `DATAOBJ_TO_PARTITION` function, including examples

DBTIMEZONE

Syntax

Purpose

`DBTIMEZONE` returns the value of the database time zone. The return type is a time zone offset (a character type in the format `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `DBTIMEZONE`

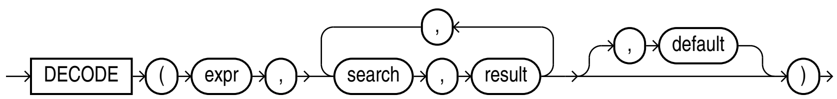
Examples

The following example assumes that the database time zone is set to UTC time zone:

```
SELECT DBTIMEZONE
  FROM DUAL;

DBTIME
------
+00:00
```

DECODE

Syntax



Purpose

`DECODE` compares *`expr`* to each *`search`* value one by one. If *`expr`* is equal to a *`search`*, then Oracle Database returns the corresponding *`result`*. If no match is found, then Oracle returns *`default`*. If *`default`* is omitted, then Oracle returns null.

The arguments can be any of the numeric types (`NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`) or character types.

- If *`expr`* and *`search`* are character data, then Oracle compares them using nonpadded comparison semantics. *`expr`*, *`search`*, and *`result`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The string returned is of `VARCHAR2` data type and is in the same character set as the first *`result`* parameter.
- If the first *`search-result`* pair are numeric, then Oracle compares all *`search-result`* expressions and the first *`expr`* to determine the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

The *`search`*, *`result`*, and *`default`* values can be derived from expressions. Oracle Database uses **short-circuit evaluation**. The database evaluates each *`search`* value only before comparing it to *`expr`*, rather than evaluating all *`search`* values before comparing any of them with *`expr`*. Consequently, Oracle never evaluates a *`search`* if a previous *`search`* is equal to *`expr`*.

Oracle automatically converts *`expr`* and each *`search`* value to the data type of the first *`search`* value before comparing. Oracle automatically converts the return value to the same data type as the first *`result`*. If the first *`result`* has the data type `CHAR` or if the first *`result`* is null, then Oracle converts the return value to the data type `VARCHAR2`.

In a `DECODE` function, Oracle considers two nulls to be equivalent. If *`expr`* is null, then Oracle returns the *`result`* of the first *`search`* that is also null.

The maximum number of components in the `DECODE` function, including *`expr`*, *`searches`*, *`results`*, and *`default`*, is 255.

See Also:

- Data Type Comparison Rules for information on comparison semantics
- Data Conversion for information on data type conversion in general
- Floating-Point Numbers for information on floating-point comparison semantics
- Implicit and Explicit Data Conversion for information on the drawbacks of implicit conversion
- COALESCE and CASE Expressions , which provide functionality similar to that of `DECODE`
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `DECODE` uses to compare characters from *`expr`* with characters from *`search`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

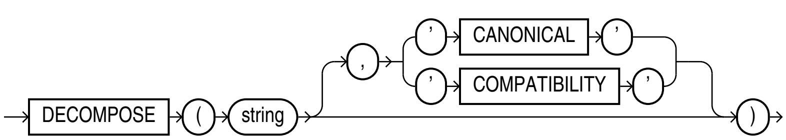
Examples

This example decodes the value `warehouse_id`. If `warehouse_id` is 1, then the function returns '`Southlake`'; if `warehouse_id` is 2, then it returns '`San Francisco`'; and so forth. If `warehouse_id` is not 1, 2, 3, or 4, then the function returns '`Non domestic`'.

```
SELECT product_id,
       DECODE (warehouse_id, 1, 'Southlake',
                             2, 'San Francisco',
                             3, 'New Jersey',
                             4, 'Seattle',
                                'Non domestic') "Location"
  FROM inventories
  WHERE product_id < 1775
  ORDER BY product_id, "Location";
```

DECOMPOSE

Syntax



Purpose

`DECOMPOSE` is valid only for Unicode characters. `DECOMPOSE` takes as its argument a string in any data type and returns a Unicode string after decomposition in the same character set as the input. For example, an o-umlaut code point will be returned as the "o" code point followed by an umlaut code point.

- *`string`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.

- `CANONICAL` causes canonical decomposition, which allows recomposition (for example, with the `COMPOSE` function) to the original string. This is the default and returns the string in NFD normal form.
- `COMPATIBILITY` causes decomposition in compatibility mode. In this mode, recomposition is not possible. This mode is useful, for example, when decomposing half-width and full-width *katakana* characters, where recomposition might not be desirable without external formatting or style information. It returns the string in NFKD normal form.

`CLOB` and `NCLOB` values are supported through implicit conversion. If *`char`* is a character LOB value, then it is converted to a `VARCHAR` value before the `COMPOSE` operation. The operation will fail if the size of the LOB value exceeds the supported length of the `VARCHAR` in the particular development environment.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode character sets and character semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `DECOMPOSE`
> - COMPOSE

Examples

The following example decomposes the string "`Château`x" into its component code points:

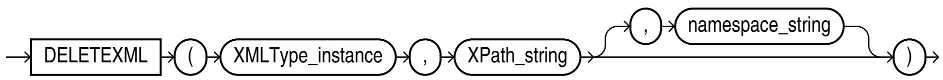
```
SELECT DECOMPOSE ('Châteaux')
  FROM DUAL;

DECOMPOSE
---------
Châteaux
```

> **Note:**
>
> The results of this example can vary depending on the character set of your operating system.

DELETEXML

> **Note:**
>
> The `DELETEXML` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`DELETEXML` deletes the node or nodes matched by the XPath expression in the target XML.

- *`XMLType_instance`* is an instance of `XMLType`.
- *`XPath_string`* is an Xpath expression indicating one or more nodes that are to be deleted. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node. Any child nodes of the nodes specified by *`XPath_string`* are also deleted.
- The optional *`namespace_string`* provides namespace information for the *`XPath_string`*. This parameter must be of type `VARCHAR2`.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

Examples

The following example removes the `/Owner` node from the `warehouse_spec` of one of the warehouses modified in the example for APPENDCHILDXML :

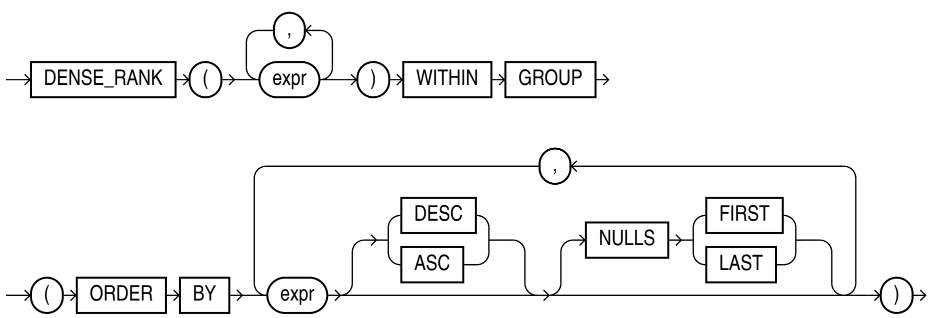
```
UPDATE warehouses
  SET warehouse_spec = DELETEXML(warehouse_spec, '/Warehouse/Building/Owner')
  WHERE warehouse_id = 2;

SELECT warehouse_id, warehouse_spec
  FROM warehouses  WHERE warehouse_id in (2,3);

        ID WAREHOUSE_SPEC
---------- -----------------------------------
         2 <?xml version="1.0"?>
           <Warehouse>
             <Building>Rented</Building>
```

```
    <Area>50000</Area>
    <Docks>1</Docks>
    <DockType>Side load</DockType>
    <WaterAccess>Y</WaterAccess>
    <RailAccess>N</RailAccess>
    <Parking>Lot</Parking>
    <VClearance>12 ft</VClearance>
  </Warehouse>

3 <?xml version="1.0"?>
  <Warehouse>
    <Building>Rented<Owner>Grandco</Owner>
    </Building>
    <Area>85700</Area>
    <DockType/>
    <WaterAccess>N</WaterAccess>
    <RailAccess>N</RailAccess>
    <Parking>Street</Parking>
    <VClearance>11.5 ft</VClearance>
  </Warehouse>
```

DENSE_RANK

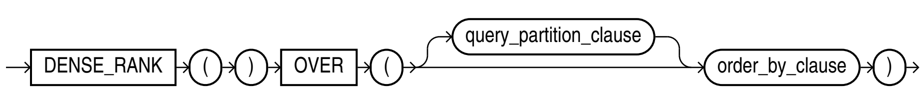
Aggregate Syntax

***dense_rank_aggregate*::=**



Analytic Syntax

***dense_rank_analytic*::=**



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

DENSE_RANK computes the rank of a row in an ordered group of rows and returns the rank as a NUMBER. The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query. Rank values are not skipped in the event of ties. Rows with equal values for the ranking criteria receive the same rank. This function is useful for top-N and bottom-N reporting.

This function accepts as arguments any numeric data type and returns NUMBER.

- As an aggregate function, DENSE_RANK calculates the dense rank of a hypothetical row identified by the arguments of the function with respect to a given sort specification. The arguments of the function must all evaluate to constant expressions within each aggregate group, because they identify a single row within each group. The constant argument expressions and the expressions in the *order_by_clause* of the aggregate match by position. Therefore, the number of arguments must be the same and types must be compatible.
- As an analytic function, DENSE_RANK computes the rank of each row returned from a query with respect to the other rows, based on the values of the *value_exprs* in the *order_by_clause*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation DENSE_RANK uses to compare character values for the ORDER BY clause

Aggregate Example

The following example computes the ranking of a hypothetical employee with the salary $15,500 and a commission of 5% in the sample table oe.employees:

```
SELECT DENSE_RANK(15500, .05) WITHIN GROUP
  (ORDER BY salary DESC, commission_pct) "Dense Rank"
  FROM employees;

Dense Rank
----------
         3
```

Analytic Example

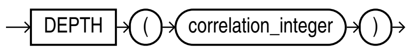
The following statement ranks the employees in the sample hr schema in department 60 based on their salaries. Identical salary values receive the same rank. However, no rank values are skipped. Compare this example with the analytic example for RANK .

```
SELECT department_id, last_name, salary,
       DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary) DENSE_RANK
  FROM employees WHERE department_id = 60
  ORDER BY DENSE_RANK, last_name;

DEPARTMENT_ID LAST_NAME                     SALARY DENSE_RANK
------------- ------------------------- ---------- ----------
           60 Lorentz                         4200          1
```

```
     60 Austin                            4800           2
     60 Pataballa                         4800           2
     60 Ernst                             6000           3
     60 Hunold                            9000           4
```

DEPTH

Syntax



Purpose

DEPTH is an ancillary function used only with the UNDER_PATH and EQUALS_PATH conditions. It returns the number of levels in the path specified by the UNDER_PATH condition with the same correlation variable.

The *correlation_integer* can be any NUMBER integer. Use it to correlate this ancillary function with its primary condition if the statement contains multiple primary conditions. Values less than 1 are treated as 1.

> **See Also:**
>
> EQUALS_PATH Condition , UNDER_PATH Condition , and the related function PATH

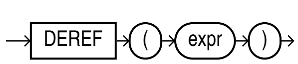
Examples

The EQUALS_PATH and UNDER_PATH conditions can take two ancillary functions, DEPTH and PATH. The following example shows the use of both ancillary functions. The example assumes the existence of the XMLSchema warehouses.xsd (created in Using XML in SQL Statements).

```
SELECT PATH(1), DEPTH(2)
  FROM RESOURCE_VIEW
  WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
    AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;

PATH(1)                            DEPTH(2)
-------------------------------- --------
. . .
www.example.com                           1
www.example.com/xwarehouses.xsd           2
. . .
```

DEREF

Syntax



Purpose

DEREF returns the object reference of argument *expr*, where *expr* must return a REF to an object. If you do not use this function in a query, then Oracle Database returns the object ID of the REF instead, as shown in the example that follows.

> **See Also:**
>
> MAKE_REF

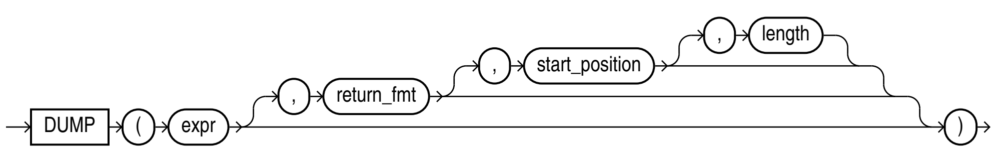
Examples

The sample schema oe contains an object type cust_address_typ. The REF Constraint Examples create a similar type, cust_address_typ_new, and a table with one column that is a REF to the type. The following example shows how to insert into such a column and how to use DEREF to extract information from the column:

```
INSERT INTO address_table VALUES
  ('1 First', 'G45 EU8', 'Paris', 'CA', 'US');

INSERT INTO customer_addresses
  SELECT 999, REF(a) FROM address_table a;

SELECT address
  FROM customer_addresses
  ORDER BY address;

ADDRESS
--------------------------------------------------------------------------------
000022020876B2245DBE325C5FE03400400B40DCB176B2245DBE305C5FE03400400B40DCB1

SELECT DEREF(address)
  FROM customer_addresses;

DEREF(ADDRESS)(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------------------
CUST_ADDRESS_TYP_NEW('1 First', 'G45 EU8', 'Paris', 'CA', 'US')
```

DUMP

Syntax



Purpose

DUMP returns a VARCHAR2 value containing the data type code, length in bytes, and internal representation of *expr*. The returned result is always in the database character set. For the data type corresponding to each code, see Table 2-1.

The argument *return_fmt* specifies the format of the return value and can have any of the following values:

- 8 returns result in octal notation.
- 10 returns result in decimal notation.
- 16 returns result in hexadecimal notation.
- 17 returns each byte printed as a character if and only if it can be interpreted as a printable character in the character set of the compiler—typically ASCII or EBCDIC. Some ASCII control characters may be printed in the form ^X as well. Otherwise the character is printed in hexadecimal notation. All NLS parameters are ignored. Do not depend on any particular output format for `DUMP` with *return_fmt* 17.

By default, the return value contains no character set information. To retrieve the character set name of *expr*, add 1000 to any of the preceding format values. For example, a *return_fmt* of 1008 returns the result in octal and provides the character set name of *expr*.

The arguments *start_position* and *length* combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

If *expr* is null, then this function returns `NULL`.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - Data Type Comparison Rules for more information
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `DUMP`

Examples

The following examples show how to extract dump information from a string expression and a column:

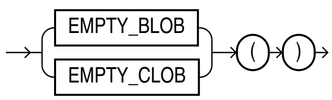
```
SELECT DUMP('abc', 1016)
  FROM DUAL;

DUMP('ABC',1016)
------------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63

SELECT DUMP(last_name, 8, 3, 2) "OCTAL"
  FROM employees
  WHERE last_name = 'Hunold'
  ORDER BY employee_id;

OCTAL
-------------------------------------------------------------------
```

```
Typ=1 Len=6: 156,157

SELECT DUMP(last_name, 10, 3, 2) "ASCII"
  FROM employees
  WHERE last_name = 'Hunold'
  ORDER BY employee_id;

ASCII
--------------------------------------------------------------------
Typ=1 Len=6: 110,111
```

EMPTY_BLOB, EMPTY_CLOB

Syntax

***empty_LOB*::=**



Purpose

`EMPTY_BLOB` and `EMPTY_CLOB` return an empty LOB locator that can be used to initialize a LOB variable or, in an `INSERT` or `UPDATE` statement, to initialize a LOB column or attribute to `EMPTY`. `EMPTY` means that the LOB is initialized, but not populated with data.

> **Note:**
>
> An empty LOB is not the same as a null LOB, and an empty `CLOB` is not the same as a LOB containing a string of 0 length. For more information, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `EMPTY_CLOB`

Restriction on LOB Locators

You cannot use the locator returned from this function as a parameter to the `DBMS_LOB` package or the OCI.

Examples

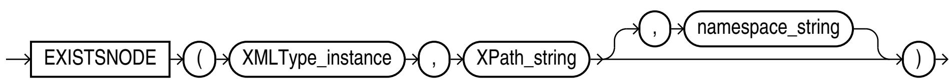
The following example initializes the `ad_photo` column of the sample `pm.print_media` table to `EMPTY`:

```
UPDATE print_media
  SET ad_photo = EMPTY_BLOB();
```

EXISTSNODE

> **Note:**
>
> The `EXISTSNODE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLEXISTS` function instead. See XMLEXISTS for more information.

Syntax



Purpose

`EXISTSNODE` determines whether traversal of an XML document using a specified path results in any nodes. It takes as arguments the `XMLType` instance containing an XML document and a `VARCHAR2` XPath string designating a path. The optional *`namespace_string`* must resolve to a `VARCHAR2` value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

The *`namespace_string`* argument defaults to the namespace of the root element. If you refer to any subelement in *`Xpath_string`*, then you must specify *`namespace_string`*, and you must specify the "who" prefix in both of these arguments.

> **See Also:**
>
> Using XML in SQL Statements for examples that specify *`namespace_string`* and use the "who" prefix.

The return value is `NUMBER`:

- 0 if no nodes remain after applying the XPath traversal on the document
- 1 if any nodes remain

Examples

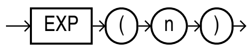
The following example tests for the existence of the `/Warehouse/Dock` node in the XML path of the `warehouse_spec` column of the sample table `oe.warehouses`:

```
SELECT warehouse_id, warehouse_name
  FROM warehouses
  WHERE EXISTSNODE(warehouse_spec, '/Warehouse/Docks') = 1
  ORDER BY warehouse_id;
```

```
WAREHOUSE_ID WAREHOUSE_NAME
------------ -----------------------------------
           1 Southlake, Texas
           2 San Francisco
           4 Seattle, Washington
```

EXP

Syntax



Purpose

`EXP` returns `e` raised to the *n*th power, where `e` = 2.71828183... . The function returns a value of the same type as the argument.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

Examples

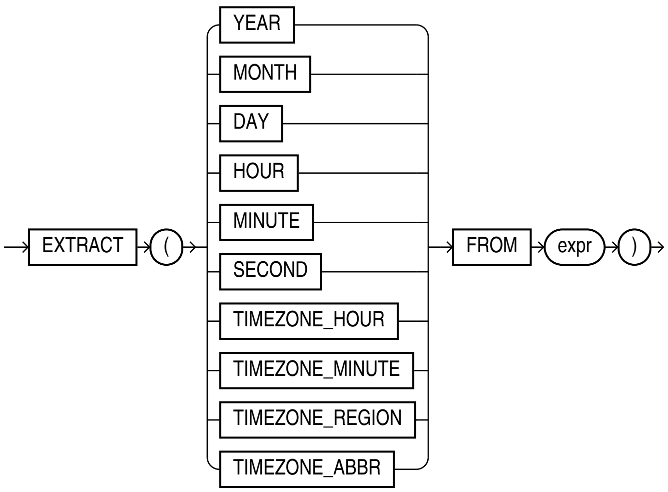
The following example returns `e` to the 4th power:

```
SELECT EXP(4) "e to the 4th power"
  FROM DUAL;

e to the 4th power
------------------
          54.59815
```

EXTRACT (datetime)

Syntax

***extract_datetime*::=**



Purpose

`EXTRACT` extracts and returns the value of a specified datetime field from a datetime or interval expression. The *`expr`* can be any expression that evaluates to a datetime or interval data type compatible with the requested field:

- If `YEAR` or `MONTH` is requested, then *`expr`* must evaluate to an expression of data type `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, or `INTERVAL YEAR TO MONTH`.
- If `DAY` is requested, then *`expr`* must evaluate to an expression of data type `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, or `INTERVAL DAY TO SECOND`.
- If `HOUR`, `MINUTE`, or `SECOND` is requested, then *`expr`* must evaluate to an expression of data type `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, or `INTERVAL DAY TO SECOND`. `DATE` is not valid here, because Oracle Database treats it as ANSI `DATE` data type, which has no time fields.
- If `TIMEZONE_HOUR`, `TIMEZONE_MINUTE`, `TIMEZONE_ABBR`, `TIMEZONE_REGION`, or `TIMEZONE_OFFSET` is requested, then *`expr`* must evaluate to an expression of data type `TIMESTAMP WITH TIME ZONE` or `TIMESTAMP WITH LOCAL TIME ZONE`.

`EXTRACT` interprets *`expr`* as an ANSI datetime data type. For example, `EXTRACT` treats `DATE` not as legacy Oracle `DATE` but as ANSI `DATE`, without time elements. Therefore, you can extract only `YEAR`, `MONTH`, and `DAY` from a `DATE` value. Likewise, you can extract `TIMEZONE_HOUR` and `TIMEZONE_MINUTE` only from the `TIMESTAMP WITH TIME ZONE` data type.

When you specify `TIMEZONE_REGION` or `TIMEZONE_ABBR` (abbreviation), the value returned is a `VARCHAR2` string containing the appropriate time zone region name or abbreviation. When you specify any of the other datetime fields, the value returned is an integer value of NUMBER data type representing the datetime value in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC. For a listing of time zone region names and their corresponding abbreviations, query the V$TIMEZONE_NAMES dynamic performance view.

This function can be very useful for manipulating datetime field values in very large tables, as shown in the first example below.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

Some combinations of datetime field and datetime or interval value expression result in ambiguity. In these cases, Oracle Database returns UNKNOWN (see the examples that follow for additional information).

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for a complete listing of the time zone region names in both files
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of EXTRACT
> - Datetime/Interval Arithmetic for a description of *datetime_value_expr* and *interval_value_expr*
> - *Oracle Database Reference* for information on the dynamic performance views

Examples

The following example returns from the oe.orders table the number of orders placed in each month:

```
SELECT EXTRACT(month FROM order_date) "Month", COUNT(order_date) "No. of Orders"
  FROM orders
  GROUP BY EXTRACT(month FROM order_date)
  ORDER BY "No. of Orders" DESC, "Month";

     Month No. of Orders
---------- -------------
        11            15
         6            14
         7            14
         3            11
         5            10
         2             9
```

```
         9              9
         8              7
        10              6
         1              5
        12              4
         4              1

12 rows selected.
```

The following example returns the year 1998.

```
SELECT EXTRACT(YEAR FROM DATE '1998-03-07')
  FROM DUAL;

EXTRACT(YEARFROMDATE'1998-03-07')
---------------------------------
                             1998
```

The following example selects from the sample table `hr.employees` all employees who were hired after 2007:

```
SELECT last_name, employee_id, hire_date
  FROM employees
  WHERE EXTRACT(YEAR FROM TO_DATE(hire_date, 'DD-MON-RR')) > 2007
  ORDER BY hire_date;

LAST_NAME                 EMPLOYEE_ID HIRE_DATE
------------------------- ----------- ---------
Johnson                           179 04-JAN-08
Grant                             199 13-JAN-08
Marvins                           164 24-JAN-08
. . .
```

The following example results in ambiguity, so Oracle returns `UNKNOWN`:

```
SELECT EXTRACT(TIMEZONE_REGION FROM TIMESTAMP '1999-01-01 10:00:00 -08:00')
  FROM DUAL;

EXTRACT(TIMEZONE_REGIONFROMTIMESTAMP'1999-01-0110:00:00-08:00')
----------------------------------------------------------------
UNKNOWN
```

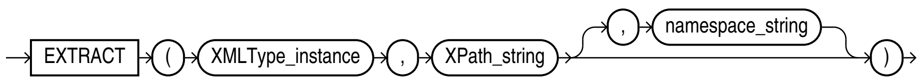
The ambiguity arises because the time zone numerical offset is provided in the expression, and that numerical offset may map to more than one time zone region name.

EXTRACT (XML)

> **Note:**
>
> The `EXTRACT` (XML) function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLQUERY` function instead. See XMLQUERY for more information.

Syntax

***extract_xml*::=**



Purpose

EXTRACT (XML) is similar to the EXISTSNODE function. It applies a VARCHAR2 XPath string and returns an XMLType instance containing an XML fragment. You can specify an absolute *XPath_string* with an initial slash or a relative *XPath_string* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node. The optional *namespace_string* is required if the XML you are handling uses a namespace prefix. This argument must resolve to a VARCHAR2 value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

Examples

The following example extracts the value of the /Warehouse/Dock node of the XML path of the warehouse_spec column in the sample table oe.warehouses:

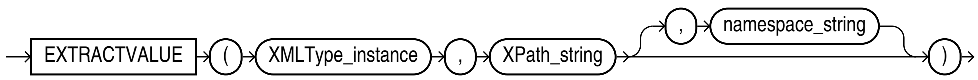
```
SELECT warehouse_name,
       EXTRACT(warehouse_spec, '/Warehouse/Docks') "Number of Docks"
  FROM warehouses
  WHERE warehouse_spec IS NOT NULL
  ORDER BY warehouse_name;

WAREHOUSE_NAME            Number of Docks
------------------------- ------------------------
New Jersey
San Francisco             <Docks>1</Docks>
Seattle, Washington       <Docks>3</Docks>
Southlake, Texas          <Docks>2</Docks>
```

Compare this example with the example for EXTRACTVALUE , which returns the scalar value of the XML fragment.

EXTRACTVALUE

> **Note:**
>
> The EXTRACTVALUE function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the XMLTABLE function, or the XMLCAST and XMLQUERY functions instead. See XMLTABLE , XMLCAST, and XMLQUERY for more information.

Syntax



The `EXTRACTVALUE` function takes as arguments an `XMLType` instance and an XPath expression and returns a scalar value of the resultant node. The result must be a single node and be either a text node, attribute, or element. If the result is an element, then the element must have a single text node as its child, and it is this value that the function returns. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, the context of the relative path defaults to the root node.

If the specified XPath points to a node with more than one child, or if the node pointed to has a non-text node child, then Oracle returns an error. The optional *`namespace_string`* must resolve to a `VARCHAR2` value that specifies a default mapping or namespace mapping for prefixes, which Oracle uses when evaluating the XPath expression(s).

For documents based on XML schemas, if Oracle can infer the type of the return value, then a scalar value of the appropriate type is returned. Otherwise, the result is of type `VARCHAR2`. For documents that are not based on XML schemas, the return type is always `VARCHAR2`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `EXTRACTVALUE`

Examples

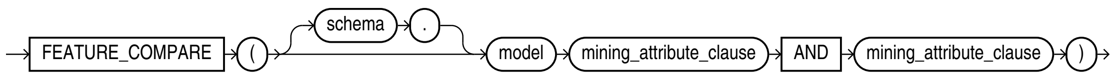
The following example takes as input the same arguments as the example for EXTRACT (XML) . Instead of returning an XML fragment, as does the `EXTRACT` function, it returns the scalar value of the XML fragment:

```
SELECT warehouse_name, EXTRACTVALUE(e.warehouse_spec, '/Warehouse/Docks') "Docks"
  FROM warehouses e
  WHERE warehouse_spec IS NOT NULL
  ORDER BY warehouse_name;

WAREHOUSE_NAME       Docks
-------------------- ------------
New Jersey
San Francisco        1
Seattle, Washington  3
Southlake, Texas     2
```

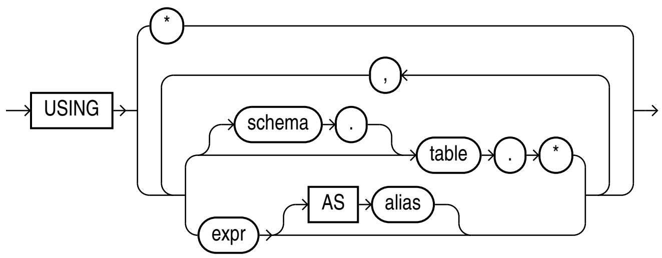
FEATURE_COMPARE

Syntax

***feature_compare*::=**



***mining_attribute_clause*::=**



Purpose

The `FEATURE_COMPARE` function uses a Feature Extraction model to compare two different documents, including short ones such as keyword phrases or two attribute lists, for similarity or dissimilarity. The `FEATURE_COMPARE` function can be used with Feature Extraction algorithms such as Singular Value Decomposition (SVD), Principal Component Analysis PCA), Non-Negative Matrix Factorization (NMF), and Explicit Semantic Analysis (ESA). This function is applicable not only to documents, but also to numeric and categorical data.

The input to the `FEATURE_COMPARE` function is a single feature model built using the Feature Extraction algorithms of Oracle Data Mining, such as NMF, SVD, and ESA. The double `USING` clause provides a mechanism to compare two different documents or constant keyword phrases, or any combination of the two, for similarity or dissimilarity using the extracted features in the model.

The syntax of the `FEATURE_COMPARE` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

The *`mining_attribute_clause`* identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. See mining_attribute_clause.

See Also:

- *Oracle Data Mining User's Guide* for information about scoring
- *Oracle Data Mining Concepts* for information about clustering

Note:

The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Examples

An ESA model is built against a 2005 Wiki dataset rendering over 200,000 features. The documents are mined as text and the document titles are considered as the Feature IDs.

The examples show the `FEATURE_COMPARE` function with the ESA algorithm, which compares a similar set of texts and then a dissimilar set of texts.

Similar texts

```
SELECT 1-FEATURE_COMPARE(esa_wiki_mod USING 'There are several PGA tour golfers from
South Africa' text AND USING 'Nick Price won the 2002 Mastercard Colonial Open' text)
similarity FROM DUAL;

SIMILARITY
----------
      .258
```

The output metric shows the results of a distance calculation. Therefore, a smaller number represents more similar texts. So 1 minus the distance in the queries represents a document similarity metric.

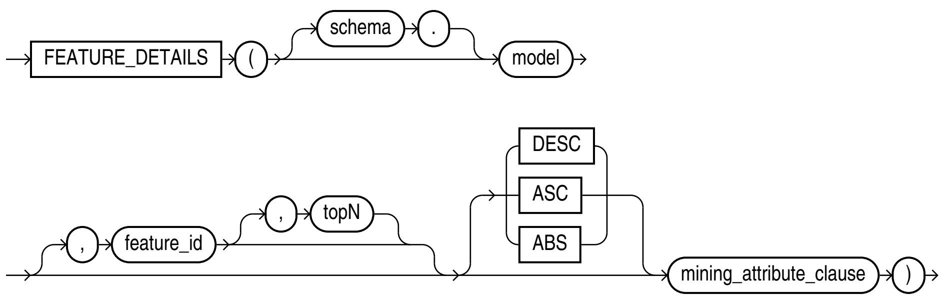
Dissimilar texts

```
SELECT 1-FEATURE_COMPARE(esa_wiki_mod USING 'There are several PGA tour golfers from
South Africa' text AND USING 'John Elway played quarterback for the Denver Broncos'
text) similarity FROM DUAL;

SIMILARITY
----------
      .007
```

FEATURE_DETAILS

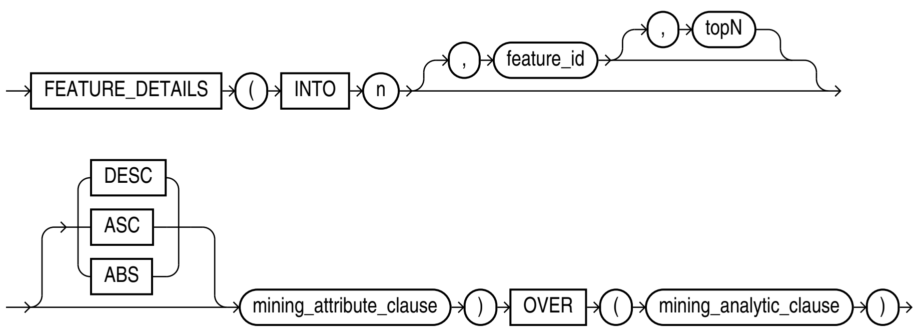
Syntax

***feature_details*::=**

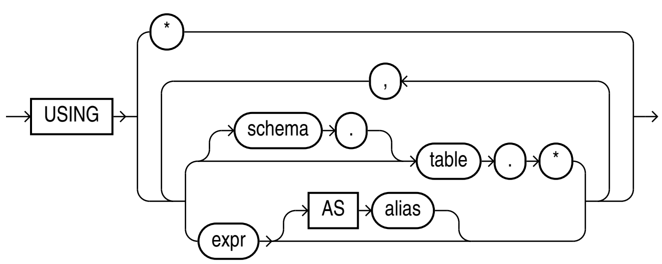


Analytic Syntax

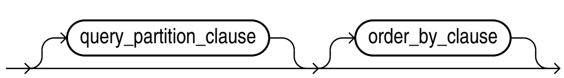
***feature_details_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



See Also:

"Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

FEATURE_DETAILS returns feature details for each row in the selection. The return value is an XML string that describes the attributes of the highest value feature or the specified *feature_id*.

topN

If you specify a value for *topN*, the function returns the *N* attributes that most influence the feature value. If you do not specify *topN*, the function returns the 5 most influential attributes.

DESC, ASC, or ABS

The returned attributes are ordered by weight. The weight of an attribute expresses its positive or negative impact on the value of the feature. A positive weight indicates a higher feature value. A negative weight indicates a lower feature value.

By default, FEATURE_DETAILS returns the attributes with the highest positive weight (DESC). If you specify ASC, the attributes with the highest negative weight are returned. If you specify ABS, the attributes with the greatest weight, whether negative or positive, are returned. The results are ordered by absolute value from highest to lowest. Attributes with a zero weight are not included in the output.

Syntax Choice

FEATURE_DETAILS can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of features to extract, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

The syntax of the FEATURE_DETAILS function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See "*mining_attribute_clause*::=".)

See Also:

- *Oracle Data Mining User's Guide* for information about scoring.
- *Oracle Data Mining Concepts* for information about feature extraction.

Note:

The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

This example uses the feature extraction model `nmf_sh_sample` to score the data. The query returns the three features that best represent customer 100002 and the attributes that most affect those features.

```
SELECT S.feature_id fid, value val,
       FEATURE_DETAILS(nmf_sh_sample, S.feature_id, 5 using T.*) det
   FROM
     (SELECT v.*, FEATURE_SET(nmf_sh_sample, 3 USING *) fset
         FROM mining_data_apply_v v
         WHERE cust_id = 100002) T,
   TABLE(T.fset) S
ORDER BY 2 DESC;

 FID    VAL  DET
---- ------  ----------------------------------------------------------------------------------
   5  3.492  <Details algorithm="Non-Negative Matrix Factorization" feature="5">
             <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".077" rank="1"/>
             <Attribute name="OCCUPATION" actualValue="Prof." weight=".062" rank="2"/>
             <Attribute name="BOOKKEEPING_APPLICATION" actualValue="1" weight=".001" rank="3"/>
             <Attribute name="OS_DOC_SET_KANJI" actualValue="0" weight="0" rank="4"/>
             <Attribute name="YRS_RESIDENCE" actualValue="4" weight="0" rank="5"/>
             </Details>
   3  1.928  <Details algorithm="Non-Negative Matrix Factorization" feature="3">
             <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".239" rank="1"/>
             <Attribute name="CUST_INCOME_LEVEL" actualValue="L: 300\,000 and above"
              weight=".051" rank="2"/>
             <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".02" rank="3"/>
             <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".006" rank="4"/>
             <Attribute name="AGE" actualValue="41" weight=".004" rank="5"/>
             </Details>
   8   .816  <Details algorithm="Non-Negative Matrix Factorization" feature="8">
             <Attribute name="EDUCATION" actualValue="Bach." weight=".211" rank="1"/>
             <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".143" rank="2"/>
             <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".137" rank="3"/>
             <Attribute name="CUST_GENDER" actualValue="F" weight=".044" rank="4"/>
             <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".032" rank="5"/>
             </Details>
```

Analytic Example

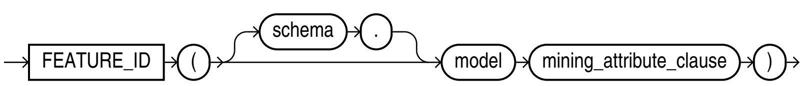
This example dynamically maps customer attributes into six features and returns the feature mapping for customer 100001.

```
SELECT feature_id, value
  FROM (
     SELECT cust_id, feature_set(INTO 6 USING *) OVER () fset
        FROM mining_data_apply_v),
  TABLE (fset)
  WHERE cust_id = 100001
  ORDER BY feature_id;

FEATURE_ID    VALUE
---------- --------
         1    2.670
         2     .000
         3    1.792
         4     .000
         5     .000
         6    3.379
```

FEATURE_ID

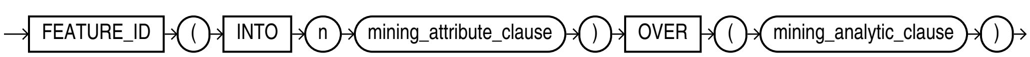
Syntax

***feature_id*::=**

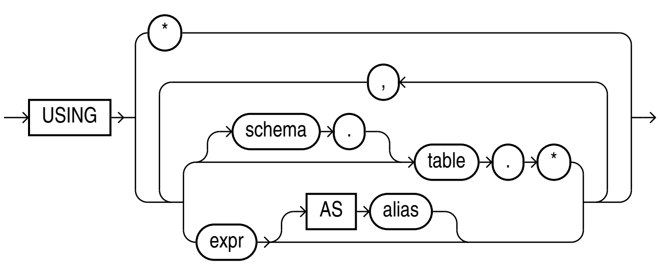


Analytic Syntax

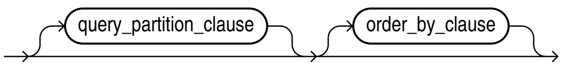
***feature_id_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of `mining_analytic_clause`

Purpose

`FEATURE_ID` returns the identifier of the highest value feature for each row in the selection. The feature identifier is returned as an Oracle `NUMBER`.

Syntax Choice

`FEATURE_ID` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *n*, where *n* is the number of features to extract, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

The syntax of the `FEATURE_ID` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about feature extraction.

Note:

The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

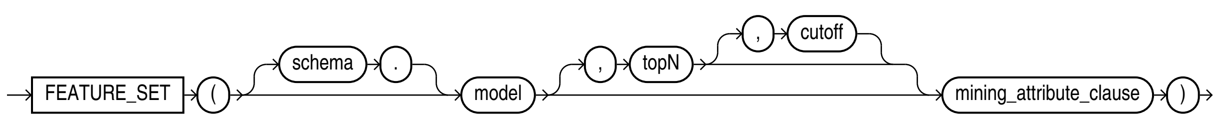
This example lists the features and corresponding count of customers in a data set.

```
SELECT FEATURE_ID(nmf_sh_sample USING *) AS feat, COUNT(*) AS cnt
  FROM nmf_sh_sample_apply_prepared
  GROUP BY FEATURE_ID(nmf_sh_sample USING *)
  ORDER BY cnt DESC, feat DESC;

      FEAT        CNT
---------- ----------
         7       1443
         2         49
         3          6
         6          1
         1          1
```

FEATURE_SET

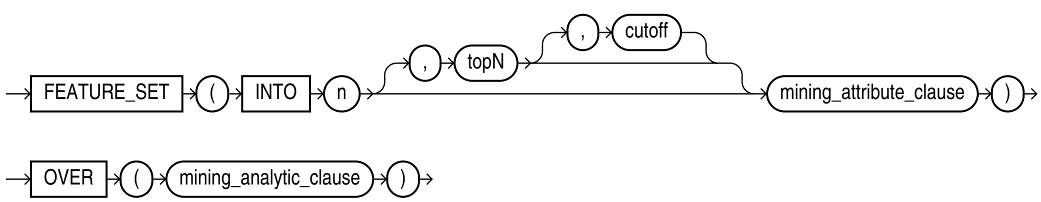
Syntax

***feature_set*::=**

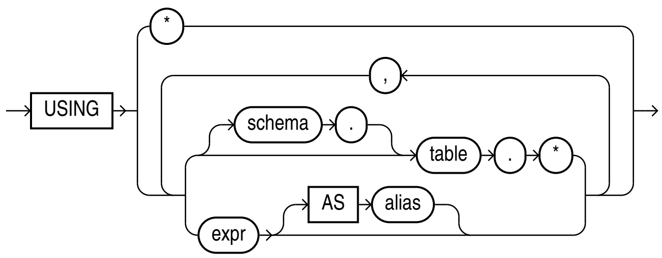


Analytic Syntax

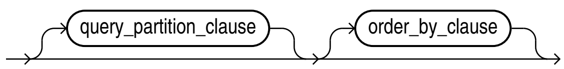
***feature_set_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

FEATURE_SET returns a set of feature ID and feature value pairs for each row in the selection. The return value is a varray of objects with field names FEATURE_ID and VALUE. The data type of both fields is NUMBER.

topN and cutoff

You can specify *topN* and *cutoff* to limit the number of features returned by the function. By default, both *topN* and *cutoff* are null and all features are returned.

- *topN* is the *N* highest value features. If multiple features have the *N*th value, then the function chooses one of them.
- *cutoff* is a value threshold. Only features that are greater than or equal to *cutoff* are returned. To filter by *cutoff* only, specify NULL for *topN*.

To return up to *N* features that are greater than or equal to *cutoff*, specify both *topN* and *cutoff*.

Syntax Choice

FEATURE_SET can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.

- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO n`, where *n* is the number of features to extract, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

The syntax of the `FEATURE_SET` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about feature extraction.

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

This example lists the top features corresponding to a given customer record and determines the top attributes for each feature (based on coefficient > 0.25).

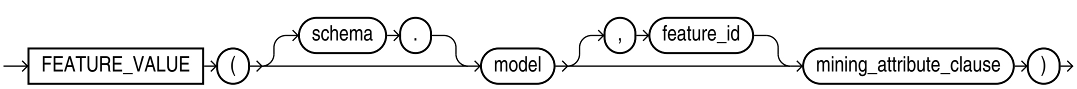
```
WITH
feat_tab AS (
SELECT F.feature_id fid,
       A.attribute_name attr,
       TO_CHAR(A.attribute_value) val,
       A.coefficient coeff
  FROM TABLE(DBMS_DATA_MINING.GET_MODEL_DETAILS_NMF('nmf_sh_sample')) F,
       TABLE(F.attribute_set) A
 WHERE A.coefficient > 0.25
),
feat AS (
SELECT fid,
       CAST(COLLECT(Featattr(attr, val, coeff))
         AS Featattrs) f_attrs
  FROM feat_tab
GROUP BY fid
),
cust_10_features AS (
SELECT T.cust_id, S.feature_id, S.value
  FROM (SELECT cust_id, FEATURE_SET(nmf_sh_sample, 10 USING *) pset
```

```
          FROM nmf_sh_sample_apply_prepared
         WHERE cust_id = 100002) T,
       TABLE(T.pset) S
)
SELECT A.value, A.feature_id fid,
       B.attr, B.val, B.coeff
  FROM cust_10_features A,
       (SELECT T.fid, F.*
          FROM feat T,
               TABLE(T.f_attrs) F) B
 WHERE A.feature_id = B.fid
ORDER BY A.value DESC, A.feature_id ASC, coeff DESC, attr ASC, val ASC;

   VALUE  FID ATTR                      VAL                       COEFF
-------- ---- ------------------------- ------------------------ -------
  6.8409    7 YRS_RESIDENCE                                       1.3879
  6.8409    7 BOOKKEEPING_APPLICATION                              .4388
  6.8409    7 CUST_GENDER               M                          .2956
  6.8409    7 COUNTRY_NAME              United States of America   .2848
  6.4975    3 YRS_RESIDENCE                                       1.2668
  6.4975    3 BOOKKEEPING_APPLICATION                              .3465
  6.4975    3 COUNTRY_NAME              United States of America   .2927
  6.4886    2 YRS_RESIDENCE                                       1.3285
  6.4886    2 CUST_GENDER               M                          .2819
  6.4886    2 PRINTER_SUPPLIES                                     .2704
  6.3953    4 YRS_RESIDENCE                                       1.2931
  5.9640    6 YRS_RESIDENCE                                       1.1585
  5.9640    6 HOME_THEATER_PACKAGE                                 .2576
  5.2424    5 YRS_RESIDENCE                                       1.0067
  2.4714    8 YRS_RESIDENCE                                        .3297
  2.3559    1 YRS_RESIDENCE                                        .2768
  2.3559    1 FLAT_PANEL_MONITOR                                   .2593
```

FEATURE_VALUE

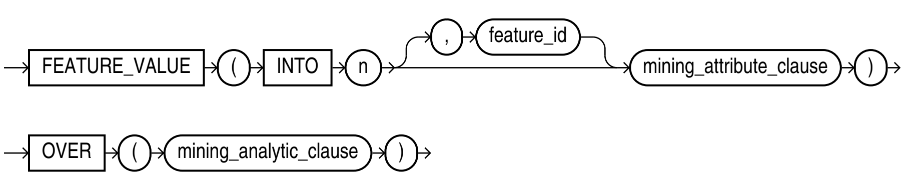
Syntax

***feature_value*::=**

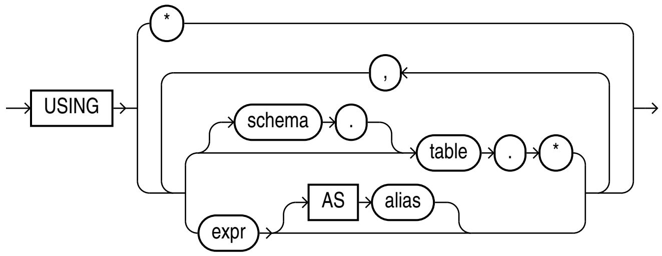


Analytic Syntax

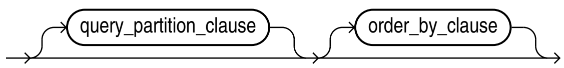
***feature_value_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

FEATURE_VALUE returns a feature value for each row in the selection. The value refers to the highest value feature or to the specified *feature_id*. The feature value is returned as BINARY_DOUBLE.

Syntax Choice

FEATURE_VALUE can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of features to extract, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

The syntax of the FEATURE_VALUE function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, this data is also used for building the transient models. The `mining_attribute_clause` behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about feature extraction.

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

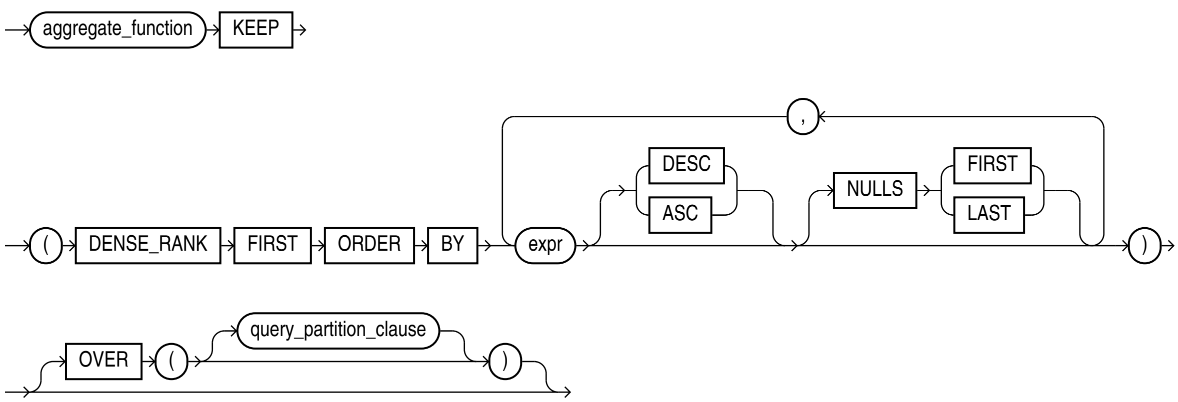
The following example lists the customers that correspond to feature 3, ordered by match quality.

```
SELECT *
  FROM (SELECT cust_id, FEATURE_VALUE(nmf_sh_sample, 3 USING *) match_quality
          FROM nmf_sh_sample_apply_prepared
          ORDER BY match_quality DESC)
  WHERE ROWNUM < 11;

   CUST_ID MATCH_QUALITY
---------- -------------
    100210    19.4101627
    100962    15.2482251
    101151    14.5685197
    101499    14.4186292
    100363    14.4037396
    100372    14.3335148
    100982    14.1716545
    101039    14.1079914
    100759    14.0913761
    100953    14.0799737
```

FIRST

Syntax

first::=



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `ORDER BY` clause and `OVER` clause

Purpose

`FIRST` and `LAST` are very similar functions. Both are aggregate and analytic functions that operate on a set of values from a set of rows that rank as the `FIRST` or `LAST` with respect to a given sorting specification. If only one row ranks as `FIRST` or `LAST`, then the aggregate operates on the set with only one element.

If you omit the `OVER` clause, then the `FIRST` and `LAST` functions are treated as aggregate functions. You can use these functions as analytic functions by specifying the `OVER` clause. The *`query_partition_clause`* is the only part of the `OVER` clause valid with these functions. If you include the `OVER` clause but omit the *`query_partition_clause`*, then the function is treated as an analytic function, but the window defined for analysis is the entire table.

These functions take as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

When you need a value from the first or last row of a sorted group, but the needed value is not the sort key, the `FIRST` and `LAST` functions eliminate the need for self-joins or views and enable better performance.

- The *`aggregate_function`* argument is any one of the `MIN`, `MAX`, `SUM`, `AVG`, `COUNT`, `VARIANCE`, or `STDDEV` functions. It operates on values from the rows that rank either `FIRST` or `LAST`. If only one row ranks as `FIRST` or `LAST`, then the aggregate operates on a singleton (nonaggregate) set.

- The `KEEP` keyword is for semantic clarity. It qualifies *`aggregate_function`*, indicating that only the `FIRST` or `LAST` values of *`aggregate_function`* will be returned.
- `DENSE_RANK FIRST` or `DENSE_RANK LAST` indicates that Oracle Database will aggregate over only those rows with the minimum (`FIRST`) or the maximum (`LAST`) dense rank (also called olympic rank).

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and LAST

Aggregate Example

The following example returns, within each department of the sample table `hr.employees`, the minimum salary among the employees who make the lowest commission and the maximum salary among the employees who make the highest commission:

```
SELECT department_id,
       MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct) "Worst",
       MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct) "Best"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

DEPARTMENT_ID      Worst       Best
------------- ---------- ----------
           10       4400       4400
           20       6000      13000
           30       2500      11000
           40       6500       6500
           50       2100       8200
           60       4200       9000
           70      10000      10000
           80       6100      14000
           90      17000      24000
          100       6900      12008
          110       8300      12008
                    7000       7000
```

Analytic Example

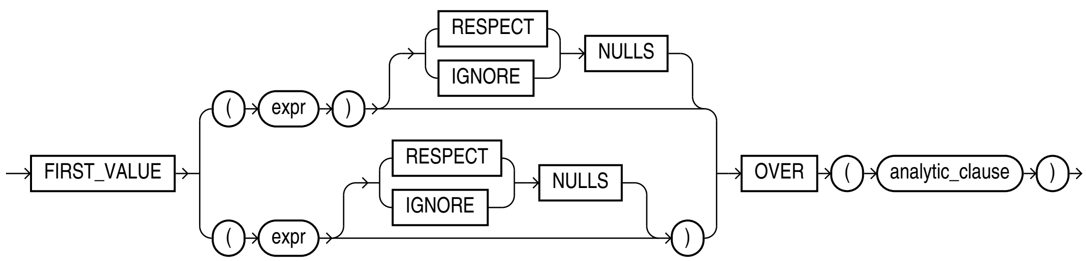
The next example makes the same calculation as the previous example but returns the result for each employee within the department:

```
SELECT last_name, department_id, salary,
       MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct)
         OVER (PARTITION BY department_id) "Worst",
       MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct)
         OVER (PARTITION BY department_id) "Best"
   FROM employees
   ORDER BY department_id, salary, last_name;

LAST_NAME           DEPARTMENT_ID     SALARY      Worst       Best
------------------- ------------- ---------- ---------- ----------
Whalen                         10       4400       4400       4400
Fay                            20       6000       6000      13000
Hartstein                      20      13000       6000      13000
```

```
. . .
Gietz                          110        8300        8300        12008
Higgins                        110       12008        8300        12008
Grant                                     7000        7000         7000
```

FIRST_VALUE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

FIRST_VALUE is an analytic function. It returns the first value in an ordered set of values. If the first value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. This setting is useful for data densification.

> **Note:**
>
> The two forms of this syntax have the same behavior. The top branch is the ANSI format, which Oracle recommends for ANSI compatibility.

{RESPECT | IGNORE} NULLS determines whether null values of *expr* are included in or eliminated from the calculation. The default is RESPECT NULLS. If you specify IGNORE NULLS, then FIRST_VALUE returns the first non-null value in the set, or NULL if all values are null. Refer to "Using Partitioned Outer Joins: Examples" for an example of data densification.

You cannot nest analytic functions by using FIRST_VALUE or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*. Refer to "About SQL Expressions " for information on valid forms of *expr*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `FIRST_VALUE` when it is a character value

Examples

The following example selects, for each employee in Department 90, the name of the employee with the lowest salary.

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);


EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        102 De Haan                        17000 13-JAN-01 De Haan
        101 Kochhar                        17000 21-SEP-05 De Haan
        100 King                           24000 17-JUN-03 De Haan
```

The example illustrates the nondeterministic nature of the `FIRST_VALUE` function. Kochhar and DeHaan have the same salary, so are in adjacent rows. Kochhar appears first because the rows returned by the subquery are ordered by `hire_date`. However, if the rows returned by the subquery are ordered by `hire_date` in descending order, as in the next example, then the function returns a different value:

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER by hire_date DESC);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        101 Kochhar                        17000 21-SEP-05 Kochhar
        102 De Haan                        17000 13-JAN-01 Kochhar
        100 King                           24000 17-JUN-03 Kochhar
```

The following two examples show how to make the `FIRST_VALUE` function deterministic by ordering on a unique key. By ordering within the function by both salary and the unique key `employee_id`, you can ensure the same result regardless of the ordering in the subquery.

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC, employee_id ROWS UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
```

```
        101 Kochhar                            17000 21-SEP-05 Kochhar
        102 De Haan                            17000 13-JAN-01 Kochhar
        100 King                               24000 17-JUN-03 Kochhar
```

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC, employee_id ROWS UNBOUNDED PRECEDING) AS fv
   FROM (SELECT * FROM employees
           WHERE department_id = 90
           ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        101 Kochhar                        17000 21-SEP-05 Kochhar
        102 De Haan                        17000 13-JAN-01 Kochhar
        100 King                           24000 17-JUN-03 Kochhar
```

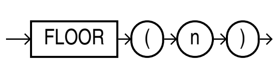
The following two examples show that the `FIRST_VALUE` function is deterministic when you use a logical offset (`RANGE` instead of `ROWS`). When duplicates are found for the `ORDER BY` expression, the `FIRST_VALUE` is the lowest value of *expr*:

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC RANGE UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        102 De Haan                        17000 13-JAN-01 De Haan
        101 Kochhar                        17000 21-SEP-05 De Haan
        100 King                           24000 17-JUN-03 De Haan
```

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC RANGE UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        102 De Haan                        17000 13-JAN-01 De Haan
        101 Kochhar                        17000 21-SEP-05 De Haan
        100 King                           24000 17-JUN-03 De Haan
```

FLOOR

Syntax



Purpose

FLOOR returns the largest integer equal to or less than n. The number n can always be written as the sum of an integer k and a positive fraction f such that $0 <= f < 1$ and $n = k + f$. The value of FLOOR is the integer k. Thus, the value of FLOOR is n itself if and only if n is precisely an integer.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and CEIL

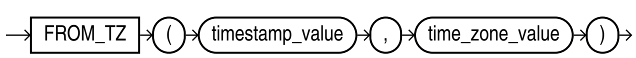
Examples

The following example returns the largest integer equal to or less than 15.7:

```
SELECT FLOOR(15.7) "Floor"
  FROM DUAL;

     Floor
----------
        15
```

FROM_TZ

Syntax



Purpose

FROM_TZ converts a timestamp value and a time zone to a TIMESTAMP WITH TIME ZONE value. *time_zone_value* is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR with optional TZD format.

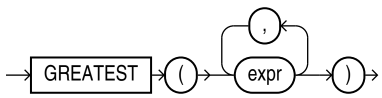
Examples

The following example returns a timestamp value to TIMESTAMP WITH TIME ZONE:

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
  FROM DUAL;

FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
---------------------------------------------------------------
28-MAR-00 08.00.000000000 AM +03:00
```

GREATEST

Syntax



Purpose

`GREATEST` returns the greatest of a list of one or more expressions. Oracle Database uses the first *`expr`* to determine the return type. If the first *`expr`* is numeric, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type before the comparison, and returns that data type. If the first *`expr`* is not numeric, then each *`expr`* after the first is implicitly converted to the data type of the first *`expr`* before the comparison.

Oracle Database compares each *`expr`* using nonpadded comparison semantics. The comparison is binary by default and is linguistic if the `NLS_COMP` parameter is set to `LINGUISTIC` and the `NLS_SORT` parameter has a setting other than `BINARY`. Character comparison is based on the numerical codes of the characters in the database character set and is performed on whole strings treated as one sequence of bytes, rather than character by character. If the value returned by this function is character data, then its data type is `VARCHAR2` if the first *`expr`* is a character data type and `NVARCHAR2` if the first *`expr`* is a national character data type.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information on character comparison
> - Table 2-8 for more information on implicit conversion and "Floating-Point Numbers " for information on binary-float comparison semantics
> - "LEAST ", which returns the least of a list of one or more expressions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `GREATEST` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following statement selects the string with the greatest value:

```
SELECT GREATEST('HARRY', 'HARRIOT', 'HAROLD') "Greatest"
  FROM DUAL;

Greatest
```

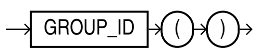
```
--------
HARRY
```

In the following statement, the first argument is numeric. Oracle Database determines that the argument with the highest numeric precedence is the second argument, converts the remaining arguments to the data type of the second argument, and returns the greatest value as that data type:

```
SELECT GREATEST (1, '3.925', '2.4') "Greatest"
  FROM DUAL;

Greatest
--------
   3.925
```

GROUP_ID

Syntax



Purpose

`GROUP_ID` distinguishes duplicate groups resulting from a `GROUP BY` specification. It is useful in filtering out duplicate groupings from the query result. It returns an Oracle `NUMBER` to uniquely identify duplicate groups. This function is applicable only in a `SELECT` statement that contains a `GROUP BY` clause.

If *n* duplicates exist for a particular grouping, then `GROUP_ID` returns numbers in the range 0 to *n*-1.

Examples

The following example assigns the value `1` to the duplicate `co.country_region` grouping from a query on the sample tables `sh.countries` and `sh.sales`:

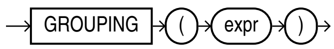
```
SELECT co.country_region, co.country_subregion,
       SUM(s.amount_sold) "Revenue", GROUP_ID() g
  FROM sales s, customers c, countries co
  WHERE s.cust_id = c.cust_id
    AND c.country_id = co.country_id
    AND s.time_id = '1-JAN-00'
    AND co.country_region IN ('Americas', 'Europe')
  GROUP BY GROUPING SETS ( (co.country_region, co.country_subregion),
                           (co.country_region, co.country_subregion) )
  ORDER BY co.country_region, co.country_subregion, "Revenue", g;

COUNTRY_REGION       COUNTRY_SUBREGION                 Revenue          G
-------------------- ------------------------------ ---------- ----------
Americas             Northern America                    944.6          0
Americas             Northern America                    944.6          1
Europe               Western Europe                     566.39          0
Europe               Western Europe                     566.39          1
```

To ensure that only rows with `GROUP_ID` < 1 are returned, add the following `HAVING` clause to the end of the statement :

```
HAVING GROUP_ID() < 1
```

GROUPING

Syntax



Purpose

`GROUPING` distinguishes superaggregate rows from regular grouped rows. `GROUP BY` extensions such as `ROLLUP` and `CUBE` produce superaggregate rows where the set of all values is represented by null. Using the `GROUPING` function, you can distinguish a null representing the set of all values in a superaggregate row from a null in a regular row.

The *`expr`* in the `GROUPING` function must match one of the expressions in the `GROUP BY` clause. The function returns a value of 1 if the value of *`expr`* in the row is a null representing the set of all values. Otherwise, it returns zero. The data type of the value returned by the `GROUPING` function is Oracle `NUMBER`. Refer to the `SELECT` *group_by_clause* for a discussion of these terms.

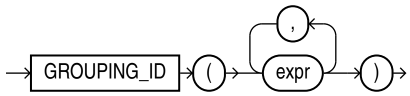
Examples

In the following example, which uses the sample tables `hr.departments` and `hr.employees`, if the `GROUPING` function returns 1 (indicating a superaggregate row rather than a regular row from the table), then the string "All Jobs" appears in the "JOB" column instead of the null that would otherwise appear:

```
SELECT
    DECODE(GROUPING(department_name), 1, 'ALL DEPARTMENTS', department_name)
      AS department,
    DECODE(GROUPING(job_id), 1, 'All Jobs', job_id) AS job,
    COUNT(*) "Total Empl",
    AVG(salary) * 12 "Average Sal"
  FROM employees e, departments d
  WHERE d.department_id = e.department_id
  GROUP BY ROLLUP (department_name, job_id)
  ORDER BY department, job;

DEPARTMENT                     JOB        Total Empl Average Sal
------------------------------ ---------- ---------- -----------
ALL DEPARTMENTS                All Jobs          106  77481.0566
Accounting                     AC_ACCOUNT          1       99600
Accounting                     AC_MGR              1      144096
Accounting                     All Jobs            2      121848
Administration                 AD_ASST             1       52800
Administration                 All Jobs            1       52800
Executive                      AD_PRES             1      288000
Executive                      AD_VP               2      204000
Executive                      All Jobs            3      232000
Finance                        All Jobs            6      103216
Finance                        FI_ACCOUNT          5       95040
. . .
```

GROUPING_ID

Syntax



Purpose

GROUPING_ID returns a number corresponding to the GROUPING bit vector associated with a row. GROUPING_ID is applicable only in a SELECT statement that contains a GROUP BY extension, such as ROLLUP or CUBE, and a GROUPING function. In queries with many GROUP BY expressions, determining the GROUP BY level of a particular row requires many GROUPING functions, which leads to cumbersome SQL. GROUPING_ID is useful in these cases.

GROUPING_ID is functionally equivalent to taking the results of multiple GROUPING functions and concatenating them into a bit vector (a string of ones and zeros). By using GROUPING_ID you can avoid the need for multiple GROUPING functions and make row filtering conditions easier to express. Row filtering is easier with GROUPING_ID because the desired rows can be identified with a single condition of GROUPING_ID = *n*. The function is especially useful when storing multiple levels of aggregation in a single table.

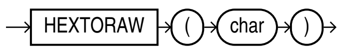
Examples

The following example shows how to extract grouping IDs from a query of the sample table sh.sales:

```
SELECT channel_id, promo_id, sum(amount_sold) s_sales,
       GROUPING(channel_id) gc,
       GROUPING(promo_id) gp,
       GROUPING_ID(channel_id, promo_id) gcp,
       GROUPING_ID(promo_id, channel_id) gpc
  FROM sales
  WHERE promo_id > 496
  GROUP BY CUBE(channel_id, promo_id)
  ORDER BY channel_id, promo_id, s_sales, gc;

CHANNEL_ID   PROMO_ID    S_SALES         GC         GP        GCP        GPC
---------- ---------- ---------- ---------- ---------- ---------- ----------
         2        999 25797563.2          0          0          0          0
         2            25797563.2          0          1          1          2
         3        999 55336945.1          0          0          0          0
         3            55336945.1          0          1          1          2
         4        999 13370012.5          0          0          0          0
         4            13370012.5          0          1          1          2
                  999 94504520.8          1          0          2          1
                      94504520.8          1          1          3          3
```

HEXTORAW

Syntax



Purpose

`HEXTORAW` converts *`char`* containing hexadecimal digits in the `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to a raw value.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information.

Examples

The following example creates a simple table with a raw column, and inserts a hexadecimal value that has been converted to `RAW`:

```
CREATE TABLE test (raw_col RAW(10));

INSERT INTO test VALUES (HEXTORAW('7D'));
```

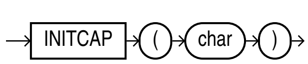
The following example converts hexadecimal digits to a raw value and casts the raw value to `VARCHAR2`:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('4041424344'))
  FROM DUAL;

UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('4041424344'))
------------------------------------------------
@ABCD
```

> **See Also:**
>
> "RAW and LONG RAW Data Types " and RAWTOHEX

INITCAP

Syntax



Purpose

`INITCAP` returns *`char`*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

`char` can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The return value is the same data type as *`char`*. The database sets the case of the initial characters based on the binary mapping defined for the underlying character set. For linguistic-sensitive uppercase and lowercase, refer to NLS_INITCAP .

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `INITCAP`

Examples

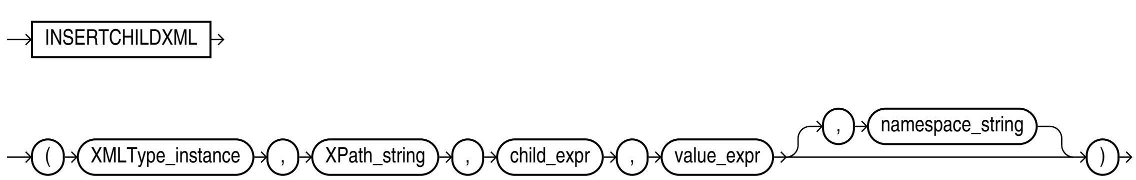
The following example capitalizes each word in the string:

```
SELECT INITCAP('the soap') "Capitals"
  FROM DUAL;

Capitals
---------
The Soap
```

INSERTCHILDXML

> **Note:**
>
> The `INSERTCHILDXML` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`INSERTCHILDXML` inserts a user-supplied value into the target XML at the node indicated by the XPath expression. Compare this function with INSERTXMLBEFORE .

- *`XMLType_instance`* is an instance of `XMLType`.
- *`XPath_string`* is an Xpath expression indicating one or more nodes into which the one or more child nodes are to be inserted. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node.
- *`child_expr`* specifies the one or more element or attribute nodes to be inserted.
- *`value_expr`* is an fragment of `XMLType` that specifies one or more notes being inserted. It must resolve to a string.
- The optional *`namespace_string`* provides namespace information for the *`XPath_string`*. This parameter must be of type `VARCHAR2`.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

Examples

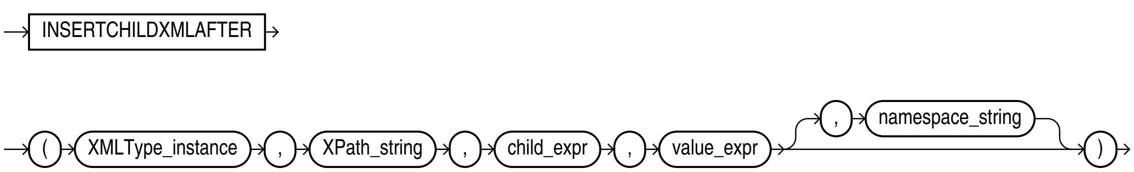
The following example adds a second `/Owner` node to the `warehouse_spec` of one of the warehouses updated in the example for APPENDCHILDXML :

```
UPDATE warehouses
  SET warehouse_spec = INSERTCHILDXML(warehouse_spec, '/Warehouse/Building',
    'Owner', XMLType('<Owner>LesserCo</Owner>'))
  WHERE warehouse_id = 3;

SELECT warehouse_spec
  FROM warehouses  WHERE warehouse_id = 3;

WAREHOUSE_SPEC
----------------------------------------------------------------------------
<?xml version="1.0"?>
<Warehouse>
  <Building>Rented
    <Owner>Grandco</Owner>
    <Owner>LesserCo</Owner>
  </Building>
  <Area>85700</Area>
  <DockType/>
  <WaterAccess>N</WaterAccess>
  <RailAccess>N</RailAccess>
  <Parking>Street</Parking>
  <VClearance>11.5 ft</VClearance>
</Warehouse>
```

INSERTCHILDXMLAFTER

> **Note:**
>
> The `INSERTCHILDXMLAFTER` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`INSERTXMLCHILDAFTER` inserts one or more collection elements as children of target parent elements. The insertion for each target occurs immediately after a specified existing collection element. The existing XML document that is the target of the insertion can be schema-based or non-schema-based.

- *`XMLType_instance`* identifies the XML data that is the target of the insertion.
- *`XPath_string`* locates the parent elements within target-data; child-data is inserted under *each* parent element.
- *`child_expr`* is a relative XPath 1.0 expression that locates the existing child that will precede the inserted child-data. It must name a child element of the element indicated by parent-xpath, and it can include a predicate.
- *`value_expr`* is the `XMLType` child element data to insert. Each top-level element node in this argument must have the same data type as the element indicated by child_expr.
- The optional *`namespace_string`* specifies the namespace for the parent elements, existing child element, and child element XML data to be inserted.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

Examples

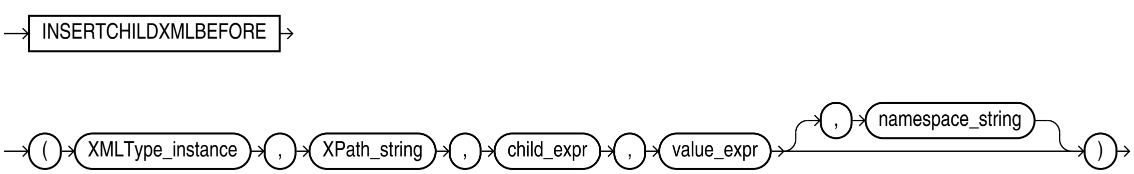
The following example is similar to that for `INSERTCHILDXML`, but it adds a third `/Owner` node after the `/Owner` node added in the other example. The output of the query has been formatted for readability.

```
UPDATE warehouses
  SET warehouse_spec = INSERTCHILDXMLAFTER(warehouse_spec, '/Warehouse/Building',
    'Owner[2]', XMLType('<Owner>ThirdOwner</Owner>'))
  WHERE warehouse_id = 3;

SELECT warehouse_name,
       EXTRACT(warehouse_spec, '/Warehouse/Building/Owner') "Owners"
  FROM warehouses
  WHERE warehouse_id = 3;

WAREHOUSE_NAME                     Owners
----------------------------------- ------------------------------
New Jersey                         <Owner>GrandCo</Owner>
                                   <Owner>LesserCo</Owner>
                                   <Owner>ThirdOwner</Owner>
```

INSERTCHILDXMLBEFORE

> **Note:**
>
> The `INSERTCHILDXMLBEFORE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`INSERTXMLCHILDBEFORE` inserts one or more collection elements as children of target parent elements. The insertion for each target occurs immediately before a specified existing collection element. The existing XML document that is the target of the insertion can be schema-based or non-schema-based.

- *`XMLType_instance`* identifies the XML data that is the target of the insertion.
- *`XPath_string`* locates the parent elements within target-data; child-data is inserted under *each* parent element.
- *`child_expr`* is a relative XPath 1.0 expression that locates the existing child that will follow the inserted child-data. It must name a child element of the element indicated by parent-xpath, and it can include a predicate.
- *`value_expr`* is the `XMLType` child element data to insert. Each top-level element node in this argument must have the same data type as the element indicated by child_expr.

- The optional *namespace_string* specifies the namespace for the parent elements, existing child element, and child element XML data to be inserted.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

Examples

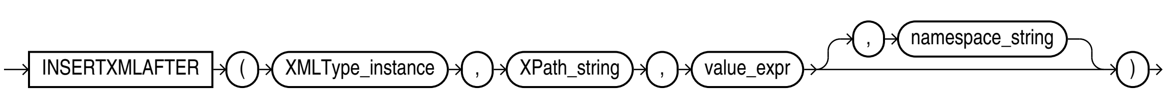
The following example is similar to that for `INSERTCHILDXML`, but it adds a third `/Owner` node before the `/Owner` node added in the other example. The output of the query has been formatted for readability.

```
UPDATE warehouses
  SET warehouse_spec = INSERTCHILDXMLBEFORE(warehouse_spec, '/Warehouse/Building',
    'Owner[2]', XMLType('<Owner>ThirdOwner</Owner>'))
  WHERE warehouse_id = 3;

SELECT warehouse_name,
       EXTRACT(warehouse_spec, '/Warehouse/Building/Owner') "Owners"
  FROM warehouses
  WHERE warehouse_id = 3;

WAREHOUSE_NAME                       Owners
----------------------------------- ------------------------------
New Jersey                           <Owner>GrandCo</Owner>
                                     <Owner>ThirdOwner</Owner>
                                     <Owner>LesserCo</Owner>
```

INSERTXMLAFTER

> **Note:**
>
> The `INSERTXMLAFTER` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`INSERTXMLAFTER` inserts one or more nodes of any kind immediately after a target node that is not an attribute node. The XML document that is the target of the insertion can be schema-based or non-schema-based. This function is similar to insertXMLbefore, but it inserts after, not before, the target node.

- *XMLType_instance* specifies the target node of the of the insertion.
- *XPath_string* is an XPath 1.0 expression that locates in the target node zero or more nodes of any kind except attribute nodes. XML-data is inserted immediately after each of these nodes; that is, each node specified becomes the preceding sibling node of a node specified in *value_expr*.
- *value_expr* is the XML data to be inserted. You can specify one or more nodes of any kind. The order of the nodes is preserved after the insertion.
- The optional *namespace_string* is the namespace for the target node.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

Examples

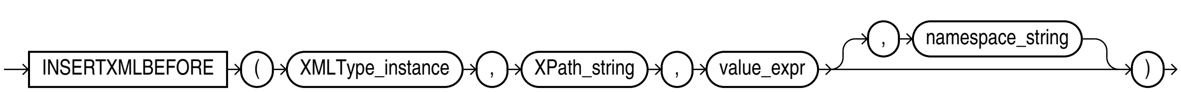
The following example is similar to that for `INSERTCHILDXML`, but it adds a third `/Owner` node after the `/Owner` node added in the other example. The output of the query has been formatted for readability.

```
UPDATE warehouses
  SET warehouse_spec = INSERTXMLAFTER(warehouse_spec,
    '/Warehouse/Building/Owner[1]', XMLType('<Owner>SecondOwner</Owner>'))
  WHERE warehouse_id = 3;

SELECT warehouse_name,
       EXTRACT(warehouse_spec, '/Warehouse/Building/Owner') "Owners"
  FROM warehouses
  WHERE warehouse_id = 3;

WAREHOUSE_NAME                      Owners
----------------------------------- ------------------------------
New Jersey                          <Owner>GrandCo</Owner>
                                    <Owner>SecondOwner</Owner>
                                    <Owner>LesserCo</Owner>
```

INSERTXMLBEFORE

> **Note:**
>
> The `INSERTXMLBEFORE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`INSERTXMLBEFORE` inserts a user-supplied value into the target XML *before* the node indicated by the XPath expression. This function is similar to INSERTXMLAFTER, but it inserts before, not after, the target node. Compare this function with INSERTCHILDXML .

- *`XMLType_instance`* is an instance of `XMLType`.
- *`XPath_string`* is an Xpath expression indicating one or more nodes into which one or more child nodes are to be inserted. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node.
- *`value_expr`* is a fragment of `XMLType` that defines one or more nodes being inserted and their position within the parent node. It must resolve to a string.
- The optional *`namespace_string`* provides namespace information for the *`XPath_string`*. This parameter must be of type `VARCHAR2`.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information about this function

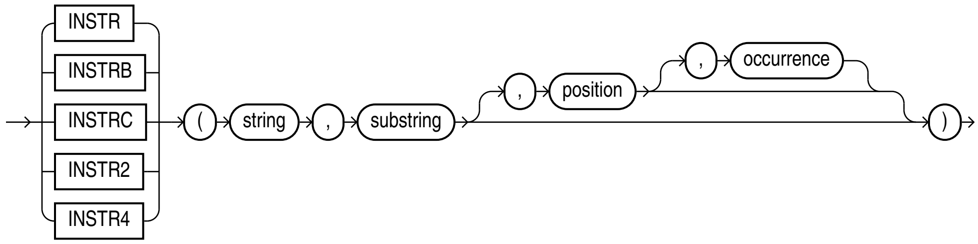
Examples

The following example is similar to that for INSERTCHILDXML , but it adds a third `/Owner` node *before* the `/Owner` node added in the other example. The output of the query has been formatted for readability.

```
UPDATE warehouses
  SET warehouse_spec = INSERTXMLBEFORE(warehouse_spec,
    '/Warehouse/Building/Owner[2]', XMLType('<Owner>ThirdOwner</Owner>'))
  WHERE warehouse_id = 3;

SELECT warehouse_name,
       EXTRACT(warehouse_spec, '/Warehouse/Building/Owner') "Owners"
  FROM warehouses
  WHERE warehouse_id = 3;

WAREHOUSE_NAME                      Owners
----------------------------------- ------------------------------
New Jersey                          <Owner>GrandCo</Owner>
                                    <Owner>ThirdOwner</Owner>
                                    <Owner>LesserCo</Owner>
```

INSTR

Syntax



Purpose

The `INSTR` functions search *string* for *substring*. The search operation is defined as comparing the *substring* argument with substrings of *string* of the same length for equality until a match is found or there are no more substrings left. Each consecutive compared substring of *string* begins one character to the right (for forward searches) or one character to the left (for backward searches) from the first character of the previous compared substring. If a substring that is equal to *substring* is found, then the function returns an integer indicating the position of the first character of this substring. If no such substring is found, then the function returns zero.

- *position* is an nonzero integer indicating the character of *string* where Oracle Database begins the search—that is, the position of the first character of the first substring to compare with *substring*. If *position* is negative, then Oracle counts backward from the end of *string* and then searches backward from the resulting position.
- *occurrence* is an integer indicating which occurrence of *substring* in *string* Oracle should search for. The value of *occurrence* must be positive. If *occurrence* is greater than 1, then the database does not return on the first match but continues comparing consecutive substrings of *string*, as described above, until match number *occurrence* has been found.

`INSTR` accepts and returns positions in characters as defined by the input character set, with the first character of string having position 1. `INSTRB` uses bytes instead of characters. `INSTRC` uses Unicode complete characters. `INSTR2` uses UCS2 code points. `INSTR4` uses UCS4 code points.

string can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The exceptions are `INSTRC`, `INSTR2`, and `INSTR4`, which do not allow *string* to be a `CLOB` or `NCLOB`.

substring can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.

The value returned is of `NUMBER` data type.

Both *position* and *occurrence* must be of data type `NUMBER`, or any data type that can be implicitly converted to `NUMBER`, and must resolve to an integer. The default values of both *position* and *occurrence* are 1, meaning Oracle begins searching at the first character of *string* for the first occurrence of *substring*. The return value is relative to the beginning of *string*, regardless of the value of *position*.

> **See Also:**
>
> - Table 2-8 for more information on implicit conversion
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation the `INSTR` functions use to compare the *`substring`* argument with substrings of *`string`*

Examples

The following example searches the string `CORPORATE FLOOR`, beginning with the third character, for the string "`OR`". It returns the position in `CORPORATE FLOOR` at which the second occurrence of "`OR`" begins:

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2) "Instring"
  FROM DUAL;

  Instring
----------
        14
```

In the next example, Oracle counts backward from the last character to the third character from the end, which is the first `O` in `FLOOR`. Oracle then searches backward for the second occurrence of `OR`, and finds that this second occurrence begins with the second character in the search string :

```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2) "Reversed Instring"
  FROM DUAL;

Reversed Instring
-----------------
                2
```

The next example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes"
  FROM DUAL;

Instring in bytes
-----------------
               27
```

ITERATION_NUMBER

Syntax

Purpose

The `ITERATION_NUMBER` function can be used only in the *`model_clause`* of the `SELECT` statement and then only when `ITERATE(`*`number`*`)` is specified in the *`model_rules_clause`*. It returns an integer representing the completed iteration through the model rules. The

ITERATION_NUMBER function returns 0 during the first iteration. For each subsequent iteration, the ITERATION_NUMBER function returns the equivalent of *iteration_number* plus one.

See Also:

model_clause and "Model Expressions" for the syntax and semantics

Examples

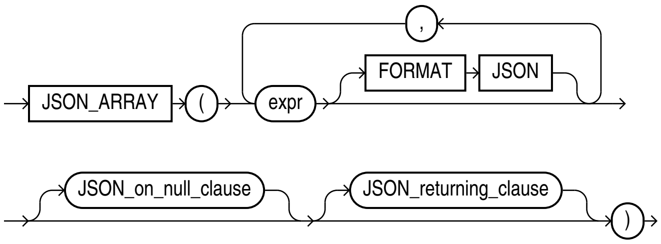
The following example assigns the sales of the Mouse Pad for the years 1998 and 1999 to the sales of the Mouse Pad for the years 2001 and 2002 respectively:

```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER ITERATE(2)
      (
        s['Mouse Pad', 2001 + ITERATION_NUMBER] =
          s['Mouse Pad', 1998 + ITERATION_NUMBER]
      )
  ORDER BY country, prod, year;

COUNTRY       PROD                                   YEAR          S
----------    -----------------------------------    --------   ---------
France        Mouse Pad                              1998     2509.42
France        Mouse Pad                              1999     3678.69
France        Mouse Pad                              2000     3000.72
France        Mouse Pad                              2001     2509.42
France        Mouse Pad                              2002     3678.69
France        Standard Mouse                         1998     2390.83
France        Standard Mouse                         1999     2280.45
France        Standard Mouse                         2000     1274.31
France        Standard Mouse                         2001     2164.54
Germany       Mouse Pad                              1998     5827.87
Germany       Mouse Pad                              1999     8346.44
Germany       Mouse Pad                              2000     7375.46
Germany       Mouse Pad                              2001     5827.87
Germany       Mouse Pad                              2002     8346.44
Germany       Standard Mouse                         1998     7116.11
Germany       Standard Mouse                         1999     6263.14
Germany       Standard Mouse                         2000     2637.31
Germany       Standard Mouse                         2001     6456.13

18 rows selected.
```

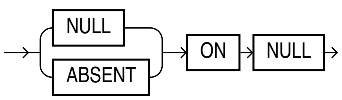
The preceding example requires the view sales_view_ref. Refer to "The MODEL clause: Examples" to create this view.

JSON_ARRAY

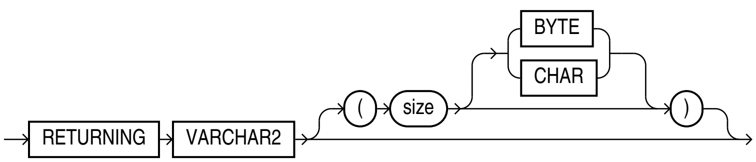
Syntax



JSON_on_null_clause::=



JSON_returning_clause::=



Purpose

The SQL/JSON function `JSON_ARRAY` takes as its input one or more SQL expressions, converts each expression to a JSON value, and returns a JSON array that contains those JSON values.

expr

For *`expr`*, you can specify any SQL expression that evaluates to a JSON object, a JSON array, a numeric literal, a text literal, or null. This function converts a numeric literal to a JSON number value and a text literal to a JSON string value.

FORMAT JSON

This clause is optional and is provided for semantic clarity.

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - If you specify this clause, then the function returns the JSON null value.

- `ABSENT ON NULL` - If you specify this clause, then the function omits the value from the JSON array. This is the default.

JSON_returning_clause

The character string returned by this function is of data type `VARCHAR2`. This clause allows you to specify the size of the `VARCHAR2` data type. Use `BYTE` to specify the size as a number of bytes or `CHAR` to specify the size as a number of characters. The default is `BYTE`. If you omit this clause, or if you specify this clause but omit the *`size`* value, then `JSON_ARRAY` returns a character string of type `VARCHAR2(4000)`.

Refer to VARCHAR2 Data Type for more information. Note that when specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in the *`JSON_returning_clause`* you can omit the size.

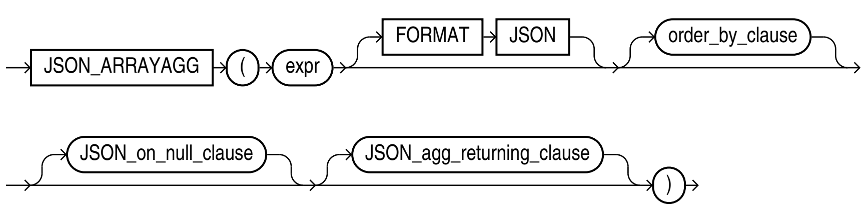
Examples

The following example constructs a JSON array from a JSON object, a JSON array, a numeric literal, a text literal, and null:

```
SELECT JSON_ARRAY (
    JSON_OBJECT('percentage' VALUE .50),
    JSON_ARRAY(1,2,3),
    100,
    'California',
    null
    NULL ON NULL
    ) "JSON Array Example"
  FROM DUAL;

JSON Array Example
--------------------------------------------------------------------------------
[{"percentage":0.5},[1,2,3],100,"California",null]
```

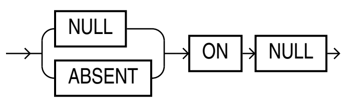
JSON_ARRAYAGG

Syntax

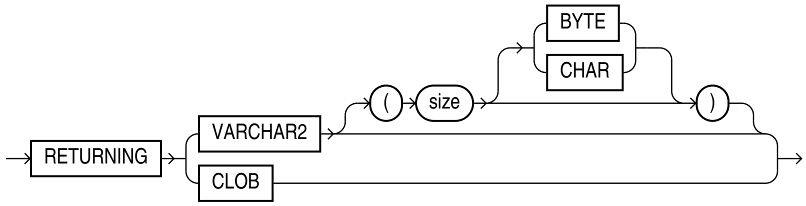


(See *order_by_clause*::= in the documentation on `SELECT` for the syntax of this clause)

***JSON_on_null_clause*::=**



***JSON_agg_returning_clause*::=**



Purpose

The SQL/JSON function `JSON_ARRAYAGG` is an aggregate function. It takes as its input a column of SQL expressions, converts each expression to a JSON value, and returns a single JSON array that contains those JSON values.

expr

For *`expr`*, you can specify any SQL expression that evaluates to a JSON object, a JSON array, a numeric literal, a text literal, or null. This function converts a numeric literal to a JSON number value and a text literal to a JSON string value.

FORMAT JSON

This clause is optional and is provided for semantic clarity.

order_by_clause

This clause allows you to order the JSON values within the JSON array returned by the statement. Refer to the *order_by_clause* in the documentation on `SELECT` for the full semantics of this clause.

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - If you specify this clause, then the function returns the JSON null value.
- `ABSENT ON NULL` - If you specify this clause, then the function omits the value from the JSON array. This is the default.

JSON_agg_returning_clause

Use this clause to specify the data type of the character string returned by this function. You can specify the following data types:

- `VARCHAR2[(`*`size`*` [BYTE,CHAR])]`

  When specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size.
- `CLOB`

If you omit this clause, or if you specify `VARCHAR2` but omit the *`size`* value, then `JSON_ARRAYAGG` returns a character string of type `VARCHAR2(4000)`.

Refer to "Data Types " for more information on the preceding data types.

Examples

The following statements creates a table `id_table`, which contains ID numbers:

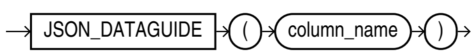
```
CREATE TABLE id_table (id NUMBER);
INSERT INTO id_table VALUES(624);
INSERT INTO id_table VALUES(null);
INSERT INTO id_table VALUES(925);
INSERT INTO id_table VALUES(585);
```

The following example constructs a JSON array from the ID numbers in table `id_table`:

```
SELECT JSON_ARRAYAGG(id ORDER BY id RETURNING VARCHAR2(100)) ID_NUMBERS
  FROM id_table;

ID_NUMBERS
-------------
[585,624,925]
```

JSON_DATAGUIDE

Syntax



Purpose

The Oracle SQL function `JSON_DATAGUIDE` takes as its input a table column of JSON data. Each row in the column is referred to as a JSON document. For each JSON document in the column, this function returns a `CLOB` value that contains a flat data guide for that JSON document.

column_name

Specify the name of the column of JSON data for which you would like to create data guides.

Restriction on JSON_DATAGUIDE

You cannot run this function on a shard catalog server.

> **See Also:**
>
> *Oracle Database JSON Developer's Guide* for more information on data guides

Examples

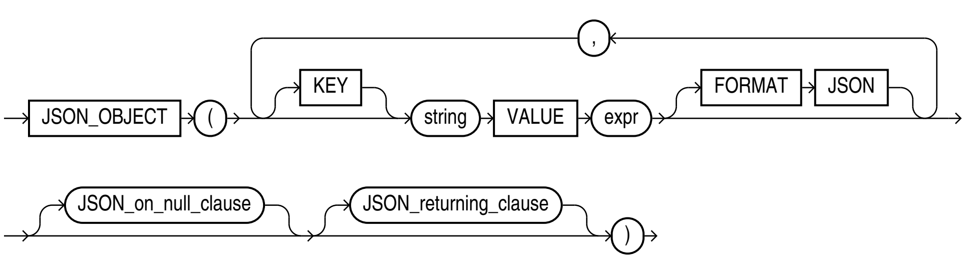
The following example uses the `j_purchaseorder` table, which is created in "Creating a Table That Contains a JSON Document: Example". This table contains a column of

JSON data called `po_document`. This examples returns a flat data guide for each JSON document in the column `po_document`.

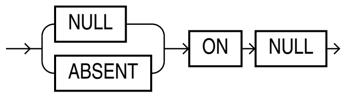
```
SELECT EXTRACT(YEAR FROM date_loaded) YEAR,
       JSON_DATAGUIDE(po_document) "DATA GUIDE"
  FROM j_purchaseorder
  GROUP BY extract(YEAR FROM date_loaded)
  ORDER BY extract(YEAR FROM date_loaded) DESC;


YEAR DATA GUIDE
---- ------------------------------------------
2016 [
       {
         "o:path" : "$.PO_ID",
         "type" : "number",
         "o:length" : 4
       },
       {
         "o:path" : "$.PO_Ref",
         "type" : "string",
         "o:length" : 16
       },
       {
         "o:path" : "$.PO_Items",
         "type" : "array",
         "o:length" : 64
       },
       {
         "o:path" : "$.PO_Items.Part_No",
         "type" : "number",
         "o:length" : 16
       },
       {
         "o:path" : "$.PO_Items.Item_Quantity",
         "type" : "number",
         "o:length" : 2
       }
     ]
. . .
```

JSON_OBJECT

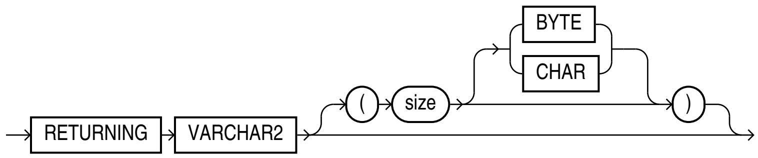
Syntax



***JSON_on_null_clause*::=**



***JSON_returning_clause*::=**



Purpose

The SQL/JSON function `JSON_OBJECT` takes as its input one or more property key-value pairs. It returns a JSON object that contains an object member for each of those key-value pairs.

[KEY] *string* VALUE *expr*

Use this clause to specify a property key-value pair.

- `KEY` is optional and is provided for semantic clarity.
- Use *`string`* to specify the property key name as a case-sensitive text literal.
- Use *`expr`* to specify the property value. For *`expr`*, you can specify any expression that evaluates to a SQL numeric literal or text literal. If *`expr`* evaluates to a numeric literal, then the resulting property value is a JSON number value; otherwise, the resulting property value is a case-sensitive JSON string value enclosed in double quotation marks.

FORMAT JSON

Specify `FORMAT JSON` after an input expression to declare that the value that results from it represents JSON data, and will therefore not be quoted in the output

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - If you specify this clause, then the function returns the JSON null value. This is the default.
- `ABSENT ON NULL` - If you specify this clause, then the function omits the property key-value pair from the JSON object.

JSON_returning_clause

The character string returned by this function is of data type `VARCHAR2`. This clause allows you to specify the size of the `VARCHAR2` data type. Use `BYTE` to specify the size as a number of bytes or `CHAR` to specify the size as a number of characters. The default is BYTE. If you omit this clause, or if you specify this clause but omit the *size* value, then JSON_OBJECT returns a character string of type VARCHAR2(4000).

Refer to VARCHAR2 Data Type for more information. Note that when specifying the VARCHAR2 data type elsewhere in SQL, you are required to specify a size. However, in the *JSON_returning_clause* you can omit the size.

Example

The following example returns JSON objects that each contain two property key-value pairs:

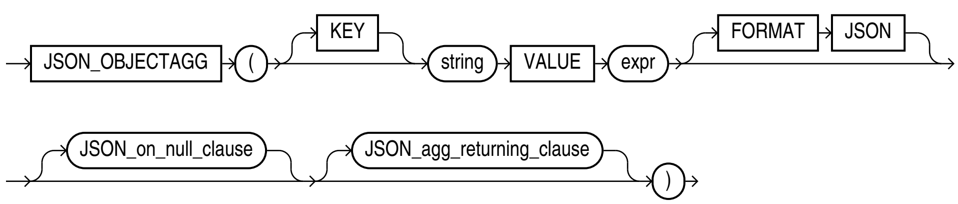
```
SELECT JSON_OBJECT (
    KEY 'deptno' IS d.department_id,
    KEY 'deptname' IS d.department_name
    ) "Department Objects"
  FROM departments d
  ORDER BY d.department_id;

Department Objects
----------------------------------------
{"deptno":10,"deptname":"Administration"}
{"deptno":20,"deptname":"Marketing"}
{"deptno":30,"deptname":"Purchasing"}
{"deptno":40,"deptname":"Human Resources"}
{"deptno":50,"deptname":"Shipping"}
. . .
```

See Also:

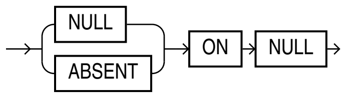
Generation of JSON Data With SQL/JSON Functions

JSON_OBJECTAGG

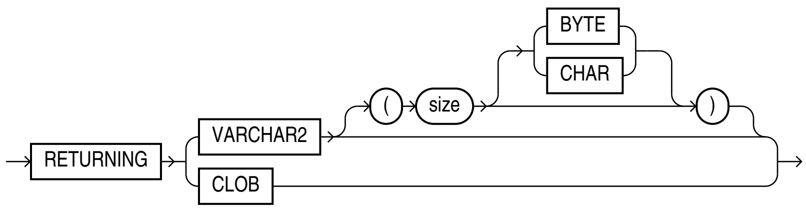
Syntax



***JSON_on_null_clause*::=**



***JSON_agg_returning_clause*::=**



Purpose

The SQL/JSON function `JSON_OBJECTAGG` is an aggregate function. It takes as its input a property key-value pair. Typically, the property key, the property value, or both are columns of SQL expressions. This function constructs an object member for each key-value pair and returns a single JSON object that contains those object members.

[KEY] *string* VALUE *expr*

Use this clause to specify property key-value pairs.

- `KEY` is optional and is provided for semantic clarity.
- Use *`string`* to specify the property key name as a case-sensitive text literal.
- Use *`expr`* to specify the property value. For *`expr`*, you can specify any expression that evaluates to a SQL numeric literal or text literal. If *`expr`* evaluates to a numeric literal, then the resulting property value is a JSON number value; otherwise, the resulting property value is a case-sensitive JSON string value enclosed in double quotation marks.

FORMAT JSON

This clause is optional and is provided for semantic clarity.

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - If you specify this clause, then the function returns the JSON null value. This is the default.
- `ABSENT ON NULL` - If you specify this clause, then the function omits the property key-value pair from the JSON object.

JSON_agg_returning_clause

Use this clause to specify the data type of the character string returned by this function. You can specify the following data types:

- `VARCHAR2[(`*`size`*` [BYTE,CHAR])]`

  When specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size.

- `CLOB`

If you omit this clause, or if you specify `VARCHAR2` but omit the *`size`* value, then `JSON_OBJECTAGG` returns a character string of type `VARCHAR2(4000)`.

Refer to "Data Types " for more information on the preceding data types.

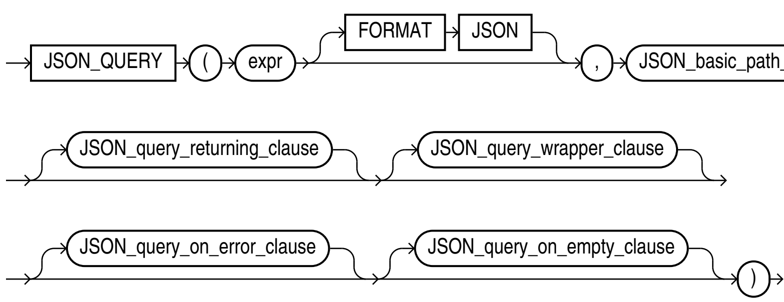
Examples

The following example constructs a JSON object whose members contain department names and department numbers:

```
SELECT JSON_OBJECTAGG(KEY department_name VALUE department_id) "Department Numbers"
  FROM departments
  WHERE department_id <= 30;

Department Numbers
----------------------------------------------------
{"Administration":10,"Marketing":20,"Purchasing":30}
```

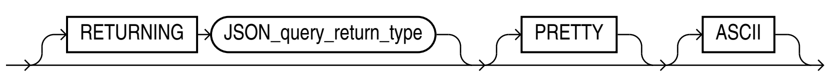
JSON_QUERY

Syntax

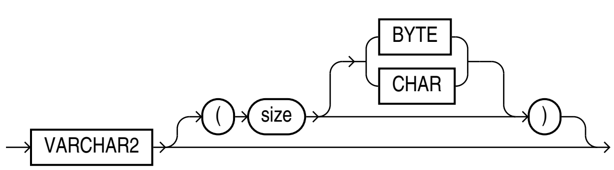


(*`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

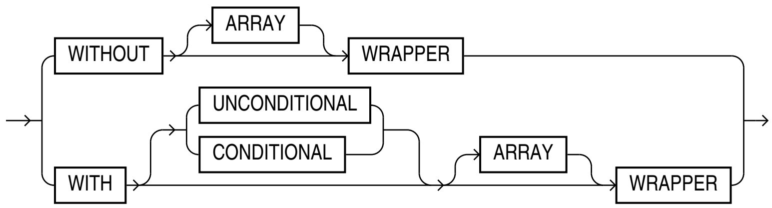
***JSON_query_returning_clause*::=**



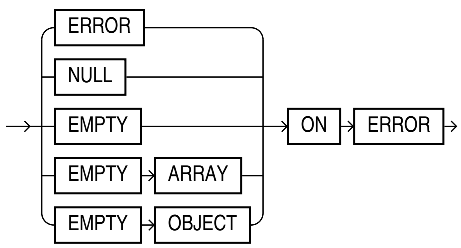
***JSON_query_return_type*::=**



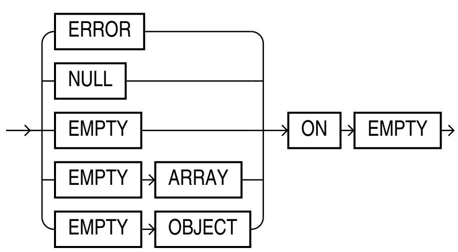
***JSON_query_wrapper_clause*::=**



***JSON_query_on_error_clause*::=**



***JSON_query_on_empty_clause*::=**



Purpose

The SQL/JSON function `JSON_QUERY` finds one or more specified JSON values in JSON data and returns the values in a character string.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character value returned by `JSON_QUERY`

expr

Use this clause to specify the JSON data to be evaluated. For `expr`, specify an expression that evaluates to a text literal. If `expr` is a column, then the column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. If `expr` is null, then the function returns null.

If `expr` is not a text literal of well-formed JSON data using strict or lax syntax, then the function returns null by default. You can use the `JSON_query_on_error_clause` to override this default behavior. Refer to *JSON_query_on_error_clause*.

FORMAT JSON

You must specify `FORMAT JSON` if `expr` is a column of data type `BLOB`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The function uses the path expression to evaluate `expr` and find one or more JSON values that match, or satisfy, the path expression. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of `JSON_basic_path_expression`.

JSON_query_returning_clause

Use this clause to specify the data type and format of the character string returned by this function.

RETURNING

Use the `RETURNING` clause to specify the data type of the character string. If you omit this clause, then `JSON_QUERY` returns a character string of type `VARCHAR2(4000)`.

You can use the `JSON_return_type_clause` to specify the following data type:

- `VARCHAR2[(size [BYTE,CHAR])]`

  When specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size. In this case, `JSON_QUERY` returns a character string of type `VARCHAR2(4000)`.

  Refer to "VARCHAR2 Data Type " for more information.

If the data type is not large enough to hold the return character string, then this function returns null by default. You can use the `JSON_query_on_error_clause` to override this default behavior. Refer to the *JSON_query_on_error_clause*.

PRETTY

Specify `PRETTY` to pretty-print the return character string by inserting newline characters and indenting.

ASCII

Specify `ASCII` to automatically escape any non-ASCII Unicode characters in the return character string, using standard ASCII Unicode escape sequences.

JSON_query_wrapper_clause

Use this clause to control whether this function wraps the values matched by the path expression in an array wrapper—that is, encloses the sequence of values in square brackets (`[]`).

- Specify `WITHOUT WRAPPER` to omit the array wrapper. You can specify this clause only if the path expression matches a single JSON object or JSON array. This is the default.
- Specify `WITH WRAPPER` to include the array wrapper. You must specify this clause if the path expression matches a single scalar value (a value that is not a JSON object or JSON array) or multiple values of any type.
- Specifying the `WITH UNCONDITIONAL WRAPPER` clause is equivalent to specifying the `WITH WRAPPER` clause. The `UNCONDITIONAL` keyword is provided for semantic clarity.
- Specify `WITH CONDITIONAL WRAPPER` to include the array wrapper only if the path expression matches a single scalar value or multiple values of any type. If the path expression matches a single JSON object or JSON array, then the array wrapper is omitted.

The `ARRAY` keyword is optional and is provided for semantic clarity.

If the function returns a single scalar value, or multiple values of any type, and you do not specify `WITH [UNCONDITIONAL | CONDITIONAL] WRAPPER`, then the function returns null by default. You can use the *`JSON_query_on_error_clause`* to override this default behavior. Refer to the *JSON_query_on_error_clause*.

JSON_query_on_error_clause

Use this clause to specify the value returned by this function when the following errors occur:

- *`expr`* is not well-formed JSON data using strict or lax JSON syntax
- No match is found when the JSON data is evaluated using the SQL/JSON path expression. You can override the behavior for this type of error by specifying the *`JSON_query_on_empty_clause`*.
- The return value data type is not large enough to hold the return character string
- The function matches a single scalar value or, multiple values of any type, and the `WITH [UNCONDITIONAL | CONDITIONAL] WRAPPER` clause is not specified

You can specify the following clauses:

- `NULL ON ERROR` - Returns null when an error occurs. This is the default.
- `ERROR ON ERROR` - Returns the appropriate Oracle error when an error occurs.
- `EMPTY ON ERROR` - Specifying this clause is equivalent to specifying `EMPTY ARRAY ON ERROR`.
- `EMPTY ARRAY ON ERROR` - Returns an empty JSON array (`[]`) when an error occurs.
- `EMPTY OBJECT ON ERROR` - Returns an empty JSON object (`{}`) when an error occurs.

JSON_query_on_empty_clause

Use this clause to specify the value returned by this function if no match is found when the JSON data is evaluated using the SQL/JSON path expression. This clause allows you to specify a different outcome for this type of error than the outcome specified with the *JSON_query_on_error_clause*.

You can specify the following clauses:

- `NULL ON EMPTY` - Returns null when no match is found.
- `ERROR ON EMPTY` - Returns the appropriate Oracle error when no match is found.
- `EMPTY ON EMPTY` - Specifying this clause is equivalent to specifying `EMPTY ARRAY ON EMPTY`.
- `EMPTY ARRAY ON EMPTY` - Returns an empty JSON array (`[]`) when no match is found.
- `EMPTY OBJECT ON EMPTY` - Returns an empty JSON object (`{}`) when no match is found.

If you omit this clause, then the *JSON_query_on_error_clause* determines the value returned when no match is found.

Examples

The following query returns the context item, or the specified string of JSON data. The path expression matches a single JSON object, which does not require an array wrapper. Note that the JSON data is converted to strict JSON syntax in the returned value—that is, the object property names are enclosed in double quotation marks.

```
SELECT JSON_QUERY('{a:100, b:200, c:300}', '$') AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
{"a":100,"b":200,"c":300}
```

The following query returns the value of the member with property name `a`. The path expression matches a scalar value, which must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('{a:100, b:200, c:300}', '$.a' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[100]
```

The following query returns the values of all object members. The path expression matches multiple values, which together must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('{a:100, b:200, c:300}', '$.*' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[100,200,300]
```

The following query returns the context item, or the specified string of JSON data. The path expression matches a single JSON array, which does not require an array wrapper.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$') AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[0,1,2,3,4]
```

The following query is similar to the previous query, except the `WITH WRAPPER` clause is specified. Therefore, the JSON array is wrapped in an array wrapper.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[[0,1,2,3,4]]
```

The following query returns all elements in a JSON array. The path expression matches multiple values, which together must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$[*]' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[0,1,2,3,4]
```

The following query returns the elements at indexes 0, 3 through 5, and 7 in a JSON array. The path expression matches multiple values, which together must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('[0,1,2,3,4,5,6,7,8]', '$[0, 3 TO 5, 7]' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[0,3,4,5,7]
```

The following query returns the fourth element in a JSON array. The path expression matches a scalar value, which must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$[3]' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[3]
```

The following query returns the first element in a JSON array. The `WITH CONDITIONAL WRAPPER` clause is specified and the path expression matches a single JSON object. Therefore, the value returned is not wrapped in an array. Note that the JSON data is converted to strict JSON syntax in the returned value—that is, the object property name is enclosed in double quotation marks.

```
SELECT JSON_QUERY('[{a:100},{b:200},{c:300}]', '$[0]'
       WITH CONDITIONAL WRAPPER) AS value
  FROM DUAL;
```

```
VALUE
--------------------------------------------------------------------------------
{"a":100}
```

The following query returns all elements in a JSON array. The `WITH CONDITIONAL WRAPPER` clause is specified and the path expression matches multiple JSON objects. Therefore, the value returned is wrapped in an array.

```
SELECT JSON_QUERY('[{"a":100},{"b":200},{"c":300}]', '$[*]'
       WITH CONDITIONAL WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[{"a":100},{"b":200},{"c":300}]
```

The following query is similar to the previous query, except that the value returned is of data type `VARCHAR2(100)`.

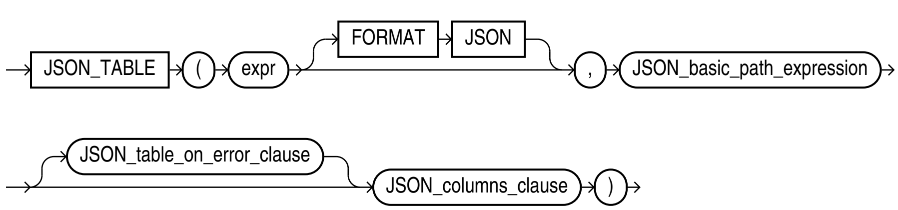
```
SELECT JSON_QUERY('[{"a":100},{"b":200},{"c":300}]', '$[*]'
       RETURNING VARCHAR2(100) WITH CONDITIONAL WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[{"a":100},{"b":200},{"c":300}]
```

The following query returns the fourth element in a JSON array. However, the supplied JSON array does not contain a fourth element, which results in an error. The `EMPTY ON ERROR` clause is specified. Therefore, the query returns an empty JSON array.

```
SELECT JSON_QUERY('[{"a":100},{"b":200},{"c":300}]', '$[3]'
       EMPTY ON ERROR) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[]
```

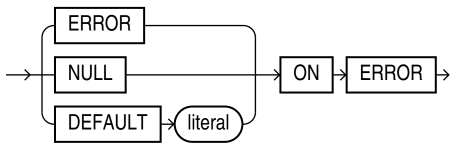
JSON_TABLE

Syntax

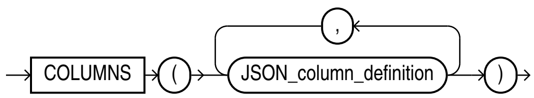


(*JSON_basic_path_expression*: See *Oracle Database JSON Developer's Guide*, *JSON_table_on_error_clause*::=, *JSON_columns_clause*::=)

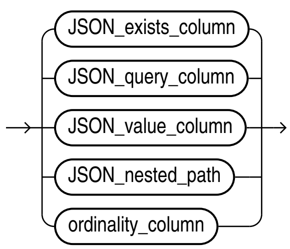
***JSON_table_on_error_clause*::=**



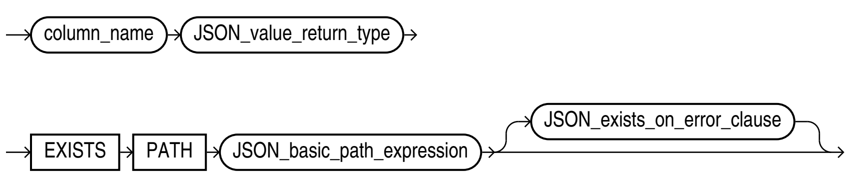
***JSON_columns_clause*::=**



***JSON_column_definition*::=**

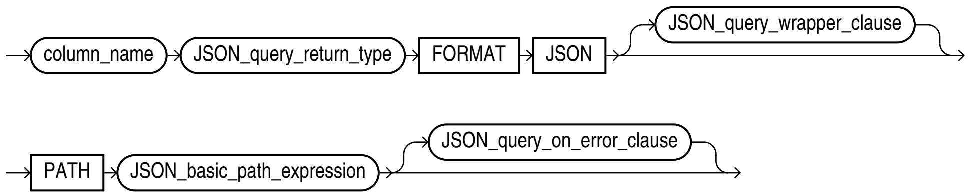


***JSON_exists_column*::=**



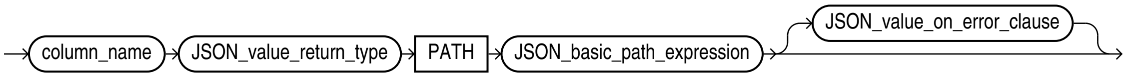
(*JSON_value_return_type*::=—part of `JSON_VALUE`, *`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*, *JSON_exists_on_error_clause*::=—part of `JSON_EXISTS`)

***JSON_query_column*::=**



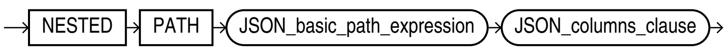
(*JSON_query_return_type*::=, *JSON_query_wrapper_clause*::=, and *JSON_query_on_error_clause*::=—part of `JSON_QUERY`, *`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

***JSON_value_column*::=**



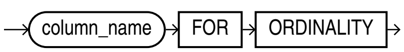
(*JSON_value_return_type*::= and *JSON_value_on_error_clause*::=—part of `JSON_VALUE`, *`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

***JSON_nested_path*::=**



(*`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*, *JSON_columns_clause*::=)

***ordinality_column*::=**



Purpose

The SQL/JSON function `JSON_TABLE` creates a relational view of JSON data. It maps the result of a JSON data evaluation into relational rows and columns. You can query the result returned by the function as a virtual relational table using SQL. The main purpose of `JSON_TABLE` is to create a row of relational data for each object inside a JSON array and output JSON values from within that object as individual SQL column values.

You must specify `JSON_TABLE` only in the `FROM` clause of a `SELECT` statement. The function first applies a path expression, called a **SQL/JSON row path expression**, to the supplied JSON data. The JSON value that matches the row path expression is called a **row source** in that it generates a row of relational data. The `COLUMNS` clause evaluates the row source, finds specific JSON values within the row source, and returns those JSON values as SQL values in individual columns of a row of relational data.

The `COLUMNS` clause enables you to search for JSON values in different ways by using the following clauses:

- *`JSON_exists_column`* - Evaluates JSON data in the same manner as the `JSON_EXISTS` condition, that is, determines if a specified JSON value exists, and returns either a `VARCHAR2` column of values '`true`' or '`false`', or a `NUMBER` column of values 1 or 0.
- *`JSON_query_column`* - Evaluates JSON data in the same manner as the `JSON_QUERY` function, that is, finds one or more specified JSON values, and returns a column of character strings that contain those JSON values.

- *JSON_value_column* - Evaluates JSON data in the same manner as the JSON_VALUE function, that is, finds a specified scalar JSON value, and returns a column of those JSON values as SQL values.
- *JSON_nested_path* - Allows you to flatten JSON values in a nested JSON object or JSON array into individual columns in a single row along with JSON values from the parent object or array. You can use this clause recursively to project data from multiple layers of nested objects or arrays into a single row.
- *ordinality_column* - Returns a column of generated row numbers.

The column definition clauses allow you to specify a name for each column of data that they return. You can reference these column names elsewhere in the SELECT statement, such as in the SELECT list and the WHERE clause.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to each character data type column in the table generated by JSON_TABLE

expr

Use this clause to specify the JSON data to be evaluated. For *expr*, specify an expression that evaluates to a text literal. If *expr* is a column, then the column must be of data type VARCHAR2, CLOB, or BLOB. If *expr* is null, then the function returns null.

If *expr* is not a text literal of well-formed JSON data using strict or lax syntax, then the function returns null by default. You can use the *JSON_table_on_error_clause* to override this default behavior. Refer to *JSON_table_on_error_clause*.

FORMAT JSON

You must specify FORMAT JSON if *expr* is a column of data type BLOB.

JSON_basic_path_expression

Use this clause to specify the SQL/JSON row path expression. The function uses the row path expression to evaluate *expr* and find the a JSON value, called the row source, that matches, or satisfy, the path expression. This row source is then evaluated by the COLUMNS clause. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of *JSON_basic_path_expression*.

JSON_table_on_error_clause

Use this clause to specify the value returned by this function when the following errors occur:

- *expr* is not well-formed JSON data using strict or lax JSON syntax
- No match is found when the JSON data is evaluated using the row path expression

You can specify the following clauses:

- `NULL ON ERROR` - Returns null when an error occurs. This is the default.
- `ERROR ON ERROR` - Returns the appropriate Oracle error when an error occurs.
- `DEFAULT` *`literal`* `ON ERROR` - Returns *`literal`* when an error occurs. If the data type of the value returned by this function is `VARCHAR2`, then you must specify a text literal. If the data type is `NUMBER`, then you must specify a numeric literal.

JSON_columns_clause

Use the `COLUMNS` clause to define the columns in the virtual relational table returned by the `JSON_TABLE` function.

JSON_exists_column

This clause evaluates JSON data in the same manner as the `JSON_EXISTS` condition, that is, it determines if a specified JSON value exists. It returns either a `VARCHAR2` column of values '`true`' or '`false`', or a `NUMBER` column of values 1 or 0. A value of '`true`' or 1 indicates that the JSON value exists and a value of '`false`' or 0 indicates that the JSON value does not exist.

You can use the *`JSON_value_return_type`* clause to control the data type of the returned column. If you omit this clause, then the data type is `VARCHAR2(4000)`. Use *`column_name`* to specify the name of the returned column. The rest of the clauses of *`JSON_exists_column`* have the same semantics here as they have for the `JSON_EXISTS` condition. For full information on these clauses, refer to "JSON_EXISTS Condition". Also see "Using *JSON_exists_column*: Examples" for an example.

JSON_query_column

This clause evaluates JSON data in the same manner as the `JSON_QUERY` function, that is, it finds one or more specified JSON values, and returns a column of character strings that contain those JSON values.

Use *`column_name`* to specify the name of the returned column. The rest of the clauses of *`JSON_query_column`* have the same semantics here as they have for the `JSON_QUERY` function. For full information on these clauses, refer to JSON_QUERY. Also see "Using *JSON_query_column*: Example" for an example.

JSON_value_column

This clause evaluates JSON data in the same manner as the `JSON_VALUE` function, that is, it finds a specified scalar JSON value, and returns a column of those JSON values as SQL values.

Use *`column_name`* to specify the name of the returned column. The rest of the clauses of *`JSON_value_column`* have the same semantics here as they have for the `JSON_VALUE` function. For full information on these clauses, refer to JSON_VALUE. Also see "Using *JSON_value_column*: Example" for an example.

JSON_nested_path

Use this clause to flatten JSON values in a nested JSON object or JSON array into individual columns in a single row along with JSON values from the parent object or array. You can use this clause recursively to project data from multiple layers of nested objects or arrays into a single row.

Specify the *JSON_basic_path_expression* clause to match the nested object or array. This path expression is relative to the SQL/JSON row path expression specified in the JSON_TABLE function.

Use the COLUMNS clause to define the columns of the nested object or array to be returned. This clause is recursive—you can specify the *JSON_nested_path* clause within another *JSON_nested_path* clause. Also see "Using *JSON_nested_path*: Examples" for an example.

ordinality_column

This clause returns a column of generated row numbers of data type NUMBER. You can specify at most one *ordinality_column*. Also see "Using *JSON_value_column*: Example" for an example of using the *ordinality_column* clause.

Examples

Creating a Table That Contains a JSON Document: Example

This example shows how to create and populate table j_purchaseorder, which is used in the rest of the JSON_TABLE examples in this section.

The following statement creates table j_purchaseorder. Column po_document is for storing JSON data and, therefore, has an IS JSON check constraint to ensure that only well-formed JSON is stored in the column.

```
CREATE TABLE j_purchaseorder
  (id RAW (16) NOT NULL,
   date_loaded TIMESTAMP(6) WITH TIME ZONE,
   po_document CLOB CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

The following statement inserts one row, or one JSON document, into table j_purchaseorder:

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    SYSTIMESTAMP,
    '{"PONumber"              : 1600,
      "Reference"             : "ABULL-20140421",
       "Requestor"            : "Alexis Bull",
       "User"                 : "ABULL",
       "CostCenter"           : "A50",
       "ShippingInstructions" : {"name"   : "Alexis Bull",
                                 "Address": {"street"   : "200 Sporting Green",
                                              "city"    : "South San Francisco",
                                              "state"   : "CA",
                                              "zipCode" : 99236,
                                              "country" : "United States of America"},
                                 "Phone" : [{"type" : "Office", "number" : "909-555-7307"},
                                            {"type" : "Mobile", "number" : "415-555-1234"}]},
       "Special Instructions" : null,
       "AllowPartialShipment" : true,
       "LineItems" : [{"ItemNumber" : 1,
                       "Part" : {"Description" : "One Magic Christmas",
                                 "UnitPrice"   : 19.95,
                                 "UPCCode"     : 13131092899},
                       "Quantity" : 9.0},
                      {"ItemNumber" : 2,
```

```
       "Part" : {"Description" : "Lethal Weapon",
                 "UnitPrice"   : 19.95,
                 "UPCCode"     : 85391628927},
       "Quantity" : 5.0}]}');
```

Using *JSON_query_column*: Example

The statement in this example queries JSON data for a specific JSON property using the *`JSON_query_column`* clause, and returns the property value in a column.

The statement first applies a SQL/JSON row path expression to column `po_document`, which results in a match to the `ShippingInstructions` property. The `COLUMNS` clause then uses the *`JSON_query_column`* clause to return the `Phone` property value in a `VARCHAR2(100)` column.

```
SELECT jt.phones
FROM j_purchaseorder,
JSON_TABLE(po_document, '$.ShippingInstructions'
COLUMNS
  (phones VARCHAR2(100) FORMAT JSON PATH '$.Phone')) AS jt;

PHONES
-------------------------------------------------------------------------------------
[{"type":"Office","number":"909-555-7307"},{"type":"Mobile","number":"415-555-1234"}]
```

Using *JSON_value_column*: Example

The statement in this example refines the statement in the previous example by querying JSON data for specific JSON values using the *`JSON_value_column`* clause, and returns the JSON values as SQL values in relational rows and columns.

The statement first applies a SQL/JSON row path expression to column `po_document`, which results in a match to the elements in the JSON array `Phone`. These elements are JSON objects that contain two members named `type` and `number`. The statement uses the `COLUMNS` clause to return the `type` value for each object in a `VARCHAR2(10)` column called `phone_type`, and the `number` value for each object in a `VARCHAR2(20)` column called `phone_num`. The statement also returns an ordinal column named `row_number`.

```
SELECT jt.*
FROM j_purchaseorder,
JSON_TABLE(po_document, '$.ShippingInstructions.Phone[*]'
COLUMNS (row_number FOR ORDINALITY,
         phone_type VARCHAR2(10) PATH '$.type',
         phone_num VARCHAR2(20) PATH '$.number'))
AS jt;

ROW_NUMBER PHONE_TYPE PHONE_NUM
---------- ---------- --------------------
         1 Office     909-555-7307
         2 Mobile     415-555-1234
```

Using *JSON_exists_column*: Examples

The statements in this example test whether a JSON value exists in JSON data using the *`JSON_exists_column`* clause. The first example returns the result of the test as a '`true`' or '`false`' value in a column. The second example uses the result of the test in the `WHERE` clause.

The following statement first applies a SQL/JSON row path expression to column `po_document`, which results in a match to the entire context item, or JSON document. It then uses the `COLUMNS` clause to return the requestor's name and a string value of '`true`' or '`false`' indicating whether the JSON data for that requestor contains a zip code. The `COLUMNS` clause first uses the *`JSON_value_column`* clause to return the `Requestor` value in a `VARCHAR2(32)` column called `requestor`. It then uses the *`JSON_exists_column`* clause to determine if the `zipCode` object exists and returns the result in a `VARCHAR2(5)` column called `has_zip`.

```
SELECT requestor, has_zip
FROM j_purchaseorder,
JSON_TABLE(po_document, '$'
COLUMNS
  (requestor VARCHAR2(32) PATH '$.Requestor',
   has_zip VARCHAR2(5) EXISTS PATH '$.ShippingInstructions.Address.zipCode'));

REQUESTOR                        HAS_ZIP
-------------------------------- -------
Alexis Bull                      true
```

The following statement is similar to the previous statement, except that it uses the value of `has_zip` in the `WHERE` clause to determine whether to return the `Requestor` value:

```
SELECT requestor
FROM j_purchaseorder,
JSON_TABLE(po_document, '$'
COLUMNS
  (requestor VARCHAR2(32) PATH '$.Requestor',
   has_zip VARCHAR2(5) EXISTS PATH '$.ShippingInstructions.Address.zipCode'))
WHERE (has_zip = 'true');

REQUESTOR
--------------------------------
Alexis Bull
```

Using *JSON_nested_path*: Examples

The following two simple statements demonstrate the functionality of the *`JSON_nested_path`* clause. They operate on a simple JSON array that contains three elements. The first two elements are numbers. The third element is a nested JSON array that contains two string value elements.

The following statement does not use the *`JSON_nested_path`* clause. It returns the three elements in the array in a single row. The nested array is returned in its entirety.

```
SELECT *
FROM JSON_TABLE('[1,2,["a","b"]]', '$'
COLUMNS (outer_value_0 NUMBER PATH '$[0]',
         outer_value_1 NUMBER PATH '$[1]',
         outer_value_2 VARCHAR2(20) FORMAT JSON PATH '$[2]'));

OUTER_VALUE_0 OUTER_VALUE_1 OUTER_VALUE_2
------------- ------------- --------------------
            1             2 ["a","b"]
```

The following statement is different from the previous statement because it uses the *`JSON_nested_path`* clause to return the individual elements of the nested array in individual columns in a single row along with the parent array elements.

```
SELECT *
FROM JSON_TABLE('[1,2,["a","b"]]', '$'
COLUMNS (outer_value_0 NUMBER PATH '$[0]',
```

```
          outer_value_1 NUMBER PATH '$[1]',
          NESTED PATH '$[2]'
          COLUMNS (nested_value_0 VARCHAR2(1) PATH '$[0]',
                   nested_value_1 VARCHAR2(1) PATH '$[1]')));

OUTER_VALUE_0 OUTER_VALUE_1 NESTED_VALUE_0 NESTED_VALUE_1
------------- ------------- -------------- --------------
            1             2 a              b
```

The previous example shows how to use *`JSON_nested_path`* with a nested JSON array. The following example shows how to use the *`JSON_nested_path`* clause with a nested JSON object by returning the individual elements of the nested object in individual columns in a single row along with the parent object elements.

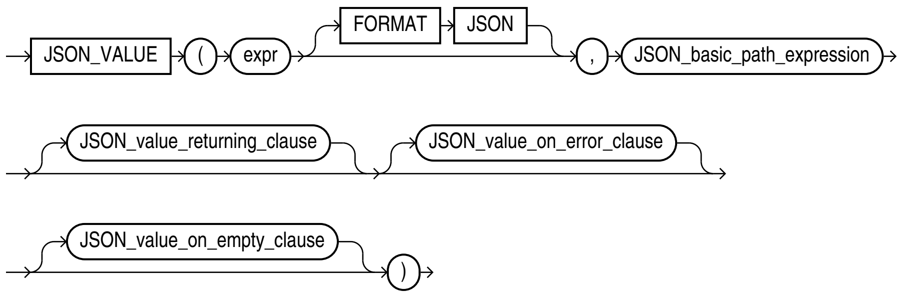
```
SELECT *
FROM JSON_TABLE('{a:100, b:200, c:{d:300, e:400}}', '$'
COLUMNS (outer_value_0 NUMBER PATH '$.a',
         outer_value_1 NUMBER PATH '$.b',
         NESTED PATH '$.c'
         COLUMNS (nested_value_0 NUMBER PATH '$.d',
                  nested_value_1 NUMBER PATH '$.e')));

OUTER_VALUE_0 OUTER_VALUE_1 NESTED_VALUE_0 NESTED_VALUE_1
------------- ------------- -------------- --------------
          100           200            300            400
```

The following statement uses the *`JSON_nested_path`* clause when querying the `j_purchaseorder` table. It first applies a row path expression to column `po_document`, which results in a match to the entire context item, or JSON document. It then uses the `COLUMNS` clause to return the `Requestor` value in a `VARCHAR2(32)` column called `requestor`. It then uses the *`JSON_nested_path`* clause to return the property values of the individual objects in each member of the nested `Phone` array. Note that a row is generated for each member of the nested array, and each row contains the corresponding `Requestor` value.

```
SELECT jt.*
FROM j_purchaseorder,
JSON_TABLE(po_document, '$'
COLUMNS
  (requestor VARCHAR2(32) PATH '$.Requestor',
   NESTED PATH '$.ShippingInstructions.Phone[*]'
     COLUMNS (phone_type VARCHAR2(32) PATH '$.type',
              phone_num VARCHAR2(20) PATH '$.number')))
AS jt;


REQUESTOR            PHONE_TYPE           PHONE_NUM
-------------------- -------------------- ---------------
Alexis Bull          Office               909-555-7307
Alexis Bull          Mobile               415-555-1234
```

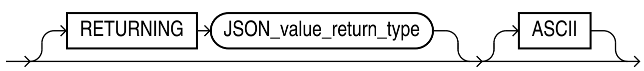
JSON_VALUE

Syntax

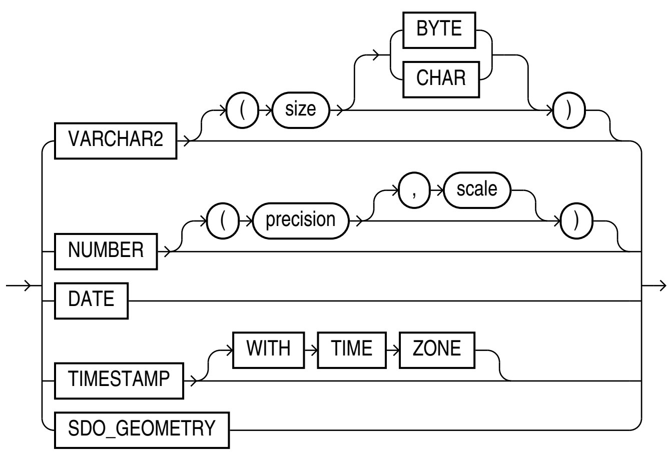


(*JSON_basic_path_expression*: See *Oracle Database JSON Developer's Guide*)

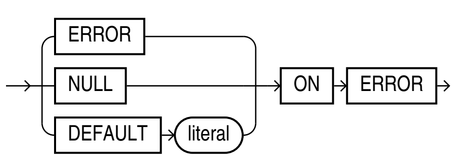
***JSON_value_returning_clause*::=**



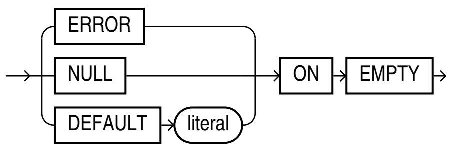
***JSON_value_return_type*::=**



***JSON_value_on_error_clause*::=**



***JSON_value_on_empty_clause*::=**



Purpose

The SQL/JSON function `JSON_VALUE` finds a specified scalar JSON value in JSON data and returns it as a SQL value.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the value returned by this function when it is a character value

expr

Use this clause to specify the JSON data to be evaluated. For *expr*, specify an expression that evaluates to a text literal. If *expr* is a column, then the column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. If *expr* is null, then the function returns null.

If *expr* is not a text literal of well-formed JSON data using strict or lax syntax, then the function returns null by default. You can use the *JSON_value_on_error_clause* to override this default behavior. Refer to the *JSON_value_on_error_clause*.

FORMAT JSON

You must specify `FORMAT JSON` if *expr* is a column of data type `BLOB`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The function uses the path expression to evaluate *expr* and find a scalar JSON value that matches, or satisfies, the path expression. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of *JSON_basic_path_expression*.

JSON_value_returning_clause

Use this clause to specify the data type and format of the value returned by this function.

RETURNING

Use the `RETURNING` clause to specify the data type of the return value. If you omit this clause, then `JSON_VALUE` returns a value of type `VARCHAR2(4000)`.

You can use *JSON_value_return_type* to specify the following data types:

- `VARCHAR2[(`*size* `[BYTE,CHAR])]`

If you specify this data type, then the scalar value returned by this function can be a character or number value. A number value will be implicitly converted to a `VARCHAR2`. When specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size. In this case, `JSON_VALUE` returns a value of type `VARCHAR2(4000)`.

- `NUMBER[(`*`precision`* `[,` *`scale`*`])]`

  If you specify this data type, then the scalar value returned by this function must be a number value.

- `DATE`

  If you specify this data type, then the scalar value returned by this function must be a character value that can be implicitly converted to a `DATE` data type.

- `TIMESTAMP`

  If you specify this data type, then the scalar value returned by this function must be a character value that can be implicitly converted to a `TIMESTAMP` data type.

- `TIMESTAMP WITH TIME ZONE`

  If you specify this data type, then the scalar value returned by this function must be a character value that can be implicitly converted to a `TIMESTAMP WITH TIME ZONE` data type.

- `SDO_GEOMETRY`

  This data type is used for Oracle Spatial and Graph data. If you specify this data type, then *`expr`* must evaluate to a text literal containing GeoJSON data, which is a format for encoding geographic data in JSON. If you specify this data type, then the scalar value returned by this function must be an object of type `SDO_GEOMETRY`.

Refer to "Data Types " for more information on the preceding data types.

If the data type is not large enough to hold the return value, then this function returns null by default. You can use the *`JSON_value_on_error_clause`* to override this default behavior. Refer to the *JSON_value_on_error_clause*.

ASCII

Specify `ASCII` to automatically escape any non-ASCII Unicode characters in the return value, using standard ASCII Unicode escape sequences.

JSON_value_on_error_clause

Use this clause to specify the value returned by this function when the following errors occur:

- *`expr`* is not well-formed JSON data using strict or lax JSON syntax
- A nonscalar value is found when the JSON data is evaluated using the SQL/JSON path expression
- No match is found when the JSON data is evaluated using the SQL/JSON path expression. You can override the behavior for this type of error by specifying the *`JSON_value_on_empty_clause`*.
- The return value data type is not large enough to hold the return value

You can specify the following clauses:

- `NULL ON ERROR` - Returns null when an error occurs. This is the default.
- `ERROR ON ERROR` - Returns the appropriate Oracle error when an error occurs.
- `DEFAULT` *`literal`* `ON ERROR` - Returns *`literal`* when an error occurs. The data type of *`literal`* must match the data type of the value returned by this function.

JSON_value_on_empty_clause

Use this clause to specify the value returned by this function if no match is found when the JSON data is evaluated using the SQL/JSON path expression. This clause allows you to specify a different outcome for this type of error than the outcome specified with the *`JSON_value_on_error_clause`*.

You can specify the following clauses:

- `NULL ON EMPTY` - Returns null when no match is found.
- `ERROR ON EMPTY` - Returns the appropriate Oracle error when no match is found.
- `DEFAULT` *`literal`* `ON EMPTY` - Returns *`literal`* when no match is found. The data type of *`literal`* must match the data type of the value returned by this function.

If you omit this clause, then the *`JSON_value_on_error_clause`* determines the value returned when no match is found.

Examples

The following query returns the value of the member with property name `a`. Because the `RETURNING` clause is not specified, the value is returned as a `VARCHAR2(4000)` data type:

```
SELECT JSON_VALUE('{a:100}', '$.a') AS value
  FROM DUAL;

VALUE
-----
100
```

The following query returns the value of the member with property name `a`. Because the `RETURNING NUMBER` clause is specified, the value is returned as a `NUMBER` data type:

```
SELECT JSON_VALUE('{a:100}', '$.a' RETURNING NUMBER) AS value
  FROM DUAL;

     VALUE
----------
       100
```

The following query returns the value of the member with property name `b`, which is in the value of the member with property name `a`:

```
SELECT JSON_VALUE('{a:{b:100}}', '$.a.b') AS value
  FROM DUAL;

VALUE
-----
100
```

The following query returns the value of the member with property name `d` in any object:

```
SELECT JSON_VALUE('{a:{b:100}, c:{d:200}, e:{f:300}}', '$.*.d') AS value
  FROM DUAL;

VALUE
-----
200
```

The following query returns the value of the first element in an array:

```
SELECT JSON_VALUE('[0, 1, 2, 3]', '$[0]') AS value
  FROM DUAL;

VALUE
-----
0
```

The following query returns the value of the third element in an array. The array is the value of the member with property name `a`.

```
SELECT JSON_VALUE('{a:[5, 10, 15, 20]}', '$.a[2]') AS value
  FROM DUAL;

VALUE
-----
15
```

The following query returns the value of the member with property name `a` in the second object in an array:

```
SELECT JSON_VALUE('[{a:100}, {a:200}, {a:300}]', '$[1].a') AS value
  FROM DUAL;

VALUE
-----
200
```

The following query returns the value of the member with property name `c` in any object in an array:

```
SELECT JSON_VALUE('[{a:100}, {b:200}, {c:300}]', '$[*].c') AS value
  FROM DUAL;

VALUE
-----
300
```

The following query attempts to return the value of the member that has property name `lastname`. However, such a member does not exist in the specified JSON data, resulting in no match. Because the `ON ERROR` clause is not specified, the statement uses the default `NULL ON ERROR` and returns null.

```
SELECT JSON_VALUE('{firstname:"John"}', '$.lastname') AS "Last Name"
  FROM DUAL;

Last Name
---------

```

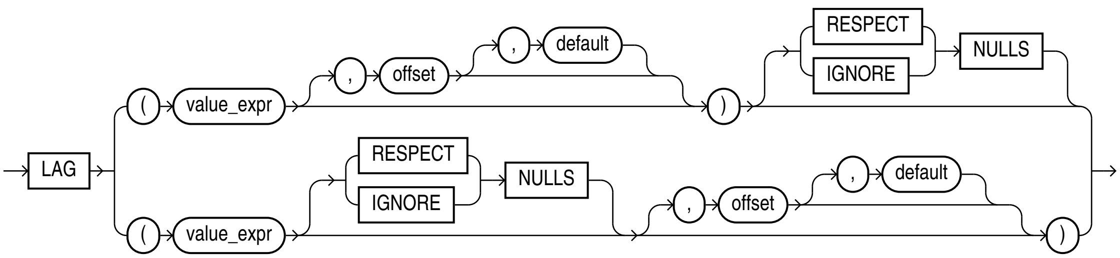
The following query results in an error because it attempts to return the value of the member with property name `lastname`, which does not exist in the specified JSON.

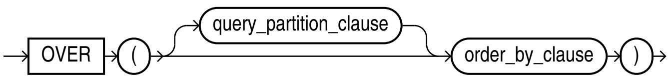
Because the ON ERROR clause is specified, the statement returns the specified text literal.

```
SELECT JSON_VALUE('{firstname:"John"}', '$.lastname'
                  DEFAULT 'No last name found' ON ERROR) AS "Last Name"
  FROM DUAL;

Last Name
---------
No last name found
```

LAG

Syntax





> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *value_expr*

Purpose

LAG is an analytic function. It provides access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position.

For the optional *offset* argument, specify an integer that is greater than zero. If you do not specify *offset*, then its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the window. If you do not specify *default*, then its default is null.

{RESPECT | IGNORE} NULLS determines whether null values of *value_expr* are included in or eliminated from the calculation. The default is RESPECT NULLS.

You cannot nest analytic functions by using LAG or any other analytic function for *value_expr*. However, you can use other built-in function expressions for *value_expr*.

See Also:

- "About SQL Expressions " for information on valid forms of *expr* and LEAD
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of LAG when it is a character value

Examples

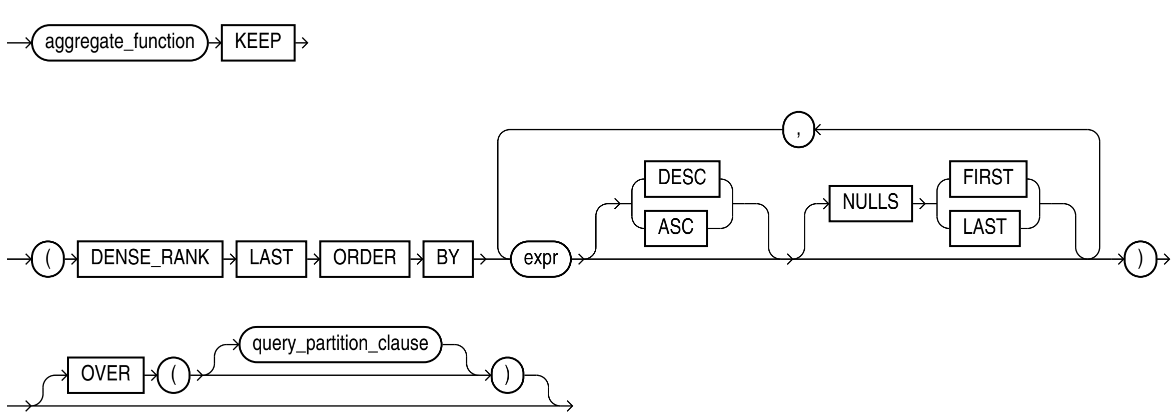
The following example provides, for each purchasing clerk in the employees table, the salary of the employee hired just before:

```
SELECT hire_date, last_name, salary,
       LAG(salary, 1, 0) OVER (ORDER BY hire_date) AS prev_sal
  FROM employees
  WHERE job_id = 'PU_CLERK'
  ORDER BY hire_date;

HIRE_DATE LAST_NAME                     SALARY   PREV_SAL
--------- ------------------------- ---------- ----------
18-MAY-03 Khoo                            3100          0
24-JUL-05 Tobias                          2800       3100
24-DEC-05 Baida                           2900       2800
15-NOV-06 Himuro                          2600       2900
10-AUG-07 Colmenares                      2500       2600
```

LAST

Syntax

***last*::=**



See Also:

"Analytic Functions " for information on syntax, semantics, and restrictions of the *query_partitioning_clause*

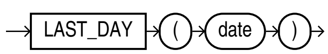
Purpose

FIRST and LAST are very similar functions. Both are aggregate and analytic functions that operate on a set of values from a set of rows that rank as the FIRST or LAST with respect to a given sorting specification. If only one row ranks as FIRST or LAST, then the aggregate operates on the set with only one element.

Refer to FIRST for complete information on this function and for examples of its use.

LAST_DAY

Syntax



Purpose

LAST_DAY returns the date of the last day of the month that contains *date*. The last day of the month is defined by the session parameter NLS_CALENDAR. The return type is always DATE, regardless of the data type of *date*.

Examples

The following statement determines how many days are left in the current month.

```
SELECT SYSDATE,
       LAST_DAY(SYSDATE) "Last",
       LAST_DAY(SYSDATE) - SYSDATE "Days Left"
  FROM DUAL;

SYSDATE   Last       Days Left
--------- --------- ----------
30-MAY-09 31-MAY-09          1
```

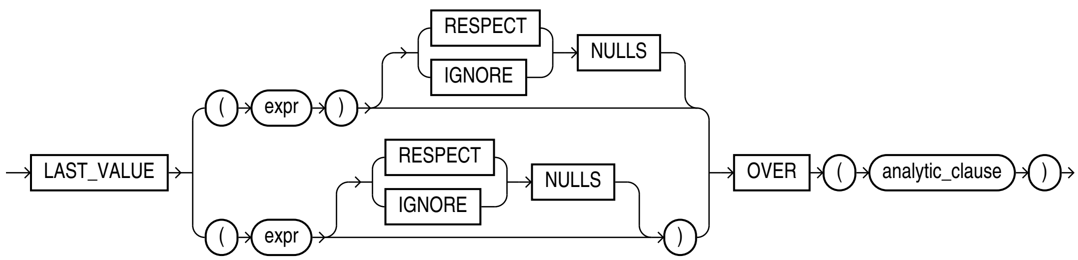
The following example adds 5 months to the hire date of each employee to give an evaluation date:

```
SELECT last_name, hire_date,
       TO_CHAR(ADD_MONTHS(LAST_DAY(hire_date), 5)) "Eval Date"
  FROM employees
  ORDER BY last_name, hire_date;

LAST_NAME                 HIRE_DATE Eval Date
------------------------- --------- ---------
Abel                      11-MAY-04 31-OCT-04
Ande                      24-MAR-08 31-AUG-08
Atkinson                  30-OCT-05 31-MAR-06
Austin                    25-JUN-05 30-NOV-05
```

```
Baer                      07-JUN-02 30-NOV-02
Baida                     24-DEC-05 31-MAY-06
Banda                     21-APR-08 30-SEP-08
Bates                     24-MAR-07 31-AUG-07
. . .
```

LAST_VALUE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

LAST_VALUE is an analytic function that is useful for data densification. It returns the last value in an ordered set of values.

> **Note:**
>
> The two forms of this syntax have the same behavior. The top branch is the ANSI format, which Oracle recommends for ANSI compatibility.

{RESPECT | IGNORE} NULLS determines whether null values of *expr* are included in or eliminated from the calculation. The default is RESPECT NULLS. If the last value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, then LAST_VALUE returns the last non-null value in the set, or NULL if all values are null. Refer to "Using Partitioned Outer Joins: Examples" for an example of data densification.

You cannot nest analytic functions by using LAST_VALUE or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*. Refer to "About SQL Expressions " for information on valid forms of *expr*.

If you omit the *windowing_clause* of the *analytic_clause*, it defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. This default sometimes returns an unexpected value, because the last value in the window is at the bottom of the window, which is not fixed. It keeps changing as the current row changes. For expected results, specify the *windowing_clause* as RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Alternatively, you can specify the *windowing_clause* as RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following example returns, for each row, the hire date of the employee earning the lowest salary:

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
               FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                           24000 17-JUN-03 13-JAN-01
        101 Kochhar                        17000 21-SEP-05 13-JAN-01
        102 De Haan                        17000 13-JAN-01 13-JAN-01
```

This example illustrates the nondeterministic nature of the LAST_VALUE function. Kochhar and De Haan have the same salary, so they are in adjacent rows. Kochhar appears first because the rows in the subquery are ordered by hire_date. However, if the rows are ordered by hire_date in descending order, as in the next example, then the function returns a different value:

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
               FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                           24000 17-JUN-03 21-SEP-05
        102 De Haan                        17000 13-JAN-01 21-SEP-05
        101 Kochhar                        17000 21-SEP-05 21-SEP-05
```

The following two examples show how to make the LAST_VALUE function deterministic by ordering on a unique key. By ordering within the function by both salary and the unique key employee_id, you can ensure the same result regardless of the ordering in the subquery.

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
```

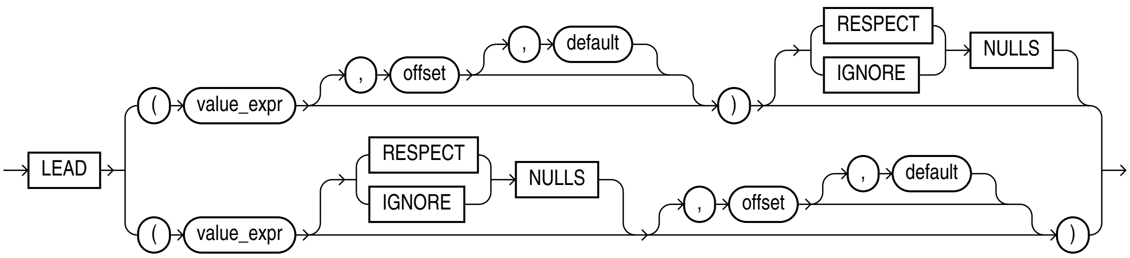
```
         OVER (ORDER BY salary DESC, employee_id ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                           24000 17-JUN-03 13-JAN-01
        101 Kochhar                        17000 21-SEP-05 13-JAN-01
        102 De Haan                        17000 13-JAN-01 13-JAN-01


SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC, employee_id ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                           24000 17-JUN-03 13-JAN-01
        101 Kochhar                        17000 21-SEP-05 13-JAN-01
        102 De Haan                        17000 13-JAN-01 13-JAN-01
```

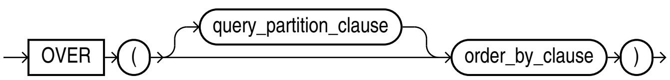
The following two examples show that the `LAST_VALUE` function is deterministic when you use a logical offset (`RANGE` instead of `ROWS`). When duplicates are found for the `ORDER BY` expression, the `LAST_VALUE` is the highest value of *`expr`*:

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC RANGE BETWEEN UNBOUNDED PRECEDING AND
               UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                           24000 17-JUN-03 21-SEP-05
        102 De Haan                        17000 13-JAN-01 21-SEP-05
        101 Kochhar                        17000 21-SEP-05 21-SEP-05


SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC RANGE BETWEEN UNBOUNDED PRECEDING AND
               UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                     SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                           24000 17-JUN-03 21-SEP-05
        102 De Haan                        17000 13-JAN-01 21-SEP-05
        101 Kochhar                        17000 21-SEP-05 21-SEP-05
```

LEAD

Syntax





> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *value_expr*

Purpose

LEAD is an analytic function. It provides access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, LEAD provides access to a row at a given physical offset beyond that position.

If you do not specify *offset*, then its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the table. If you do not specify *default*, then its default value is null.

{RESPECT | IGNORE} NULLS determines whether null values of *value_expr* are included in or eliminated from the calculation. The default is RESPECT NULLS.

You cannot nest analytic functions by using LEAD or any other analytic function for *value_expr*. However, you can use other built-in function expressions for *value_expr*.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *expr* and LAG
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of LEAD when it is a character value

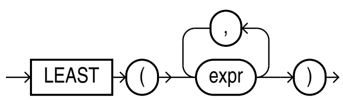
Examples

The following example provides, for each employee in Department 30 in the `employees` table, the hire date of the employee hired just after:

```
SELECT hire_date, last_name,
       LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "NextHired"
  FROM employees
  WHERE department_id = 30
  ORDER BY hire_date;

HIRE_DATE LAST_NAME                 Next Hired
--------- ------------------------- ----------
07-DEC-02 Raphaely                  18-MAY-03
18-MAY-03 Khoo                      24-JUL-05
24-JUL-05 Tobias                    24-DEC-05
24-DEC-05 Baida                     15-NOV-06
15-NOV-06 Himuro                    10-AUG-07
10-AUG-07 Colmenares
```

LEAST

Syntax



Purpose

`LEAST` returns the least of a list of one or more expressions. Oracle Database uses the first *`expr`* to determine the return type. If the first *`expr`* is numeric, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type before the comparison, and returns that data type. If the first *`expr`* is not numeric, then each *`expr`* after the first is implicitly converted to the data type of the first *`expr`* before the comparison.

Oracle Database compares each *`expr`* using nonpadded comparison semantics. The comparison is binary by default and is linguistic if the `NLS_COMP` parameter is set to `LINGUISTIC` and the `NLS_SORT` parameter has a setting other than `BINARY`. Character comparison is based on the numerical codes of the characters in the database character set and is performed on whole strings treated as one sequence of bytes, rather than character by character. If the value returned by this function is character data, then its data type is `VARCHAR2` if the first *`expr`* is a character data type and `NVARCHAR2` if the first *`expr`* is a national character data type.

See Also:

- "Data Type Comparison Rules " for more information on character comparison
- Table 2-8 for more information on implicit conversion and "Floating-Point Numbers " for information on binary-float comparison semantics
- "GREATEST ", which returns the greatest of a list of one or more expressions
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `LEAST` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following statement selects the string with the least value:

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "Least"
  FROM DUAL;

Least
------
HAROLD
```

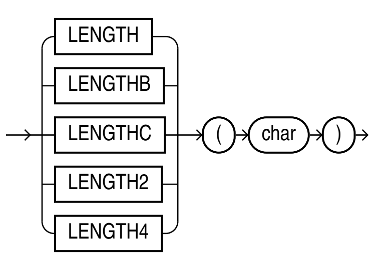
In the following statement, the first argument is numeric. Oracle Database determines that the argument with the highest numeric precedence is the third argument, converts the remaining arguments to the data type of the third argument, and returns the least value as that data type:

```
SELECT LEAST (1, '2.1', '.000832') "Least"
  FROM DUAL;

Least
-------
.000832
```

LENGTH

Syntax

***length*::=**



Purpose

The `LENGTH` functions return the length of *`char`*. `LENGTH` calculates length using characters as defined by the input character set. `LENGTHB` uses bytes instead of characters. `LENGTHC` uses Unicode complete characters. `LENGTH2` uses UCS2 code points. `LENGTH4` uses UCS4 code points.

`char` can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The exceptions are `LENGTHC`, `LENGTH2`, and `LENGTH4`, which do not allow *`char`* to be a `CLOB` or `NCLOB`. The return value is of data type `NUMBER`. If *`char`* has data type `CHAR`, then the length includes all trailing blanks. If *`char`* is null, then this function returns null.

Restriction on LENGTHB

The `LENGTHB` function is supported for single-byte LOBs only. It cannot be used with `CLOB` and `NCLOB` data in a multibyte character set.

Examples

The following example uses the `LENGTH` function using a single-byte database character set:

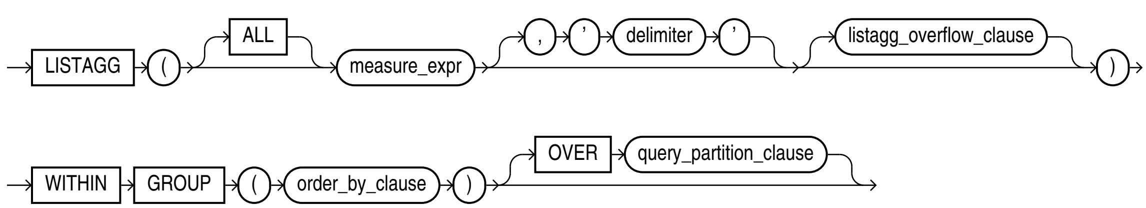
```
SELECT LENGTH('CANDIDE') "Length in characters"
  FROM DUAL;

Length in characters
--------------------
                   7
```

The next example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
  FROM DUAL;

Length in bytes
---------------
             14
```

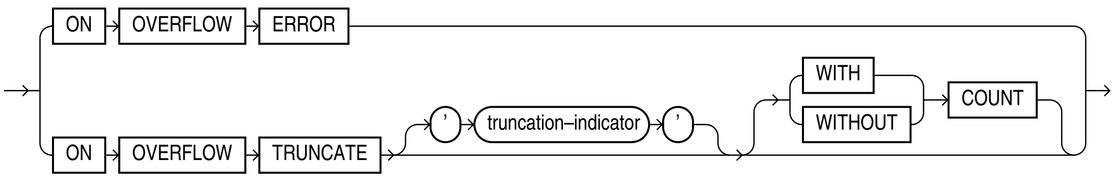
LISTAGG

Syntax



(*listagg_overflow_clause::=*, *order_by_clause::=*, *query_partition_clause::=*)

***listagg_overflow_clause*::=**



See Also:

"Analytic Functions " for information on syntax, semantics, and restrictions of the `ORDER BY` clause and `OVER` clause

Purpose

For a specified measure, `LISTAGG` orders data within each group specified in the `ORDER BY` clause and then concatenates the values of the measure column.

- As a single-set aggregate function, `LISTAGG` operates on all rows and returns a single output row.
- As a group-set aggregate, the function operates on and returns an output row for each group defined by the `GROUP BY` clause.
- As an analytic function, `LISTAGG` partitions the query result set into groups based on one or more expression in the *`query_partition_clause`*.

The arguments to the function are subject to the following rules:

- The `ALL` keyword is optional and is provided for semantic clarity.
- The *`measure_expr`* is the measure column and can be any expression. Null values in the measure column are ignored.
- The *`delimiter`* designates the string that is to separate the measure column values. This clause is optional and defaults to `NULL`.

  If `measure_expr` is of type `RAW`, then the delimiter must be of type `RAW`. You can achieve this by specifying the delimiter as a character string that can be implicitly converted to `RAW`, or by explicitly converting the delimiter to `RAW`, for example, using the `UTL_RAW.CAST_TO_RAW` function.
- The *`order_by_clause`* determines the order in which the concatenated values are returned. The function is deterministic only if the `ORDER BY` column list achieved unique ordering.

If the measure column is of type `RAW`, then the return data type is `RAW`. Otherwise, the return data type is `VARCHAR2`.

The maximum length of the return data type depends on the value of the `MAX_STRING_SIZE` initialization parameter. If `MAX_STRING_SIZE = EXTENDED`, then the maximum length is 32767 bytes for the `VARCHAR2` and `RAW` data types. If `MAX_STRING_SIZE = STANDARD`, then the maximum length is 4000 bytes for the `VARCHAR2` data type and 2000 bytes for the `RAW` data type. A final delimiter is not included when determining if the return value fits in the return data type.

> **See Also:**
>
> - Extended Data Types for more information on the `MAX_STRING_SIZE` initialization parameter
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `LISTAGG`

listagg_overflow_clause

This clause controls how the function behaves when the return value exceeds the maximum length of the return data type.

ON OVERFLOW ERROR If you specify this clause, then the function returns an ORA-01489 error. This is the default.

ON OVERFLOW TRUNCATE If you specify this clause, then the function returns a truncated list of measure values.

- The *`truncation_indicator`* designates the string that is to be appended to the truncated list of measure values. If you omit this clause, then the truncation indicator is an ellipsis (`...`).

  If *`measure_expr`* is of type `RAW`, then the truncation indicator must be of type `RAW`. You can achieve this by specifying the truncation indicator as a character string that can be implicitly converted to `RAW`, or by explicitly converting the truncation indicator to `RAW`, for example, using the `UTL_RAW.CAST_TO_RAW` function.
- If you specify `WITH COUNT`, then after the truncation indicator, the database appends the number of truncated values, enclosed in parentheses. In this case, the database truncates enough measure values to allow space in the return value for a final delimiter, the truncation indicator, and 24 characters for the number value enclosed in parentheses.
- If you specify `WITHOUT COUNT`, then the database omits the number of truncated values from the return value. In this case, the database truncates enough measure values to allow space in the return value for a final delimiter and the truncation indicator.

If you do not specify `WITH COUNT` or `WITHOUT COUNT`, then the default is `WITH COUNT`.

Aggregate Examples

The following single-set aggregate example lists all of the employees in Department 30 in the `hr.employees` table, ordered by hire date and last name:

```
SELECT LISTAGG(last_name, '; ')
         WITHIN GROUP (ORDER BY hire_date, last_name) "Emp_list",
       MIN(hire_date) "Earliest"
  FROM employees
  WHERE department_id = 30;

Emp_list                                                     Earliest
```

```
------------------------------------------------------------ ---------
Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares          07-DEC-02
```

The following group-set aggregate example lists, for each department ID in the `hr.employees` table, the employees in that department in order of their hire date:

```
SELECT department_id "Dept.",
       LISTAGG(last_name, '; ') WITHIN GROUP (ORDER BY hire_date) "Employees"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Dept. Employees
------ ------------------------------------------------------------
    10 Whalen
    20 Hartstein; Fay
    30 Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares
    40 Mavris
    50 Kaufling; Ladwig; Rajs; Sarchand; Bell; Mallin; Weiss; Davie
       s; Marlow; Bull; Everett; Fripp; Chung; Nayer; Dilly; Bissot
       ; Vollman; Stiles; Atkinson; Taylor; Seo; Fleaur; Matos; Pat
       el; Walsh; Feeney; Dellinger; McCain; Vargas; Gates; Rogers;
        Mikkilineni; Landry; Cabrio; Jones; Olson; OConnell; Sulliv
       an; Mourgos; Gee; Perkins; Grant; Geoni; Philtanker; Markle
    60 Austin; Hunold; Pataballa; Lorentz; Ernst
    70 Baer
. . .
```

The following example is identical to the previous example, except it contains the `ON OVERFLOW TRUNCATE` clause. For the purpose of this example, assume that the maximum length of the return value is an artificially small number of 200 bytes. Because the list of employees for department 50 exceeds 200 bytes, the list is truncated and appended with a final delimiter '`; `', the specified truncation indicator '`...`', and the number of truncated values '`(23)`'.

```
SELECT department_id "Dept.",
       LISTAGG(last_name, '; ' ON OVERFLOW TRUNCATE '...')
               WITHIN GROUP (ORDER BY hire_date) "Employees"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Dept. Employees
------ ------------------------------------------------------------
    10 Whalen
    20 Hartstein; Fay
    30 Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares
    40 Mavris
    50 Kaufling; Ladwig; Rajs; Sarchand; Bell; Mallin; Weiss; Davie
       s; Marlow; Bull; Everett; Fripp; Chung; Nayer; Dilly; Bissot
       ; Vollman; Stiles; Atkinson; Taylor; Seo; Fleaur; ... (23)
    70 Baer
. . .
```

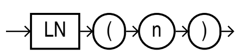
Analytic Example

The following analytic example shows, for each employee hired earlier than September 1, 2003, the employee's department, hire date, and all other employees in that department also hired before September 1, 2003:

```
SELECT department_id "Dept", hire_date "Date", last_name "Name",
       LISTAGG(last_name, '; ') WITHIN GROUP (ORDER BY hire_date, last_name)
         OVER (PARTITION BY department_id) as "Emp_list"
  FROM employees
  WHERE hire_date < '01-SEP-2003'
  ORDER BY "Dept", "Date", "Name";

 Dept Date      Name            Emp_list
----- --------- --------------- ---------------------------------------------
   30 07-DEC-02 Raphaely        Raphaely; Khoo
   30 18-MAY-03 Khoo            Raphaely; Khoo
   40 07-JUN-02 Mavris          Mavris
   50 01-MAY-03 Kaufling        Kaufling; Ladwig
   50 14-JUL-03 Ladwig          Kaufling; Ladwig
   70 07-JUN-02 Baer            Baer
   90 13-JAN-01 De Haan         De Haan; King
   90 17-JUN-03 King            De Haan; King
  100 16-AUG-02 Faviet          Faviet; Greenberg
  100 17-AUG-02 Greenberg       Faviet; Greenberg
  110 07-JUN-02 Gietz           Gietz; Higgins
  110 07-JUN-02 Higgins         Gietz; Higgins
```

LN

Syntax



Purpose

`LN` returns the natural logarithm of *n*, where *n* is greater than 0.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

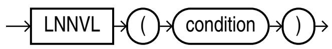
Examples

The following example returns the natural logarithm of 95:

```
SELECT LN(95) "Natural log of 95"
  FROM DUAL;

Natural log of 95
-----------------
       4.55387689
```

LNNVL

Syntax



Purpose

LNNVL provides a concise way to evaluate a condition when one or both operands of the condition may be null. The function can be used in the WHERE clause of a query, or as the WHEN condition in a searched CASE expression. It takes as an argument a condition and returns TRUE if the condition is FALSE or UNKNOWN and FALSE if the condition is TRUE. LNNVL can be used anywhere a scalar expression can appear, even in contexts where the IS [NOT] NULL, AND, or OR conditions are not valid but would otherwise be required to account for potential nulls.

Oracle Database sometimes uses the LNNVL function internally in this way to rewrite NOT IN conditions as NOT EXISTS conditions. In such cases, output from EXPLAIN PLAN shows this operation in the plan table output. The *condition* can evaluate any scalar values but cannot be a compound condition containing AND, OR, or BETWEEN.

The table that follows shows what LNNVL returns given that a = 2 and b is null.

Condition	Truth of Condition	LNNVL Return Value
a = 1	FALSE	TRUE
a = 2	TRUE	FALSE
a IS NULL	FALSE	TRUE
b = 1	UNKNOWN	TRUE
b IS NULL	TRUE	FALSE
a = b	UNKNOWN	TRUE

Examples

Suppose that you want to know the number of employees with commission rates of less than 20%, including employees who do not receive commissions. The following query returns only employees who actually receive a commission of less than 20%:

```
SELECT COUNT(*)
  FROM employees
  WHERE commission_pct < .2;

  COUNT(*)
----------
        11
```

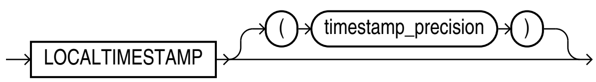
To include the 72 employees who receive no commission at all, you could rewrite the query using the LNNVL function as follows:

```
SELECT COUNT(*)
  FROM employees
```

```
  WHERE LNNVL(commission_pct >= .2);

  COUNT(*)
----------
        83
```

LOCALTIMESTAMP

Syntax



Purpose

`LOCALTIMESTAMP` returns the current date and time in the session time zone in a value of data type `TIMESTAMP`. The difference between this function and `CURRENT_TIMESTAMP` is that `LOCALTIMESTAMP` returns a `TIMESTAMP` value while `CURRENT_TIMESTAMP` returns a `TIMESTAMP WITH TIME ZONE` value.

The optional argument *`timestamp_precision`* specifies the fractional second precision of the time value returned.

> **See Also:**
>
> CURRENT_TIMESTAMP , "TIMESTAMP Data Type ", and "TIMESTAMP WITH TIME ZONE Data Type "

Examples

This example illustrates the difference between `LOCALTIMESTAMP` and `CURRENT_TIMESTAMP`:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                    LOCALTIMESTAMP
-------------------------------------------------------------------
04-APR-00 01.27.18.999220 PM -05:00  04-APR-00 01.27.19 PM

ALTER SESSION SET TIME_ZONE = '-8:00';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                    LOCALTIMESTAMP
-----------------------------------  ------------------------------
04-APR-00 10.27.45.132474 AM -08:00  04-APR-00 10.27.451 AM
```

When you use the `LOCALTIMESTAMP` with a format mask, take care that the format mask matches the value returned by the function. For example, consider the following table:

```
CREATE TABLE local_test (col1 TIMESTAMP WITH LOCAL TIME ZONE);
```

The following statement fails because the mask does not include the `TIME ZONE` portion of the return type of the function:

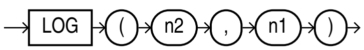
```
INSERT INTO local_test
  VALUES (TO_TIMESTAMP(LOCALTIMESTAMP, 'DD-MON-RR HH.MI.SSXFF'));
```

The following statement uses the correct format mask to match the return type of `LOCALTIMESTAMP`:

```
INSERT INTO local_test
  VALUES (TO_TIMESTAMP(LOCALTIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM'));
```

LOG

Syntax



Purpose

`LOG` returns the logarithm, base *`n2`*, of *`n1`*. The base *`n2`* can be any positive value other than 0 or 1 and *`n1`* can be any positive value.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If any argument is `BINARY_FLOAT` or `BINARY_DOUBLE`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns `NUMBER`.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

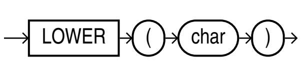
Examples

The following example returns the log of 100:

```
SELECT LOG(10,100) "Log base 10 of 100"
  FROM DUAL;

Log base 10 of 100
------------------
                 2
```

LOWER

Syntax



Purpose

LOWER returns *char*, with all letters lowercase. *char* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same data type as *char*. The database sets the case of the characters based on the binary mapping defined for the underlying character set. For linguistic-sensitive lowercase, refer to NLS_LOWER .

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of LOWER

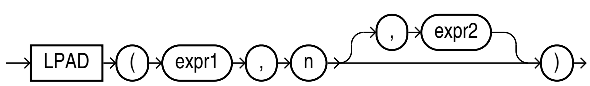
Examples

The following example returns a string in lowercase:

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
  FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax



Purpose

LPAD returns *expr1*, left-padded to length *n* characters with the sequence of characters in *expr2*. This function is useful for formatting the output of a query.

Both *expr1* and *expr2* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 data type if *expr1* is a character data type, NVARCHAR2 if *expr1* is a national character data type, and a LOB if *expr1* is a LOB data type. The string returned is in the same character set as *expr1*. The argument *n* must be a NUMBER integer or a value that can be implicitly converted to a NUMBER integer.

If you do not specify *expr2*, then the default is a single blank. If *expr1* is longer than *n*, then this function returns the portion of *expr1* that fits in *n*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `LPAD`

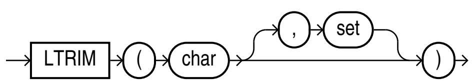
Examples

The following example left-pads a string with the asterisk (*) and period (.) characters:

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
  FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax



Purpose

`LTRIM` removes from the left end of *`char`* all of the characters contained in *`set`*. If you do not specify *`set`*, then it defaults to a single blank. Oracle Database begins scanning *`char`* from its first character and removes all characters that appear in *`set`* until reaching a character not in *`set`* and then returns the result.

Both *`char`* and *`set`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`char`* is a character data type, `NVARCHAR2` if *`char`* is a national character data type, and a LOB if *`char`* is a LOB data type.

See Also:

- RTRIM
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `LTRIM` uses to compare characters from *`set`* with characters from *`char`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

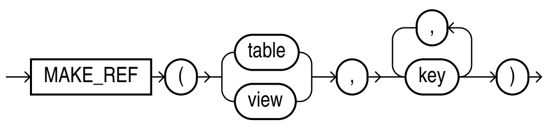
Examples

The following example trims all the left-most occurrences of less than sign (<), greater than sign (>) , and equal sign (=) from a string:

```
SELECT LTRIM('<=====>BROWNING<=====>', '<>=') "LTRIM Example"
  FROM DUAL;

LTRIM Example
---------------
BROWNING<=====>
```

MAKE_REF

Syntax



Purpose

`MAKE_REF` creates a `REF` to a row of an object view or a row in an object table whose object identifier is primary key based. This function is useful, for example, if you are creating an object view

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information about object views and DEREF

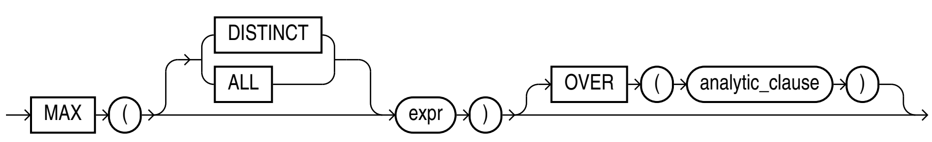
Examples

The sample schema `oe` contains an object view `oc_inventories` based on `inventory_typ`. The object identifier is `product_id`. The following example creates a `REF` to the row in the `oc_inventories` object view with a `product_id` of 3003:

```
SELECT MAKE_REF (oc_inventories, 3003)
  FROM DUAL;

MAKE_REF(OC_INVENTORIES,3003)
------------------------------------------------------------------
00004A038A0046857C14617141109EE03408002082543600000014260100010001
00290090606002A00078401FE0000000B03C21F040000000000000000000000000
0000000000
```

MAX

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`MAX` returns maximum value of *`expr`*. You can use it as an aggregate or analytic function.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *`expr`*, "Floating-Point Numbers " for information on binary-float comparison semantics, and "Aggregate Functions "
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `MAX` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Aggregate Example

The following example determines the highest salary in the `hr.employees` table:

```
SELECT MAX(salary) "Maximum"
  FROM employees;

   Maximum
----------
     24000
```

Analytic Examples

The following example calculates, for each employee, the highest salary of the employees reporting to the same manager as the employee.

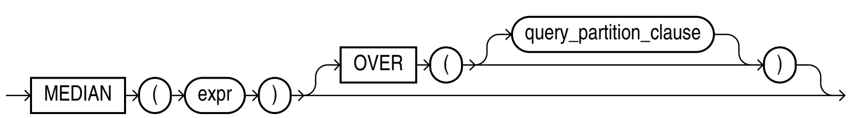
```
SELECT manager_id, last_name, salary,
       MAX(salary) OVER (PARTITION BY manager_id) AS mgr_max
  FROM employees
  ORDER BY manager_id, last_name, salary;

MANAGER_ID LAST_NAME                     SALARY    MGR_MAX
---------- ------------------------- ---------- ----------
       100 Cambrault                      11000      17000
       100 De Haan                        17000      17000
       100 Errazuriz                      12000      17000
       100 Fripp                           8200      17000
       100 Hartstein                      13000      17000
       100 Kaufling                        7900      17000
       100 Kochhar                        17000      17000
. . .
```

If you enclose this query in the parent query with a predicate, then you can determine the employee who makes the highest salary in each department:

```
SELECT manager_id, last_name, salary
  FROM (SELECT manager_id, last_name, salary,
               MAX(salary) OVER (PARTITION BY manager_id) AS rmax_sal
          FROM employees)
  WHERE salary = rmax_sal
  ORDER BY manager_id, last_name, salary;

MANAGER_ID LAST_NAME                     SALARY
---------- ------------------------- ----------
       100 De Haan                        17000
       100 Kochhar                        17000
       101 Greenberg                      12008
       101 Higgins                        12008
       102 Hunold                          9000
       103 Ernst                           6000
       108 Faviet                          9000
       114 Khoo                            3100
       120 Nayer                           3200
       120 Taylor                          3200
       121 Sarchand                        4200
       122 Chung                           3800
       123 Bell                            4000
       124 Rajs                            3500
       145 Tucker                         10000
       146 King                           10000
       147 Vishney                        10500
       148 Ozer                           11500
       149 Abel                           11000
       201 Fay                             6000
       205 Gietz                           8300
           King                           24000

22 rows selected.
```

MEDIAN

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`MEDIAN` is an inverse distribution function that assumes a continuous distribution model. It takes a numeric or datetime value and returns the middle value or an interpolated value that would be the middle value once the values are sorted. Nulls are ignored in the calculation.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If you specify only *expr*, then the function returns the same data type as the numeric data type of the argument. If you specify the `OVER` clause, then Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

The result of `MEDIAN` is computed by first ordering the rows. Using `N` as the number of rows in the group, Oracle calculates the row number (`RN`) of interest with the formula `RN` = (`1` + (`0.5`*(`N`-`1`)). The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers `CRN` = `CEILING`(`RN`) and `FRN` = `FLOOR`(`RN`).

The final result will be:

```
if (CRN = FRN = RN) then
   (value of expression from row at RN)
else
   (CRN - RN) * (value of expression for row at FRN) +
   (RN - FRN) * (value of expression for row at CRN)
```

You can use `MEDIAN` as an analytic function. You can specify only the *`query_partition_clause`* in its `OVER` clause. It returns, for each row, the value that would fall in the middle among a set of values within each partition.

Compare this function with these functions:

- PERCENTILE_CONT , which returns, for a given percentile, the value that corresponds to that percentile by way of interpolation. `MEDIAN` is the specific case of `PERCENTILE_CONT` where the percentile value defaults to 0.5.
- PERCENTILE_DISC , which is useful for finding values for a given percentile without interpolation.

Aggregate Example

The following query returns the median salary for each department in the `hr.employees` table:

```
SELECT department_id, MEDIAN(salary)
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

DEPARTMENT_ID MEDIAN(SALARY)
------------- --------------
           10           4400
           20           9500
           30           2850
           40           6500
           50           3100
           60           4800
           70          10000
```

```
           80           8900
           90          17000
          100           8000
          110          10154
                        7000
```

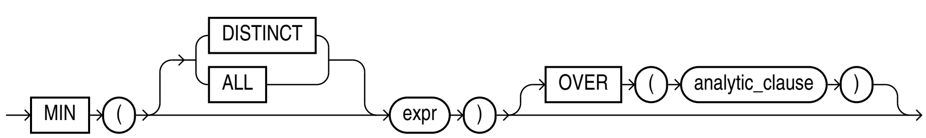
Analytic Example

The following query returns the median salary for each manager in a subset of departments in the `hr.employees` table:

```
SELECT manager_id, employee_id, salary,
       MEDIAN(salary) OVER (PARTITION BY manager_id) "Median by Mgr"
  FROM employees
  WHERE department_id > 60
  ORDER BY manager_id, employee_id;

MANAGER_ID EMPLOYEE_ID     SALARY Median by Mgr
---------- ----------- ---------- -------------
       100         101      17000         13500
       100         102      17000         13500
       100         145      14000         13500
       100         146      13500         13500
       100         147      12000         13500
       100         148      11000         13500
       100         149      10500         13500
       101         108      12008         12008
       101         204      10000         12008
       101         205      12008         12008
       108         109       9000          7800
       108         110       8200          7800
       108         111       7700          7800
       108         112       7800          7800
       108         113       6900          7800
       145         150      10000          8500
       145         151       9500          8500
       145         152       9000          8500
. . .
```

MIN

Syntax



See Also:

"Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`MIN` returns minimum value of *`expr`*. You can use it as an aggregate or analytic function.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *`expr`*, "Floating-Point Numbers " for information on binary-float comparison semantics, and "Aggregate Functions "
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `MIN` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Aggregate Example

The following statement returns the earliest hire date in the `hr.employees` table:

```
SELECT MIN(hire_date) "Earliest"
  FROM employees;

Earliest
---------
13-JAN-01
```

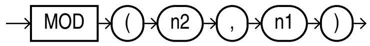
Analytic Example

The following example determines, for each employee, the employees who were hired on or before the same date as the employee. It then determines the subset of employees reporting to the same manager as the employee, and returns the lowest salary in that subset.

```
SELECT manager_id, last_name, hire_date, salary,
       MIN(salary) OVER(PARTITION BY manager_id ORDER BY hire_date
         RANGE UNBOUNDED PRECEDING) AS p_cmin
  FROM employees
  ORDER BY manager_id, last_name, hire_date, salary;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     P_CMIN
---------- ------------------------- --------- ---------- ----------
       100 Cambrault                 15-OCT-07      11000       6500
       100 De Haan                   13-JAN-01      17000      17000
       100 Errazuriz                 10-MAR-05      12000       7900
       100 Fripp                     10-APR-05       8200       7900
       100 Hartstein                 17-FEB-04      13000       7900
       100 Kaufling                  01-MAY-03       7900       7900
       100 Kochhar                   21-SEP-05      17000       7900
       100 Mourgos                   16-NOV-07       5800       5800
       100 Partners                  05-JAN-05      13500       7900
       100 Raphaely                  07-DEC-02      11000      11000
       100 Russell                   01-OCT-04      14000       7900

. . .
```

MOD

Syntax



Purpose

`MOD` returns the remainder of *n2* divided by *n1*. Returns *n2* if *n1* is 0.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Examples

The following example returns the remainder of 11 divided by 4:

```
SELECT MOD(11,4) "Modulus"
  FROM DUAL;

   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the `MOD` function with this formula:

```
n2 - n1 * FLOOR(n2/n1)
```

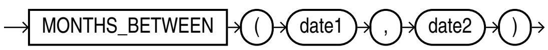
The following table illustrates the difference between the `MOD` function and the classical modulus:

n2	n1	MOD(n2,n1)	Classical Modulus
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3

> **See Also:**
>
> FLOOR and REMAINDER , which is similar to MOD, but uses ROUND in its formula instead of FLOOR

MONTHS_BETWEEN

Syntax



Purpose

MONTHS_BETWEEN returns number of months between dates *date1* and *date2*. The month and the last day of the month are defined by the parameter NLS_CALENDAR. If *date1* is later than *date2*, then the result is positive. If *date1* is earlier than *date2*, then the result is negative. If *date1* and *date2* are either the same days of the month or both last days of months, then the result is always an integer. Otherwise Oracle Database calculates the fractional portion of the result based on a 31-day month and considers the difference in time components *date1* and *date2*.

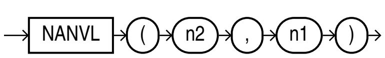
Examples

The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN
       (TO_DATE('02-02-1995','MM-DD-YYYY'),
        TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
  FROM DUAL;

    Months
----------
1.03225806
```

NANVL

Syntax



Purpose

The NANVL function is useful only for floating-point numbers of type BINARY_FLOAT or BINARY_DOUBLE. It instructs Oracle Database to return an alternative value *n1* if the input value *n2* is NaN (not a number). If *n2* is **not** NaN, then Oracle returns *n2*.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion, "Floating-Point Numbers " for information on binary-float comparison semantics, and "Numeric Precedence " for information on numeric precedence

Examples

Using table `float_point_demo` created for TO_BINARY_DOUBLE , insert a second entry into the table:

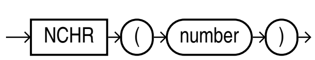
```
INSERT INTO float_point_demo
  VALUES (0,'NaN','NaN');

SELECT *
  FROM float_point_demo;

   DEC_NUM BIN_DOUBLE  BIN_FLOAT
---------- ---------- ----------
   1234.56 1.235E+003 1.235E+003
         0        Nan        Nan
```

The following example returns *`bin_float`* if it is a number. Otherwise, 0 is returned.

```
SELECT bin_float, NANVL(bin_float,0)
  FROM float_point_demo;

 BIN_FLOAT NANVL(BIN_FLOAT,0)
---------- ------------------
1.235E+003         1.235E+003
       Nan                  0
```

NCHR

Syntax



Purpose

`NCHR` returns the character having the binary equivalent to *`number`* in the national character set. The value returned is always `NVARCHAR2`. This function is equivalent to using the `CHR` function with the `USING NCHAR_CS` clause.

This function takes as an argument a `NUMBER` value, or any value that can be implicitly converted to `NUMBER`, and returns a character.

See Also:

- CHR
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `NCHR`

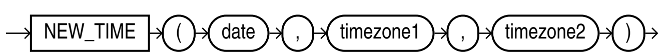
Examples

The following examples return the nchar character 187:

```
SELECT NCHR(187)
  FROM DUAL;

N
-
>

SELECT CHR(187 USING NCHAR_CS)
  FROM DUAL;

C
-
>
```

NEW_TIME

Syntax



Purpose

`NEW_TIME` returns the date and time in time zone *`timezone2`* when date and time in time zone *`timezone1`* are *`date`*. Before using this function, you must set the `NLS_DATE_FORMAT` parameter to display 24-hour time. The return type is always `DATE`, regardless of the data type of *`date`*.

Note:

This function takes as input only a limited number of time zones. You can have access to a much greater number of time zones by combining the `FROM_TZ` function and the datetime expression. See FROM_TZ and the example for "Datetime Expressions ".

The arguments *`timezone1`* and *`timezone2`* can be any of these text strings:

- AST, ADT: Atlantic Standard or Daylight Time

- BST, BDT: Bering Standard or Daylight Time
- CST, CDT: Central Standard or Daylight Time
- EST, EDT: Eastern Standard or Daylight Time
- GMT: Greenwich Mean Time
- HST, HDT: Alaska-Hawaii Standard Time or Daylight Time.
- MST, MDT: Mountain Standard or Daylight Time
- NST: Newfoundland Standard Time
- PST, PDT: Pacific Standard or Daylight Time
- YST, YDT: Yukon Standard or Daylight Time

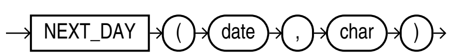
Examples

The following example returns an Atlantic Standard time, given the Pacific Standard time equivalent:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';

SELECT NEW_TIME(TO_DATE('11-10-09 01:23:45', 'MM-DD-YY HH24:MI:SS'), 'AST',
'PST')
         "New Date and Time"
  FROM DUAL;

New Date and Time
--------------------
09-NOV-2009 21:23:45
```

NEXT_DAY

Syntax



Purpose

`NEXT_DAY` returns the date of the first weekday named by *`char`* that is later than the date *`date`*. The return type is always `DATE`, regardless of the data type of *`date`*. The argument *`char`* must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *`date`*.

Examples

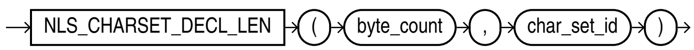
This example returns the date of the next Tuesday after October 15, 2009:

```
SELECT NEXT_DAY('15-OCT-2009','TUESDAY') "NEXT DAY"
  FROM DUAL;

NEXT DAY
```

```
--------------------
20-OCT-2009 00:00:00
```

NLS_CHARSET_DECL_LEN

Syntax



Purpose

`NLS_CHARSET_DECL_LEN` returns the declaration length (in number of characters) of an `NCHAR` column. The *`byte_count`* argument is the width of the column. The *`char_set_id`* argument is the character set ID of the column.

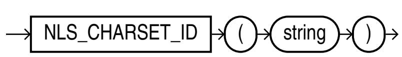
Examples

The following example returns the number of characters that are in a 200-byte column when you are using a multibyte character set:

```
SELECT NLS_CHARSET_DECL_LEN(200, nls_charset_id('ja16eucfixed'))
  FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
--------------------------------------------------------
                                                     100
```

NLS_CHARSET_ID

Syntax



Purpose

`NLS_CHARSET_ID` returns the character set ID number corresponding to character set name *`string`*. The *`string`* argument is a run-time `VARCHAR2` value. The *`string`* value '`CHAR_CS`' returns the database character set ID number of the server. The *`string`* value '`NCHAR_CS`' returns the national character set ID number of the server.

Invalid character set names return null.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for a list of character sets

Examples

The following example returns the character set ID of a character set:

```
SELECT NLS_CHARSET_ID('ja16euc')
  FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
-------------------------
                      830
```

NLS_CHARSET_NAME

Syntax

Purpose

`NLS_CHARSET_NAME` returns the name of the character set corresponding to ID number *`number`*. The character set name is returned as a `VARCHAR2` value in the database character set. If *`number`* is not recognized as a valid character set ID, then this function returns null.

This function returns a `VARCHAR2` value.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `NLS_CHARSET_NAME`

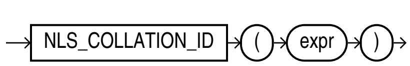
Examples

The following example returns the character set corresponding to character set ID number 2:

```
SELECT NLS_CHARSET_NAME(2)
  FROM DUAL;

NLS_CH
------
WE8DEC
```

NLS_COLLATION_ID

Syntax



Purpose

NLS_COLLATION_ID takes as its argument a collation name and returns the corresponding collation ID number. Collation IDs are used in the data dictionary tables and in Oracle Call Interface (OCI). Collation names are used in SQL statements and data dictionary views

For *expr*, specify the collation name as a VARCHAR2 value. You can specify a valid named collation or a pseudo-collation, in any combination of uppercase and lowercase letters.

This function returns a NUMBER value. If you specify an invalid collation name, then this function returns null.

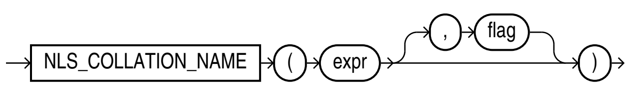
Examples

The following example returns the collation ID of collation BINARY_CI:

```
SELECT NLS_COLLATION_ID('BINARY_CI')
  FROM DUAL;

NLS_COLLATION_ID('BINARY_CI')
-----------------------------
                       147455
```

NLS_COLLATION_NAME

Syntax



Purpose

NLS_COLLATION_NAME takes as its argument a collation ID number and returns the corresponding collation name. Collation IDs are used in the data dictionary tables and in Oracle Call Interface (OCI). Collation names are used in SQL statements and data dictionary views

For *expr*, specify the collation ID as a NUMBER value.

This function returns a VARCHAR2 value. If you specify an invalid collation ID, then this function returns null.

The optional *flag* parameter applies only to Unicode Collation Algorithm (UCA) collations. This parameter determines whether the function returns the short form or long form of the collation name. The parameter must be a character expression evaluating to the value 'S', 's', 'L', or 'l', with the following meaning:

- 'S' or 's' – Returns the short form of the collation name
- 'L' or 'l' – Returns the long form of the collation name

If you omit *flag*, then the default is 'L'.

See Also:

- *Oracle Database Globalization Support Guide* for more information on UCA collations
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `NLS_COLLATION_NAME`

Examples

The following example returns the name of the collation corresponding to collation ID number 81919:

```
SELECT NLS_COLLATION_NAME(81919)
  FROM DUAL;

NLS_COLLA
---------
BINARY_AI
```

The following example returns the short form of the name of the UCA collation corresponding to collation ID number 208897:

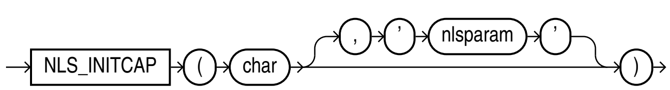
```
SELECT NLS_COLLATION_NAME(208897,'S')
  FROM DUAL;

NLS_COLLATION
-------------
UCA0610_DUCET
```

The following example returns the long form of the name of the UCA collation corresponding to collation ID number 208897:

```
SELECT NLS_COLLATION_NAME(208897,'L')
  FROM DUAL;

NLS_COLLATION_NAME(208897,'L')
----------------------------------------
UCA0610_DUCET_S4_VS_BN_NY_EN_FN_HN_DN_MN
```

NLS_INITCAP

Syntax



Purpose

`NLS_INITCAP` returns *`char`*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Both *char* and *'nlsparam'* can be any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as *char*.

The value of *'nlsparam'* can have this form:

```
'NLS_SORT = sort'
```

where *sort* is a named collation. The collation handles special linguistic requirements for case conversions. These requirements can result in a return value of a different length than the *char*. If you omit 'nlsparam', then this function uses the determined collation of the function.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for NLS_INITCAP , and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

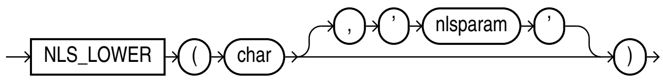
The following examples show how the linguistic sort sequence results in a different return value from the function:

```
SELECT NLS_INITCAP('ijsland') "InitCap"
  FROM DUAL;

InitCap
-------
Ijsland

SELECT NLS_INITCAP('ijsland', 'NLS_SORT = XDutch') "InitCap"
  FROM DUAL;

InitCap
-------
IJsland
```

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on collations

NLS_LOWER

Syntax



Purpose

`NLS_LOWER` returns *`char`*, with all letters lowercase.

Both *`char`* and *`'nlsparam'`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`char`* is a character data type and a LOB if *`char`* is a LOB data type. The return string is in the same character set as *`char`*.

The *`'nlsparam'`* can have the same form and serve the same purpose as in the `NLS_INITCAP` function.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `NLS_LOWER`, and for the collation derivation rules, which define the collation assigned to the character return value of this function

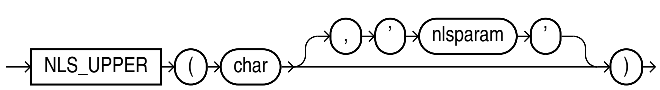
Examples

The following statement returns the lowercase form of the character string '`NOKTASINDA`' using the XTurkish linguistic sort sequence. The Turkish uppercase I becoming a small, dotless i.

```
SELECT NLS_LOWER('NOKTASINDA', 'NLS_SORT = XTurkish') "Lowercase"
  FROM DUAL;
```

NLS_UPPER

Syntax



Purpose

`NLS_UPPER` returns *`char`*, with all letters uppercase.

Both *`char`* and *`'nlsparam'`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`char`* is a character data type and a LOB if *char* is a LOB data type. The return string is in the same character set as *char*.

The '*nlsparam*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for NLS_UPPER, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

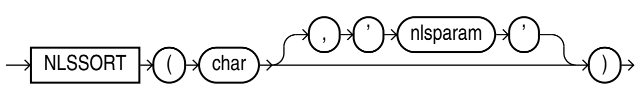
The following example returns a string with all the letters converted to uppercase:

```
SELECT NLS_UPPER('große') "Uppercase"
  FROM DUAL;

Upper
-----
GROßE

SELECT NLS_UPPER('große', 'NLS_SORT = XGerman') "Uppercase"
  FROM DUAL;

Upperc
------
GROSSE
```

> **See Also:**
>
> NLS_INITCAP

NLSSORT

Syntax



Purpose

NLSSORT returns a collation key for the character value *char* and an explicitly or implicitly specified collation. A collation key is a string of bytes used to sort *char* according to the specified collation. The property of the collation keys is that mutual ordering of two such keys generated for the given collation when compared according to their binary order is the same as mutual ordering of the source character values when compared according to the given collation.

Both *char* and '*nlsparam*' can be any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

The value of '*nlsparam*' must have the form

```
'NLS_SORT = collation'
```

where *collation* is the name of a linguistic collation or BINARY. NLSSORT uses the specified collation to generate the collation key. If you omit '*nlsparam*', then this function uses the derived collation of the argument *char*. If you specify BINARY, then this function returns the *char* value itself cast to RAW and possibly truncated as described below.

If you specify '*nlsparam*', then you can append to the linguistic collation name the suffix _ai to request an accent-insensitive collation or _ci to request a case-insensitive collation. Refer to *Oracle Database Globalization Support Guide* for more information on accent- and case-insensitive sorting. Using accent-insensitive or case-insensitive collations with the ORDER BY query clause is not recommended as it leads to a nondeterministic sort order.

The returned collation key is of RAW data type. The length of the collation key resulting from a given *char* value for a given collation may exceed the maximum length of the RAW value returned by NLSSORT. In this case, the behavior of NLSSORT depends on the value of the initialization parameter MAX_STRING_SIZE. If MAX_STRING_SIZE = EXTENDED, then the maximum length of the return value is 32767 bytes. If the collation key exceeds this limit, then the function fails with the error "ORA-12742: unable to create the collation key". This error may also be reported for short input strings if they contain a high percentage of Unicode characters with very high decomposition ratios.

See Also:

Oracle Database Globalization Support Guide for details of when the ORA-12742 error is reported and how to prevent application availability issues that the error could cause

If MAX_STRING_SIZE = STANDARD, then the maximum length of the return value is 2000 bytes. If the value to be returned exceeds the limit, then NLSSORT calculates the collation key for a maximum prefix, or initial substring, of *char* so that the calculated result does not exceed the maximum length. For monolingual collations, for example FRENCH, the prefix length is typically 1000 characters. For multilingual collations, for example GENERIC_M, the prefix is typically 500 characters. For Unicode Collation Algorithm (UCA) collations, for example UCA0610_DUCET, the prefix is typically 285 characters. The exact length may be lower or higher depending on the collation and the characters contained in *char*.

The behavior when MAX_STRING_SIZE = STANDARD implies that two character values whose collation keys (NLSSORT results) are compared to find the linguistic ordering are considered equal if they do not differ in the prefix even though they may differ at some further character position. Because the NLSSORT function is used implicitly to find linguistic ordering for comparison conditions, the BETWEEN condition, the IN condition, ORDER BY, GROUP BY, and COUNT(DISTINCT), those operations may return results that are only approximate for long character values. If you want guarantee that the results of those operations are exact, then migrate your database to use MAX_STRING_SIZE = EXTENDED.

Refer to "Extended Data Types" for more information on the MAX_STRING_SIZE initialization parameter.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for NLSSORT

Examples

This function can be used to specify sorting and comparison operations based on a linguistic sort sequence rather than on the binary value of a string. The following example creates a test table containing two values and shows how the values returned can be ordered by the NLSSORT function:

```
CREATE TABLE test (name VARCHAR2(15));
INSERT INTO test VALUES ('Gaardiner');
INSERT INTO test VALUES ('Gaberd');
INSERT INTO test VALUES ('Gaasten');

SELECT *
  FROM test
  ORDER BY name;

NAME
---------------
Gaardiner
Gaasten
Gaberd

SELECT *
  FROM test
  ORDER BY NLSSORT(name, 'NLS_SORT = XDanish');

NAME
---------------
Gaberd
Gaardiner
Gaasten
```

The following example shows how to use the NLSSORT function in comparison operations:

```
SELECT *
  FROM test
  WHERE name > 'Gaberd'
  ORDER BY name;

no rows selected
```

```
SELECT *
  FROM test
  WHERE NLSSORT(name, 'NLS_SORT = XDanish') >
        NLSSORT('Gaberd', 'NLS_SORT = XDanish')
  ORDER BY name;

NAME
---------------
Gaardiner
Gaasten
```

If you frequently use `NLSSORT` in comparison operations with the same linguistic sort sequence, then consider this more efficient alternative: Set the `NLS_COMP` parameter (either for the database or for the current session) to `LINGUISTIC`, and set the `NLS_SORT` parameter for the session to the desired sort sequence. Oracle Database will use that sort sequence by default for all sorting and comparison operations during the current session:

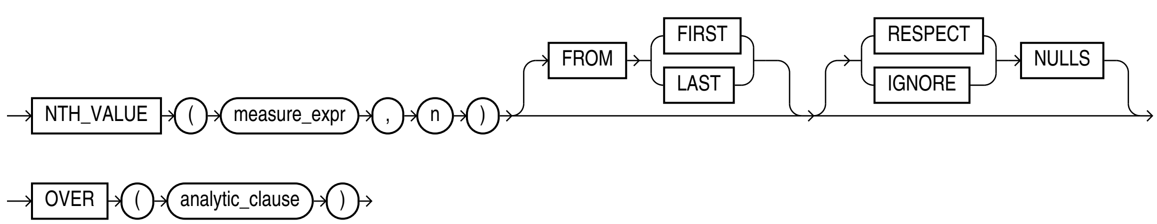
```
ALTER SESSION SET NLS_COMP = 'LINGUISTIC';
ALTER SESSION SET NLS_SORT = 'XDanish';

SELECT *
  FROM test
  WHERE name > 'Gaberd'
  ORDER BY name;

NAME
---------------
Gaardiner
Gaasten
```

See Also:

Oracle Database Globalization Support Guide for information on sort sequences

NTH_VALUE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the *analytic_clause*

Purpose

NTH_VALUE returns the *measure_expr* value of the *n*th row in the window defined by the *analytic_clause*. The returned value has the data type of the *measure_expr*.

- {RESPECT | IGNORE} NULLS determines whether null values of *measure_expr* are included in or eliminated from the calculation. The default is RESPECT NULLS.
- *n* determines the *n*th row for which the measure value is to be returned. *n* can be a constant, bind variable, column, or an expression involving them, as long as it resolves to a positive integer. The function returns NULL if the data source window has fewer than *n* rows. If *n* is null, then the function returns an error.
- FROM {FIRST | LAST} determines whether the calculation begins at the first or last row of the window. The default is FROM FIRST.

If you omit the *windowing_clause* of the *analytic_clause*, it defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. This default sometimes returns an unexpected value for NTH_VALUE ... FROM LAST ... , because the last value in the window is at the bottom of the window, which is not fixed. It keeps changing as the current row changes. For expected results, specify the *windowing_clause* as RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Alternatively, you can specify the *windowing_clause* as RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for more information on the use of this function
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of NTH_VALUE when it is a character value

Examples

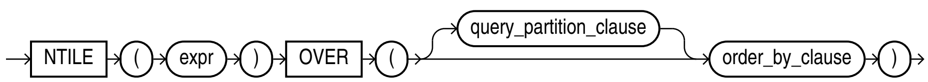
The following example shows the minimum *amount_sold* value for the second *channel_id* in ascending order for each *prod_id* between 13 and 16:

```
SELECT prod_id, channel_id, MIN(amount_sold),
    NTH_VALUE(MIN(amount_sold), 2) OVER (PARTITION BY prod_id ORDER BY channel_id
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) nv
  FROM sales
  WHERE prod_id BETWEEN 13 and 16
  GROUP BY prod_id, channel_id;

   PROD_ID CHANNEL_ID MIN(AMOUNT_SOLD)         NV
---------- ---------- ---------------- ----------
        13          2           907.34      906.2
```

```
        13          3        906.2      906.2
        13          4       842.21      906.2
        14          2      1015.94    1036.72
        14          3      1036.72    1036.72
        14          4       935.79    1036.72
        15          2       871.19     871.19
        15          3       871.19     871.19
        15          4       871.19     871.19
        16          2       266.84     266.84
        16          3       266.84     266.84
        16          4       266.84     266.84
        16          9        11.99     266.84

13 rows selected.
```

NTILE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

`NTILE` is an analytic function. It divides an ordered data set into a number of buckets indicated by *expr* and assigns the appropriate bucket number to each row. The buckets are numbered 1 through *expr*. The *expr* value must resolve to a positive constant for each partition. Oracle Database expects an integer, and if *expr* is a noninteger constant, then Oracle truncates the value to an integer. The return value is `NUMBER`.

The number of rows in the buckets can differ by at most 1. The remainder values (the remainder of number of rows divided by buckets) are distributed one for each bucket, starting with bucket 1.

If *expr* is greater than the number of rows, then a number of buckets equal to the number of rows will be filled, and the remaining buckets will be empty.

You cannot nest analytic functions by using `NTILE` or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*.

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *expr* and Table 2-8 for more information on implicit conversion

Examples

The following example divides into 4 buckets the values in the `salary` column of the `oe.employees` table from Department 100. The `salary` column has 6 values in this department, so the two extra values (the remainder of 6 / 4) are allocated to buckets 1 and 2, which therefore have one more value than buckets 3 or 4.

```
SELECT last_name, salary, NTILE(4) OVER (ORDER BY salary DESC) AS quartile
  FROM employees
  WHERE department_id = 100
  ORDER BY last_name, salary, quartile;

LAST_NAME                     SALARY   QUARTILE
------------------------- ---------- ----------
Chen                            8200          2
Faviet                          9000          1
Greenberg                      12008          1
Popp                            6900          4
Sciarra                         7700          3
Urman                           7800          2
```

NULLIF

Syntax

Purpose

`NULLIF` compares *`expr1`* and *`expr2`*. If they are equal, then the function returns null. If they are not equal, then the function returns *`expr1`*. You cannot specify the literal `NULL` for *`expr1`*.

If both arguments are numeric data types, then Oracle Database determines the argument with the higher numeric precedence, implicitly converts the other argument to that data type, and returns that data type. If the arguments are not numeric, then they must be of the same data type, or Oracle returns an error.

The `NULLIF` function is logically equivalent to the following `CASE` expression:

```
CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END
```

See Also:

- "CASE Expressions "
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `NULLIF` uses to compare characters from *`expr1`* with characters from *`expr2`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

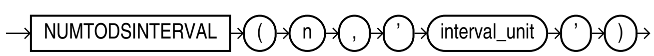
Examples

The following example selects those employees from the sample schema `hr` who have changed jobs since they were hired, as indicated by a `job_id` in the `job_history` table different from the current `job_id` in the `employees` table:

```
SELECT e.last_name, NULLIF(j.job_id, e.job_id) "Old Job ID"
  FROM employees e, job_history j
  WHERE e.employee_id = j.employee_id
  ORDER BY last_name, "Old Job ID";

LAST_NAME                 Old Job ID
------------------------- ----------
De Haan                   IT_PROG
Hartstein                 MK_REP
Kaufling                  ST_CLERK
Kochhar                   AC_ACCOUNT
Kochhar                   AC_MGR
Raphaely                  ST_CLERK
Taylor                    SA_MAN
Taylor
Whalen                    AC_ACCOUNT
Whalen
```

NUMTODSINTERVAL

Syntax



Purpose

`NUMTODSINTERVAL` converts *`n`* to an `INTERVAL DAY TO SECOND` literal. The argument *`n`* can be any `NUMBER` value or an expression that can be implicitly converted to a `NUMBER` value. The argument *`interval_unit`* can be of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type. The value for *`interval_unit`* specifies the unit of *`n`* and must resolve to one of the following string values:

- '`DAY`'
- '`HOUR`'
- '`MINUTE`'
- '`SECOND`'

`interval_unit` is case insensitive. Leading and trailing values within the parentheses are ignored. By default, the precision of the return is 9.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

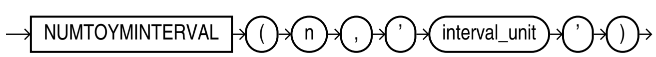
Examples

The following example uses NUMTODSINTERVAL in a COUNT analytic function to calculate, for each employee, the number of employees hired by the same manager within the past 100 days from his or her hire date. Refer to "Analytic Functions " for more information on the syntax of the analytic functions.

```
SELECT manager_id, last_name, hire_date,
       COUNT(*) OVER (PARTITION BY manager_id ORDER BY hire_date
       RANGE NUMTODSINTERVAL(100, 'day') PRECEDING) AS t_count
  FROM employees
  ORDER BY last_name, hire_date;

MANAGER_ID LAST_NAME                 HIRE_DATE    T_COUNT
---------- ------------------------- --------- ----------
       149 Abel                      11-MAY-04          1
       147 Ande                      24-MAR-08          3
       121 Atkinson                  30-OCT-05          2
       103 Austin                    25-JUN-05          1
. . .
       124 Walsh                     24-APR-06          2
       100 Weiss                     18-JUL-04          1
       101 Whalen                    17-SEP-03          1
       100 Zlotkey                   29-JAN-08          2
```

NUMTOYMINTERVAL

Syntax



Purpose

NUMTOYMINTERVAL converts number *n* to an INTERVAL YEAR TO MONTH literal. The argument *n* can be any NUMBER value or an expression that can be implicitly converted to a NUMBER value. The argument *interval_unit* can be of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The value for *interval_unit* specifies the unit of *n* and must resolve to one of the following string values:

- 'YEAR'
- 'MONTH'

interval_unit is case insensitive. Leading and trailing values within the parentheses are ignored. By default, the precision of the return is 9.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

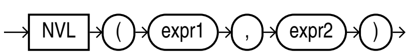
Examples

The following example uses `NUMTOYMINTERVAL` in a `SUM` analytic function to calculate, for each employee, the total salary of employees hired in the past one year from his or her hire date. Refer to "Analytic Functions " for more information on the syntax of the analytic functions.

```
SELECT last_name, hire_date, salary,
       SUM(salary) OVER (ORDER BY hire_date
       RANGE NUMTOYMINTERVAL(1,'year') PRECEDING) AS t_sal
  FROM employees
  ORDER BY last_name, hire_date;

LAST_NAME                 HIRE_DATE     SALARY      T_SAL
------------------------- --------- ---------- ----------
Abel                      11-MAY-04      11000      90300
Ande                      24-MAR-08       6400     112500
Atkinson                  30-OCT-05       2800     177000
Austin                    25-JUN-05       4800     134700
. . .
Walsh                     24-APR-06       3100     186200
Weiss                     18-JUL-04       8000      70900
Whalen                    17-SEP-03       4400      54000
Zlotkey                   29-JAN-08      10500     119000
```

NVL

Syntax



Purpose

`NVL` lets you replace null (returned as a blank) with a string in the results of a query. If *`expr1`* is null, then `NVL` returns *`expr2`*. If *`expr1`* is not null, then `NVL` returns *`expr1`*.

The arguments *`expr1`* and *`expr2`* can have any data type. If their data types are different, then Oracle Database implicitly converts one to the other. If they cannot be converted implicitly, then the database returns an error. The implicit conversion is implemented as follows:

- If *`expr1`* is character data, then Oracle Database converts *`expr2`* to the data type of *`expr1`* before comparing them and returns `VARCHAR2` in the character set of *`expr1`*.
- If *`expr1`* is numeric, then Oracle Database determines which argument has the highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

See Also:

- Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence
- "COALESCE " and "CASE Expressions ", which provide functionality similar to that of NVL
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of NVL when it is a character value

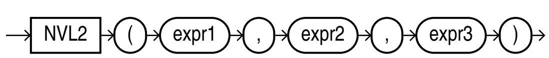
Examples

The following example returns a list of employee names and commissions, substituting "Not Applicable" if the employee receives no commission:

```
SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable') commission
  FROM employees
  WHERE last_name LIKE 'B%'
  ORDER BY last_name;

LAST_NAME                 COMMISSION
------------------------- ----------------------------------------
Baer                      Not Applicable
Baida                     Not Applicable
Banda                     .1
Bates                     .15
Bell                      Not Applicable
Bernstein                 .25
Bissot                    Not Applicable
Bloom                     .2
Bull                      Not Applicable
```

NVL2

Syntax



Purpose

NVL2 lets you determine the value returned by a query based on whether a specified expression is null or not null. If *expr1* is not null, then NVL2 returns *expr2*. If *expr1* is null, then NVL2 returns *expr3*.

The argument *expr1* can have any data type. The arguments *expr2* and *expr3* can have any data types except LONG.

If the data types of *expr2* and *expr3* are different, then Oracle Database implicitly converts one to the other. If they cannot be converted implicitly, then the database returns an error. If *expr2* is character or numeric data, then the implicit conversion is implemented as follows:

- If *expr2* is character data, then Oracle Database converts *expr3* to the data type of *expr2* before returning a value unless *expr3* is a null constant. In that case, a data type conversion is not necessary, and the database returns `VARCHAR2` in the character set of *expr2*.
- If *expr2* is numeric data, then Oracle Database determines which argument has the highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

> **See Also:**
>
> - Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `NVL2` when it is a character value

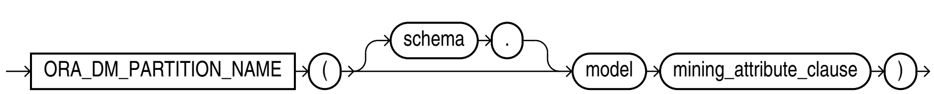
Examples

The following example shows whether the income of some employees is made up of salary plus commission, or just salary, depending on whether the `commission_pct` column of `employees` is null or not.

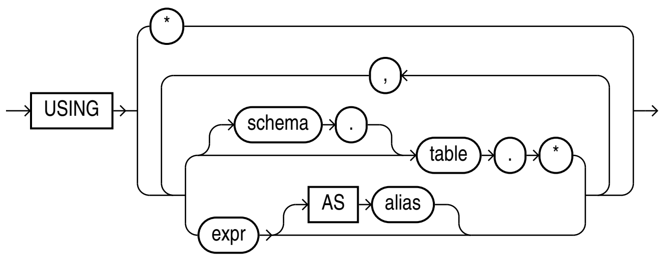
```
SELECT last_name, salary,
       NVL2(commission_pct, salary + (salary * commission_pct), salary) income
  FROM employees
  WHERE last_name like 'B%'
  ORDER BY last_name;

LAST_NAME                     SALARY     INCOME
------------------------- ---------- ----------
Baer                           10000      10000
Baida                           2900       2900
Banda                           6200       6820
Bates                           7300       8395
Bell                            4000       4000
Bernstein                       9500      11875
Bissot                          3300       3300
Bloom                          10000      12000
Bull                            4100       4100
```

ORA_DM_PARTITION_NAME

Syntax



***mining_attribute_clause*::=**



Purpose

`ORA_DM_PARTITION_NAME` is a single row function that works along with other existing functions. This function returns the name of the partition associated with the input row. When `ORA_DM_PARTITION_NAME` is used on a non-partitioned model, the result is `NULL`.

The syntax of the `ORA_DM_PARTITION_NAME` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

The *`mining_attribute_clause`* identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. See mining_attribute_clause.

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring
> - *Oracle Data Mining Concepts* for information about clustering

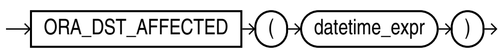
> **Note:**
>
> The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

```
SELECT prediction(mymodel using *) pred, ora_dm_partition_name(mymodel USING
*) pname FROM customers;
```

ORA_DST_AFFECTED

Syntax



Purpose

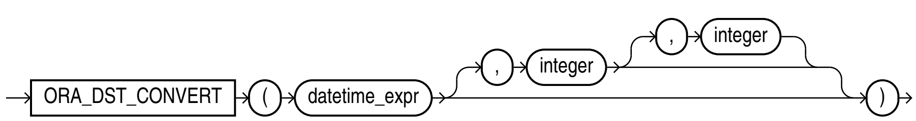
`ORA_DST_AFFECTED` is useful when you are changing the time zone data file for your database. The function takes as an argument a datetime expression that resolves to a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values. The function returns `1` if the datetime value is affected by or will result in a "nonexisting time" or "duplicate time" error with the new time zone data. Otherwise, it returns 0.

This function can be issued only when changing the time zone data file of the database and upgrading the timestamp with the time zone data, and only between the execution of the `DBMS_DST.BEGIN_PREPARE` and the `DBMS_DST.END_PREPARE` procedures or between the execution of the `DBMS_DST.BEGIN_UPGRADE` and the `DBMS_DST.END_UPGRADE` procedures.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on time zone data files and on how Oracle Database handles daylight saving time, and *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_DST` package

ORA_DST_CONVERT

Syntax



Purpose

`ORA_DST_CONVERT` is useful when you are changing the time zone data file for your database. The function lets you specify error handling for a specified datetime expression.

- For *`datetime_expr`*, specify a datetime expression that resolves to a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values.

- The optional second argument specifies handling of "duplicate time" errors. Specify `0` (false) to suppress the error by returning the source datetime value. This is the default. Specify `1` (true) to allow the database to return the duplicate time error.
- The optional third argument specifies handling of "nonexisting time" errors. Specify `0` (false) to suppress the error by returning the source datetime value. This is the default. Specify `1` (true) to allow the database to return the nonexisting time error.

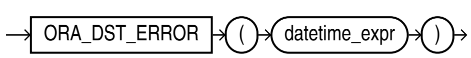
If no error occurs, this function returns a value of the same data type as *`datetime_expr`* (a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values). The returned datetime value when interpreted with the new time zone file corresponds to *`datetime_expr`* interpreted with the old time zone file.

This function can be issued only when changing the time zone data file of the database and upgrading the timestamp with the time zone data, and only between the execution of the `DBMS_DST.BEGIN_UPGRADE` and the `DBMS_DST.END_UPGRADE` procedures.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on time zone data files and on how Oracle Database handles daylight saving time, and *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_DST` package

ORA_DST_ERROR

Syntax



Purpose

`ORA_DST_ERROR` is useful when you are changing the time zone data file for your database. The function takes as an argument a datetime expression that resolves to a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values, and indicates whether the datetime value will result in an error with the new time zone data. The return values are:

- `0`: the datetime value does not result in an error with the new time zone data.
- `1878`: the datetime value results in a "nonexisting time" error.
- `1883`: the datetime value results in a "duplicate time" error.

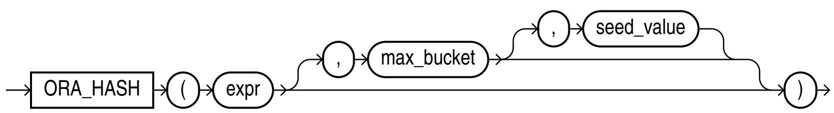
This function can be issued only when changing the time zone data file of the database and upgrading the timestamp with the time zone data, and only between the execution of the `DBMS_DST.BEGIN_PREPARE` and the `DBMS_DST.END_PREPARE` procedures or between the execution of the `DBMS_DST.BEGIN_UPGRADE` and the `DBMS_DST.END_UPGRADE` procedures.

See Also:

Oracle Database Globalization Support Guide for more information on time zone data files and on how Oracle Database handles daylight saving time, and *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_DST` package

ORA_HASH

Syntax



Purpose

`ORA_HASH` is a function that computes a hash value for a given expression. This function is useful for operations such as analyzing a subset of data and generating a random sample.

- The *`expr`* argument determines the data for which you want Oracle Database to compute a hash value. There are no restrictions on the length of data represented by *`expr`*, which commonly resolves to a column name. The *`expr`* cannot be a `LONG` or LOB type. It cannot be a user-defined object type unless it is a nested table type. The hash value for nested table types does not depend on the order of elements in the collection. All other data types are supported for *`expr`*.
- The optional *`max_bucket`* argument determines the maximum bucket value returned by the hash function. You can specify any value between 0 and 4294967295. The default is 4294967295.
- The optional *`seed_value`* argument enables Oracle to produce many different results for the same set of data. Oracle applies the hash function to the combination of *`expr`* and *`seed_value`*. You can specify any value between 0 and 4294967295. The default is 0.

The function returns a `NUMBER` value.

Examples

The following example creates a hash value for each combination of customer ID and product ID in the `sh.sales` table, divides the hash values into a maximum of 100 buckets, and returns the sum of the `amount_sold` values in the first bucket (bucket 0). The third argument (5) provides a seed value for the hash function. You can obtain different hash results for the same query by changing the seed value.

```
SELECT SUM(amount_sold)
  FROM sales
  WHERE ORA_HASH(CONCAT(cust_id, prod_id), 99, 5) = 0;

SUM(AMOUNT_SOLD)
```

```
----------------
       989431.14
```

ORA_INVOKING_USER

Syntax

Purpose

`ORA_INVOKING_USER` returns the name of the database user who invoked the current statement or view. This function takes into account the `BEQUEATH` property of intervening views referenced in the statement. If this function is invoked from within a definer's rights context, then it returns the name of the owner of the definer's rights object. If the invoking user is a Real Application Security user, then it returns user `XS$NULL`.

This function returns a `VARCHAR2` value.

> **See Also:**
>
> - BEQUEATH clause of the `CREATE VIEW` statement
> - *Oracle Database 2 Day + Security Guide* for more information on user `XS$NULL`
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `ORA_INVOKING_USER`

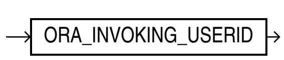
Examples

The following example returns the name of the database user who invoked the statement:

```
SELECT ORA_INVOKING_USER FROM DUAL;
```

ORA_INVOKING_USERID

Syntax



Purpose

`ORA_INVOKING_USERID` returns the identifier of the database user who invoked the current statement or view. This function takes into account the `BEQUEATH` property of intervening views referenced in the statement.

This function returns a `NUMBER` value.

See Also:

- ORA_INVOKING_USER to learn how Oracle Database determines the database user who invoked the current statement or view
- BEQUEATH clause of the `CREATE VIEW` statement

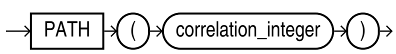
Examples

The following example returns the identifier of the database user who invoked the statement:

```
SELECT ORA_INVOKING_USERID FROM DUAL;
```

PATH

Syntax



Purpose

`PATH` is an ancillary function used only with the `UNDER_PATH` and `EQUALS_PATH` conditions. It returns the relative path that leads to the resource specified in the parent condition.

The *`correlation_integer`* can be any `NUMBER` integer and is used to correlate this ancillary function with its primary condition. Values less than 1 are treated as 1.

See Also:

- EQUALS_PATH Condition and UNDER_PATH Condition
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `PATH`

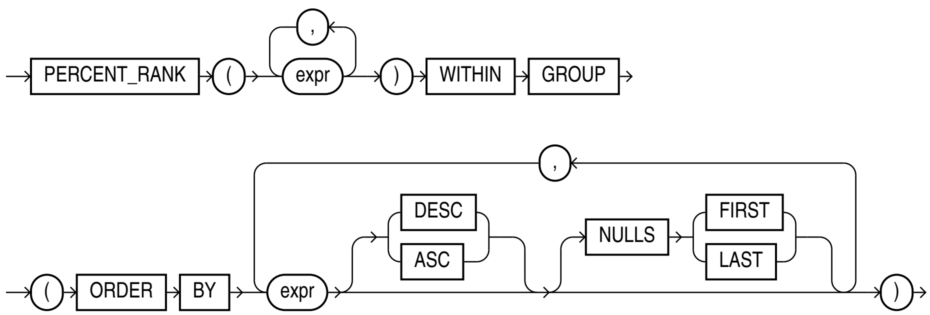
Examples

Refer to the related function DEPTH for an example using both of these ancillary functions of the `EQUALS_PATH` and `UNDER_PATH` conditions.

PERCENT_RANK

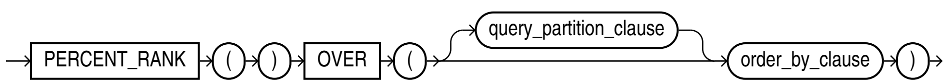
Aggregate Syntax

***percent_rank_aggregate*::=**



Analytic Syntax

***percent_rank_analytic*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`PERCENT_RANK` is similar to the `CUME_DIST` (cumulative distribution) function. The range of values returned by `PERCENT_RANK` is 0 to 1, inclusive. The first row in any set has a `PERCENT_RANK` of 0. The return value is `NUMBER`.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

- As an aggregate function, `PERCENT_RANK` calculates, for a hypothetical row *r* identified by the arguments of the function and a corresponding sort specification, the rank of row *r* minus 1 divided by the number of rows in the aggregate group. This calculation is made as if the hypothetical row *r* were inserted into the group of rows over which Oracle Database is to aggregate.

  The arguments of the function identify a single hypothetical row within each aggregate group. Therefore, they must all evaluate to constant expressions within each aggregate group. The constant argument expressions and the expressions in the `ORDER BY` clause of the aggregate match by position. Therefore the number of arguments must be the same and their types must be compatible.

- As an analytic function, for a row *r*, `PERCENT_RANK` calculates the rank of *r* minus 1, divided by 1 less than the number of rows being evaluated (the entire query result set or a partition).

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `PERCENT_RANK` uses to compare character values for the `ORDER BY` clause

Aggregate Example

The following example calculates the percent rank of a hypothetical employee in the sample table `hr.employees` with a salary of $15,500 and a commission of 5%:

```
SELECT PERCENT_RANK(15000, .05) WITHIN GROUP
       (ORDER BY salary, commission_pct) "Percent-Rank"
  FROM employees;

Percent-Rank
------------
  .971962617
```

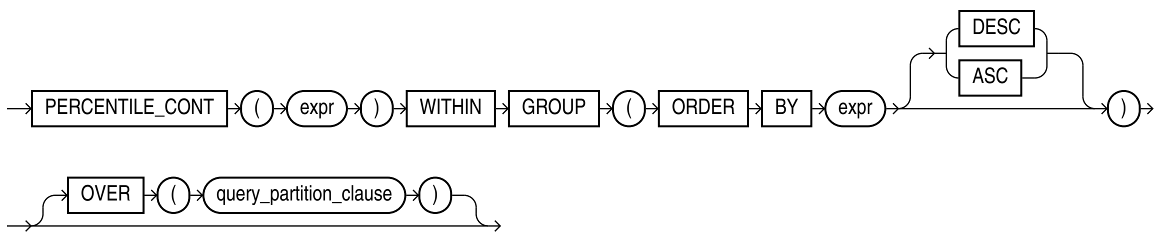
Analytic Example

The following example calculates, for each employee, the percent rank of the employee's salary within the department:

```
SELECT department_id, last_name, salary, PERCENT_RANK()
       OVER (PARTITION BY department_id ORDER BY salary DESC) AS pr
  FROM employees
  ORDER BY pr, salary, last_name;

DEPARTMENT_ID LAST_NAME                     SALARY         PR
------------- ------------------------- ---------- ----------
           10 Whalen                          4400          0
           40 Mavris                          6500          0
              Grant                           7000          0
. . .
           80 Vishney                        10500 .181818182
           80 Zlotkey                        10500 .181818182
           30 Khoo                            3100         .2
. . .
           50 Markle                          2200 .954545455
           50 Philtanker                      2200 .954545455
           50 Olson                           2100          1
. . .
```

PERCENTILE_CONT

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `OVER` clause

Purpose

`PERCENTILE_CONT` is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into that percentile value with respect to the sort specification. Nulls are ignored in the calculation.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

The first *`expr`* must evaluate to a numeric value between 0 and 1, because it is a percentile value. This *`expr`* must be constant within each aggregation group. The `ORDER BY` clause takes a single expression that must be a numeric or datetime value, as these are the types over which Oracle can perform interpolation.

The result of `PERCENTILE_CONT` is computed by linear interpolation between values after ordering them. Using the percentile value (P) and the number of rows (N) in the aggregation group, you can compute the row number you are interested in after ordering the rows with respect to the sort specification. This row number (RN) is computed according to the formula `RN = (1+(P*(N-1))`. The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers `CRN = CEILING(RN)` and `FRN = FLOOR(RN)`.

The final result will be:

```
If (CRN = FRN = RN) then the result is
  (value of expression from row at RN)
Otherwise the result is
  (CRN - RN) * (value of expression for row at FRN) +
  (RN - FRN) * (value of expression for row at CRN)
```

You can use the `PERCENTILE_CONT` function as an analytic function. You can specify only the *`query_partitioning_clause`* in its `OVER` clause. It returns, for each row, the value that would fall into the specified percentile among a set of values within each partition.

The `MEDIAN` function is a specific case of `PERCENTILE_CONT` where the percentile value defaults to 0.5. For more information, refer to MEDIAN .

> **Note:**
>
> Before processing a large amount of data with the `PERCENTILE_CONT` function, consider using one of the following methods to obtain approximate results more quickly than exact results:
>
> - Set the `APPROX_FOR_PERCENTILE` initialization parameter to `PERCENTILE_CONT` or `ALL` before using the `PERCENTILE_CONT` function. Refer to *Oracle Database Reference* for more information on this parameter.
> - Use the `APPROX_PERCENTILE` function instead of the `PERCENTILE_CONT` function. Refer to APPROX_PERCENTILE.

Aggregate Example

The following example computes the median salary in each department:

```
SELECT department_id,
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary DESC) "Median cont",
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY salary DESC) "Median disc"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

DEPARTMENT_ID Median cont Median disc
------------- ----------- -----------
           10        4400        4400
           20        9500       13000
           30        2850        2900
           40        6500        6500
           50        3100        3100
           60        4800        4800
           70       10000       10000
           80        8900        9000
           90       17000       17000
          100        8000        8200
          110       10154       12008
                     7000        7000
```

`PERCENTILE_CONT` and `PERCENTILE_DISC` may return different results. `PERCENTILE_CONT` returns a computed result after doing linear interpolation. `PERCENTILE_DISC` simply returns a value from the set of values that are aggregated over. When the percentile value is 0.5, as in this example, `PERCENTILE_CONT` returns the average of the two middle values for groups with even number of elements, whereas `PERCENTILE_DISC` returns the value of the first one among the two middle values. For aggregate groups with an odd number of elements, both functions return the value of the middle element.

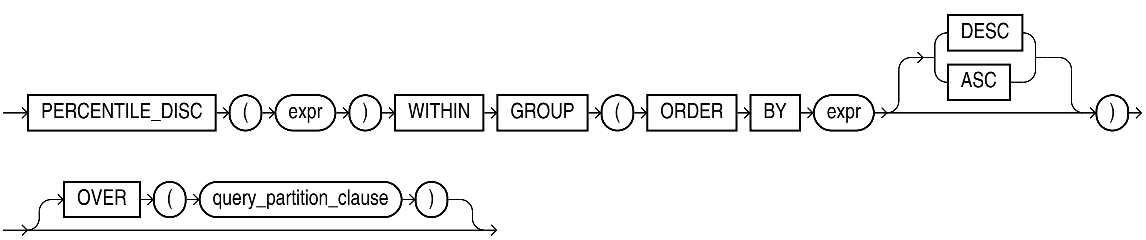
Analytic Example

In the following example, the median for Department 60 is 4800, which has a corresponding percentile (`Percent_Rank`) of 0.5. None of the salaries in Department 30 have a percentile of 0.5, so the median value must be interpolated between 2900 (percentile 0.4) and 2800 (percentile 0.6), which evaluates to 2850.

```
SELECT last_name, salary, department_id,
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary DESC)
         OVER (PARTITION BY department_id) "Percentile_Cont",
       PERCENT_RANK()
        OVER (PARTITION BY department_id ORDER BY salary DESC) "Percent_Rank"
  FROM employees
  WHERE department_id IN (30, 60)
  ORDER BY last_name, salary, department_id;

LAST_NAME                     SALARY DEPARTMENT_ID Percentile_Cont Percent_Rank
------------------------- ---------- ------------- --------------- ------------
Austin                          4800            60            4800           .5
Baida                           2900            30            2850           .4
Colmenares                      2500            30            2850            1
Ernst                           6000            60            4800          .25
Himuro                          2600            30            2850           .8
Hunold                          9000            60            4800            0
Khoo                            3100            30            2850           .2
Lorentz                         4200            60            4800            1
Pataballa                       4800            60            4800           .5
Raphaely                       11000            30            2850            0
Tobias                          2800            30            2850           .6
```

PERCENTILE_DISC

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `OVER` clause

Purpose

PERCENTILE_DISC is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the set. Nulls are ignored in the calculation.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

The first *expr* must evaluate to a numeric value between 0 and 1, because it is a percentile value. This expression must be constant within each aggregate group. The ORDER BY clause takes a single expression that can be of any type that can be sorted.

For a given percentile value P, PERCENTILE_DISC sorts the values of the expression in the ORDER BY clause and returns the value with the smallest CUME_DIST value (with respect to the same sort specification) that is greater than or equal to P.

> **Note:**
>
> Before processing a large amount of data with the PERCENTILE_DISC function, consider using one of the following methods to obtain approximate results more quickly than exact results:
>
> - Set the APPROX_FOR_PERCENTILE initialization parameter to PERCENTILE_DISC or ALL before using the PERCENTILE_DISC function. Refer to *Oracle Database Reference* for more information on this parameter.
> - Use the APPROX_PERCENTILE function instead of the PERCENTILE_DISC function. Refer to APPROX_PERCENTILE.

Aggregate Example

See aggregate example for PERCENTILE_CONT .

Analytic Example

The following example calculates the median discrete percentile of the salary of each employee in the sample table hr.employees:

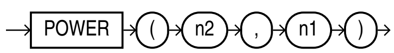
```
SELECT last_name, salary, department_id,
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY salary DESC)
         OVER (PARTITION BY department_id) "Percentile_Disc",
       CUME_DIST() OVER (PARTITION BY department_id
         ORDER BY salary DESC) "Cume_Dist"
  FROM employees
  WHERE department_id in (30, 60)
```

```
  ORDER BY last_name, salary, department_id;

LAST_NAME                     SALARY DEPARTMENT_ID Percentile_Disc  Cume_Dist
------------------------- ---------- ------------- --------------- ----------
Austin                          4800            60            4800         .8
Baida                           2900            30            2900         .5
Colmenares                      2500            30            2900          1
Ernst                           6000            60            4800         .4
Himuro                          2600            30            2900 .833333333
Hunold                          9000            60            4800         .2
Khoo                            3100            30            2900 .333333333
Lorentz                         4200            60            4800          1
Pataballa                       4800            60            4800         .8
Raphaely                       11000            30            2900 .166666667
Tobias                          2800            30            2900 .666666667
```

The median value for Department 30 is 2900, which is the value whose corresponding percentile (`Cume_Dist`) is the smallest value greater than or equal to 0.5. The median value for Department 60 is 4800, which is the value whose corresponding percentile is the smallest value greater than or equal to 0.5.

POWER

Syntax



Purpose

`POWER` returns *`n2`* raised to the *`n1`* power. The base *`n2`* and the exponent *`n1`* can be any numbers, but if *`n2`* is negative, then *`n1`* must be an integer.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If any argument is `BINARY_FLOAT` or `BINARY_DOUBLE`, then the function returns `BINARY_DOUBLE`. Otherwise, the function returns `NUMBER`.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

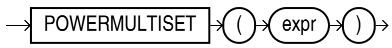
Examples

The following example returns 3 squared:

```
SELECT POWER(3,2) "Raised"
  FROM DUAL;

    Raised
----------
         9
```

POWERMULTISET

Syntax



Purpose

`POWERMULTISET` takes as input a nested table and returns a nested table of nested tables containing all nonempty subsets (called submultisets) of the input nested table.

- *`expr`* can be any expression that evaluates to a nested table.
- If *`expr`* resolves to null, then Oracle Database returns `NULL`.
- If *`expr`* resolves to a nested table that is empty, then Oracle returns an error.
- The element types of the nested table must be comparable. Refer to "Comparison Conditions " for information on the comparability of nonscalar types.

> **Note:**
>
> This function is not supported in PL/SQL.

Examples

First, create a data type that is a nested table of the `cust_address_tab_type` data type:

```
CREATE TYPE cust_address_tab_tab_typ
  AS TABLE OF cust_address_tab_typ;
/
```

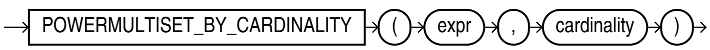
Now, select the nested table column `cust_address_ntab` from the `customers_demo` table using the `POWERMULTISET` function:

```
SELECT CAST(POWERMULTISET(cust_address_ntab) AS cust_address_tab_tab_typ)
  FROM customers_demo;

CAST(POWERMULTISET(CUST_ADDRESS_NTAB) AS CUST_ADDRESS_TAB_TAB_TYP)
  (STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
------------------------------------------------------------------
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('514 W Superior St', '46901', 'Kokomo', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US')))
. . .
```

The preceding example requires the `customers_demo` table and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table columns.

POWERMULTISET_BY_CARDINALITY

Syntax



Purpose

`POWERMULTISET_BY_CARDINALITY` takes as input a nested table and a cardinality and returns a nested table of nested tables containing all nonempty subsets (called submultisets) of the nested table of the specified cardinality.

- *`expr`* can be any expression that evaluates to a nested table.
- *`cardinality`* can be any positive integer.
- If *`expr`* resolves to null, then Oracle Database returns `NULL`.
- If *`expr`* resolves to a nested table that is empty, then Oracle returns an error.
- The element types of the nested table must be comparable. Refer to "Comparison Conditions " for information on the comparability of nonscalar types.

> **Note:**
>
> This function is not supported in PL/SQL.

Examples

First, create a data type that is a nested table of the `cust_address_tab_type` data type:

```
CREATE TYPE cust_address_tab_tab_typ
  AS TABLE OF cust_address_tab_typ;
/
```

Next, duplicate the elements in all the nested table rows to increase the cardinality of the nested table rows to 2:

```
UPDATE customers_demo
  SET cust_address_ntab = cust_address_ntab MULTISET UNION cust_address_ntab;
```

Now, select the nested table column `cust_address_ntab` from the `customers_demo` table using the `POWERMULTISET_BY_CARDINALITY` function:

```
SELECT CAST(POWERMULTISET_BY_CARDINALITY(cust_address_ntab, 2)
         AS cust_address_tab_tab_typ)
  FROM customers_demo;

CAST(POWERMULTISET_BY_CARDINALITY(CUST_ADDRESS_NTAB,2) AS CUST_ADDRESS_TAB_TAB_TYP)
  (STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
------------------------------------------------------------------------------------
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
```

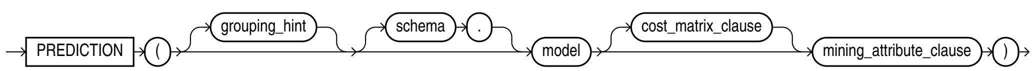
```
  (CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'),
   CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
  (CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'),
   CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
  (CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'),
   CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US')))
. . .
```

The preceding example requires the `customers_demo` table and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table columns.

PREDICTION

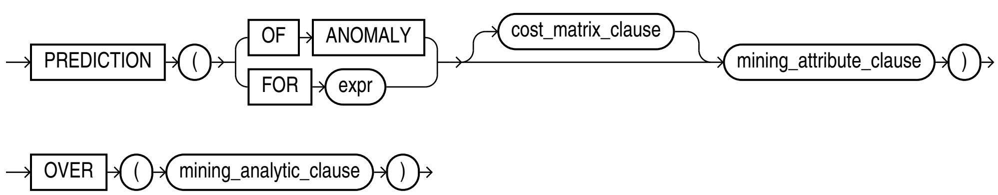
Syntax

prediction::=

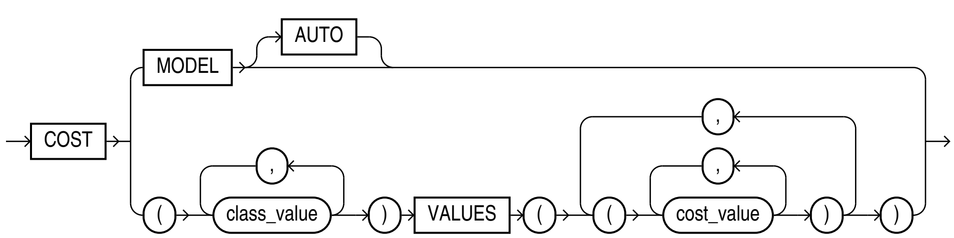


Analytic Syntax

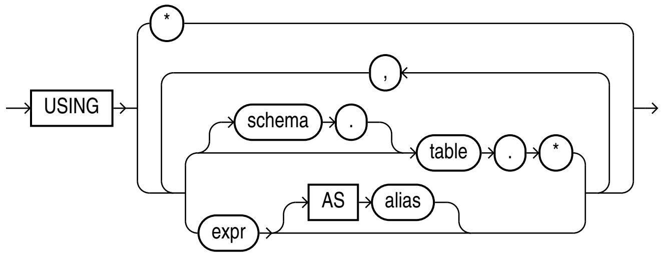
prediction_analytic::=



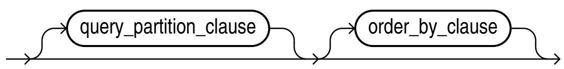
cost_matrix_clause::=



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



See Also:

"Analytic Functions " for information on the syntax, semantics, and restrictions of *`mining_analytic_clause`*

Purpose

`PREDICTION` returns a prediction for each row in the selection. The data type of the returned prediction depends on whether the function performs Regression, Classification, or Anomaly Detection.

- **Regression**: Returns the expected target value for each row. The data type of the return value is the data type of the target.
- **Classification**: Returns the most probable target class (or lowest cost target class, if costs are specified) for each row. The data type of the return value is the data type of the target.
- **Anomaly Detection**: Returns `1` or `0` for each row. Typical rows are classified as 1. Rows that differ significantly from the rest of the data are classified as 0.

cost_matrix_clause

Costs are a biasing factor for minimizing the most harmful kinds of misclassifications. You can specify *`cost_matrix_clause`* for Classification or Anomaly Detection. Costs are not relevant for Regression. The *`cost_matrix_clause`* behaves as described for "PREDICTION_COST ".

Syntax Choice

`PREDICTION` can score data in one of two ways: It can apply a mining model object to the data, or it can dynamically score the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax**: Use this syntax to score the data with a pre-defined model. Supply the name of a model that performs Classification, Regression, or Anomaly Detection.
- **Analytic Syntax**: Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *`mining_analytic_clause`* , which specifies if the data should be partitioned for multiple model builds. The *`mining_analytic_clause`* supports a *`query_partition_clause`* and an *`order_by_clause`*. (See "*analytic_clause*::=".)
  - For Regression, specify `FOR` *`expr`*, where *`expr`* is an expression that identifies a target column that has a numeric data type.
  - For Classification, specify `FOR` *`expr`*, where *`expr`* is an expression that identifies a target column that has a character data type.
  - For Anomaly Detection, specify the keywords `OF ANOMALY`.

The syntax of the `PREDICTION` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring.

- If you specify `USING *`, all the relevant attributes present in the input row are used.
- If you invoke the function with the analytic syntax, the *`mining_attribute_clause`* is used both for building the transient models and for scoring.
- It you invoke the function with a pre-defined model, the *`mining_attribute_clause`* should include all or some of the attributes that were used to create the model. The following conditions apply:
  - If *`mining_attribute_clause`* includes an attribute with the same name but a different data type from the one that was used to create the model, then the data type is converted to the type expected by the model.
  - If you specify more attributes for scoring than were used to create the model, then the extra attributes are silently ignored.
  - If you specify fewer attributes for scoring than were used to create the model, then scoring is performed on a best-effort basis.

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about predictive data mining.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PREDICTION` when it is a character value

> **Note:**
>
> The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

In this example, the model `dt_sh_clas_sample` predicts the gender and age of customers who are most likely to use an affinity card (target = 1). The `PREDICTION` function takes into account the cost matrix associated with the model and uses marital status, education, and household size as predictors.

```
SELECT cust_gender, COUNT(*) AS cnt, ROUND(AVG(age)) AS avg_age
   FROM mining_data_apply_v
   WHERE PREDICTION(dt_sh_clas_sample COST MODEL
      USING cust_marital_status, education, household_size) = 1
   GROUP BY cust_gender
   ORDER BY cust_gender;

CUST_GENDER         CNT    AVG_AGE
------------ ---------- ----------
F                   170         38
M                   685         42
```

The cost matrix associated with the model `dt_sh_clas_sample` is stored in the table `dt_sh_sample_costs`. The cost matrix specifies that the misclassification of 1 is 8 times more costly than the misclassification of 0.

```
SQL> select * from dt_sh_sample_cost;

ACTUAL_TARGET_VALUE PREDICTED_TARGET_VALUE         COST
------------------- ---------------------- ------------
                  0                      0   .000000000
                  0                      1  1.000000000
                  1                      0  8.000000000
                  1                      1   .000000000
```

Analytic Example

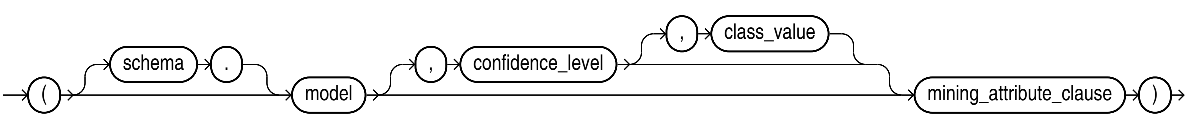
In this example, dynamic regression is used to predict the age of customers who are likely to use an affinity card. The query returns the 3 customers whose predicted age is most different from the actual. The query includes information about the predictors that have the greatest influence on the prediction.

```
SELECT cust_id, age, pred_age, age-pred_age age_diff, pred_det FROM
   (SELECT cust_id, age, pred_age, pred_det,
          RANK() OVER (ORDER BY ABS(age-pred_age) desc) rnk FROM
   (SELECT cust_id, age,
           PREDICTION(FOR age USING *) OVER () pred_age,
           PREDICTION_DETAILS(FOR age ABS USING *) OVER () pred_det
    FROM mining_data_apply_v))
  WHERE rnk <= 3;

CUST_ID  AGE PRED_AGE AGE_DIFF PRED_DET
------- ---- -------- -------- ----------------------------------------------------------
 100910   80    40.67    39.33 <Details algorithm="Support Vector Machines">
```

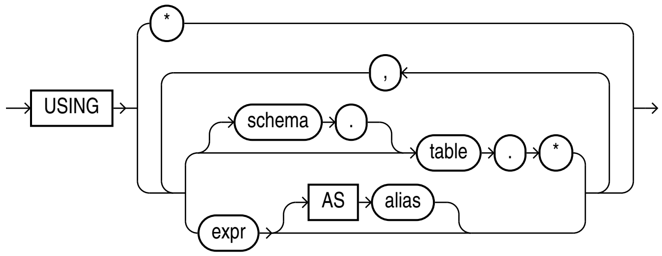
```
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1"
weight=".059"
                                rank="1"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="2"/>
                               <Attribute name="AFFINITY_CARD" actualValue="0" weight=".059"
                                rank="3"/>
                               <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".059"
                                rank="4"/>
                               <Attribute name="YRS_RESIDENCE" actualValue="4" weight=".059"
                                rank="5"/>
                               </Details>

 101285    79   42.18    36.82 <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1"
weight=".059"
                                rank="1"/>
                               <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".059"
                                rank="2"/>
                               <Attribute name="CUST_MARITAL_STATUS" actualValue="Mabsent"
                                weight=".059" rank="3"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="4"/>
                               <Attribute name="OCCUPATION" actualValue="Prof." weight=".059"
                                rank="5"/>
                               </Details>

 100694     77  41.04    35.96 <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1"
weight=".059"
                                rank="1"/>
                               <Attribute name="EDUCATION" actualValue="&lt; Bach." weight=".059"
                                rank="2"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="3"/>
                               <Attribute name="CUST_ID" actualValue="100694" weight=".059"
                                rank="4"/>
                               <Attribute name="COUNTRY_NAME" actualValue="United States of
                                America" weight=".059" rank="5"/>
                               </Details>
```

PREDICTION_BOUNDS

Syntax



***mining_attribute_clause*::=**



Purpose

PREDICTION_BOUNDS applies a Generalized Linear Model (GLM) to predict a class or a value for each row in the selection. The function returns the upper and lower bounds of each prediction in a varray of objects with fields UPPER and LOWER.

GLM can perform either regression or binary classification:

- The bounds for regression refer to the predicted target value. The data type of UPPER and LOWER is the data type of the target.
- The bounds for binary classification refer to the probability of either the predicted target class or the specified *class_value*. The data type of UPPER and LOWER is BINARY_DOUBLE.

If the model was built using ridge regression, or if the covariance matrix is found to be singular during the build, then PREDICTION_BOUNDS returns NULL for both bounds.

confidence_level is a number in the range (0,1). The default value is 0.95. You can specify *class_value* while leaving *confidence_level* at its default by specifying NULL for *confidence_level*.

The syntax of the PREDICTION_BOUNDS function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. This clause behaves as described for the PREDICTION function. (Note that the reference to analytic syntax does not apply.) See "*mining_attribute_clause*::=".

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring
> - *Oracle Data Mining Concepts* for information about Generalized Linear Models

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

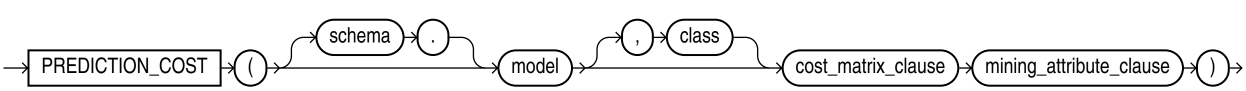
The following example returns the distribution of customers whose ages are predicted with 98% confidence to be greater than 24 and less than 46.

```
SELECT count(cust_id) cust_count, cust_marital_status
  FROM (SELECT cust_id, cust_marital_status
    FROM mining_data_apply_v
    WHERE PREDICTION_BOUNDS(glmr_sh_regr_sample,0.98 USING *).LOWER > 24 AND
          PREDICTION_BOUNDS(glmr_sh_regr_sample,0.98 USING *).UPPER < 46)
    GROUP BY cust_marital_status;

    CUST_COUNT CUST_MARITAL_STATUS
-------------- --------------------
            46 NeverM
             7 Mabsent
             5 Separ.
            35 Divorc.
            72 Married
```

PREDICTION_COST

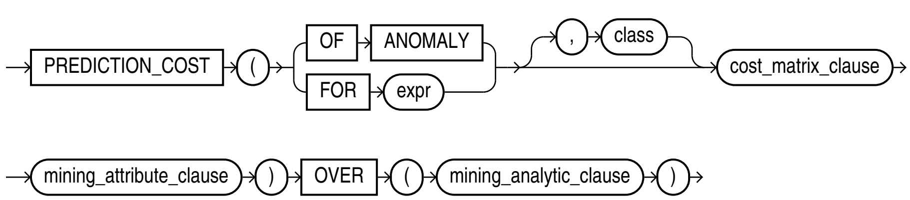
Syntax

***prediction_cost*::=**

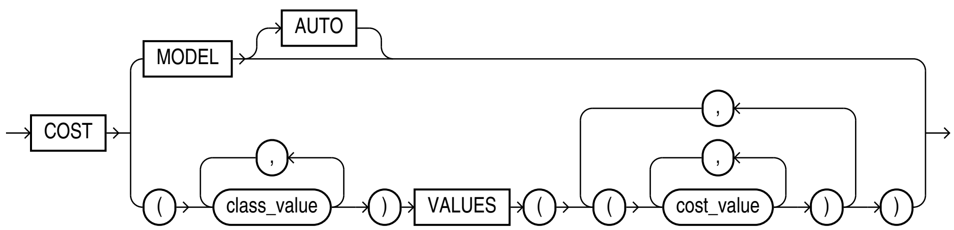


Analytic Syntax

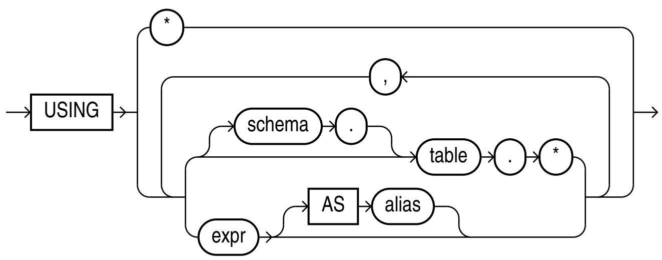
***prediction_cost_analytic*::=**



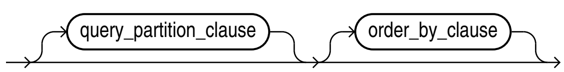
***cost_matrix_clause*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of `mining_analytic_clause`

Purpose

`PREDICTION_COST` returns a cost for each row in the selection. The cost refers to the lowest cost class or to the specified *`class`*. The cost is returned as `BINARY_DOUBLE`.

`PREDICTION_COST` can perform classification or anomaly detection. For classification, the returned cost refers to a predicted target class. For anomaly detection, the returned cost refers to a classification of `1` (for typical rows) or `0` (for anomalous rows).

You can use `PREDICTION_COST` in conjunction with the `PREDICTION` function to obtain the prediction and the cost of the prediction.

cost_matrix_clause

Costs are a biasing factor for minimizing the most harmful kinds of misclassifications. For example, false positives might be considered more costly than false negatives. Costs are specified in a cost matrix that can be associated with the model or defined inline in a VALUES clause. All classification algorithms can use costs to influence scoring.

Decision Tree is the only algorithm that can use costs to influence the model build. The cost matrix used to build a Decision Tree model is also the default scoring cost matrix for the model.

The following cost matrix table specifies that the misclassification of 1 is five times more costly than the misclassification of 0.

```
ACTUAL_TARGET_VALUE PREDICTED_TARGET_VALUE       COST
------------------- ---------------------- ----------
                  0                      0          0
                  0                      1          1
                  1                      0          5
                  1                      1          0
```

In *cost_matrix_clause*:

- COST MODEL indicates that scoring should be performed by taking into account the scoring cost matrix associated with the model. If the cost matrix does not exist, then the function returns an error.
- COST MODEL AUTO indicates that the existence of a cost matrix is unknown. If a cost matrix exists, then the function uses it to return the lowest cost prediction. Otherwise the function returns the highest probability prediction.
- The VALUES clause specifies an inline cost matrix for *class_value*. For example, you could specify that the misclassification of 1 is five times more costly than the misclassification of 0 as follows:

  ```
  PREDICTION (nb_model COST (0,1) VALUES ((0, 1),(1, 5)) USING *)
  ```

  If a model that has a scoring cost matrix is invoked with an inline cost matrix, then the inline costs are used.

> **See Also:**
>
> *Oracle Data Mining User's Guide* for more information about cost-sensitive prediction.

Syntax Choice

PREDICTION_COST can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a model that performs classification or anomaly detection.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause* , which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

- For classification, specify `FOR` *`expr`*, where *`expr`* is an expression that identifies a target column that has a character data type.
- For anomaly detection, specify the keywords `OF ANOMALY`.

The syntax of the `PREDICTION_COST` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about classification with costs

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

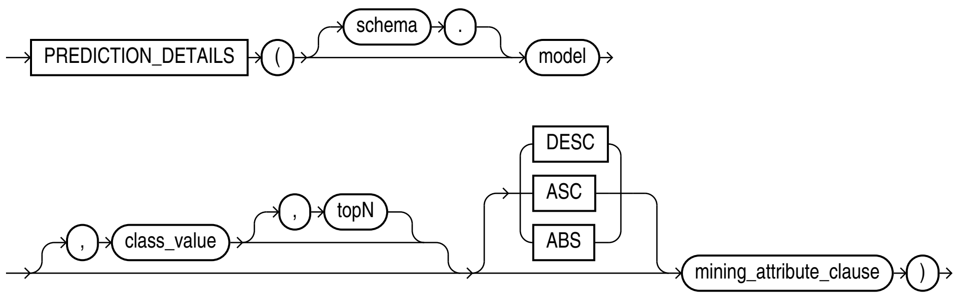
This example predicts the ten customers in Italy who would respond to the least expensive sales campaign (offering an affinity card).

```
SELECT cust_id
FROM (SELECT cust_id,rank()
       OVER (ORDER BY PREDICTION_COST(DT_SH_Clas_sample, 1 COST MODEL USING *)
            ASC, cust_id) rnk
        FROM mining_data_apply_v
        WHERE country_name = 'Italy')
  WHERE rnk <= 10
  ORDER BY rnk;

   CUST_ID
----------
    100081
    100179
    100185
    100324
    100344
    100554
    100662
    100733
    101250
    101306
```

PREDICTION_DETAILS

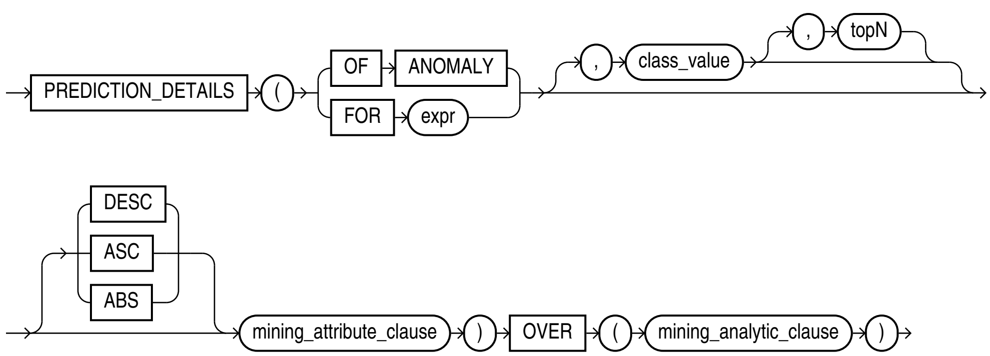
Syntax

***prediction_details*::=**

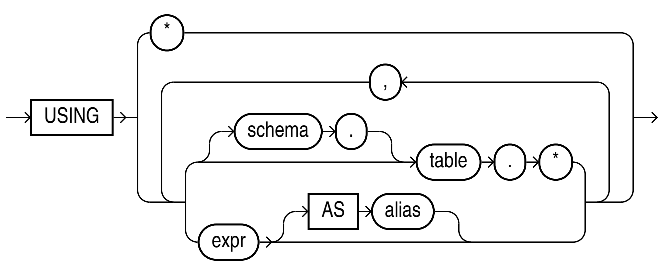


Analytic Syntax

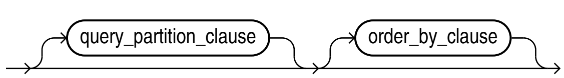
***prediction_details_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

PREDICTION_DETAILS returns prediction details for each row in the selection. The return value is an XML string that describes the attributes of the prediction.

For regression, the returned details refer to the predicted target value. For classification and anomaly detection, the returned details refer to the highest probability class or the specified *class_value*.

topN

If you specify a value for *topN*, the function returns the *N* attributes that have the most influence on the prediction (the score). If you do not specify *topN*, the function returns the 5 most influential attributes.

DESC, ASC, or ABS

The returned attributes are ordered by weight. The weight of an attribute expresses its positive or negative impact on the prediction. For regression, a positive weight indicates a higher value prediction; a negative weight indicates a lower value prediction. For classification and anomaly detection, a positive weight indicates a higher probability prediction; a negative weight indicates a lower probability prediction.

By default, PREDICTION_DETAILS returns the attributes with the highest positive weight (DESC). If you specify ASC, the attributes with the highest negative weight are returned. If you specify ABS, the attributes with the greatest weight, whether negative or positive, are returned. The results are ordered by absolute value from highest to lowest. Attributes with a zero weight are not included in the output.

Syntax Choice

PREDICTION_DETAILS can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a model that performs classification, regression, or anomaly detection.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)
  – For classification, specify FOR *expr*, where *expr* is an expression that identifies a target column that has a character data type.
  – For regression, specify FOR *expr*, where *expr* is an expression that identifies a target column that has a numeric data type.
  – For anomaly detection, specify the keywords OF ANOMALY.

The syntax of the `PREDICTION_DETAILS` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about predictive data mining.

> **Note:**
>
> The following examples are excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

This example uses the model `svmr_sh_regr_sample` to score the data. The query returns the three attributes that have the greatest influence on predicting a higher value for customer age.

```
SELECT PREDICTION_DETAILS(svmr_sh_regr_sample, null, 3 USING *) prediction_details
    FROM mining_data_apply_v
    WHERE cust_id = 100001;

PREDICTION_DETAILS
---------------------------------------------------------------------------------------
<Details algorithm="Support Vector Machines">
<Attribute name="CUST_MARITAL_STATUS" actualValue="Widowed" weight=".361" rank="1"/>
<Attribute name="CUST_GENDER" actualValue="F" weight=".14" rank="2"/>
<Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".135" rank="3"/>
</Details>
```

Analytic Syntax

This example dynamically identifies customers whose age is not typical for the data. The query returns the attributes that predict or detract from a typical age.

```
SELECT cust_id, age, pred_age, age-pred_age age_diff, pred_det
    FROM (SELECT cust_id, age, pred_age, pred_det,
          RANK() OVER (ORDER BY ABS(age-pred_age) DESC) rnk
          FROM (SELECT cust_id, age,
             PREDICTION(FOR age USING *) OVER () pred_age,
             PREDICTION_DETAILS(FOR age ABS USING *) OVER () pred_det
             FROM mining_data_apply_v))
```

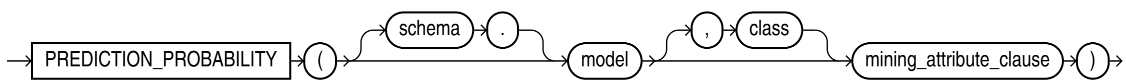
```
    WHERE rnk <= 5;

CUST_ID AGE PRED_AGE AGE_DIFF  PRED_DET
------- --- -------- -------- ------------------------------------------------------------------
 100910  80    40.67    39.33 <Details algorithm="Support Vector Machines">
                              <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                               rank="1"/>
                              <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                               rank="2"/>
                              <Attribute name="AFFINITY_CARD" actualValue="0" weight=".059"
                               rank="3"/>
                              <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".059"
                               rank="4"/>
                              <Attribute name="YRS_RESIDENCE" actualValue="4" weight=".059"
                               rank="5"/>
                              </Details>

 101285  79    42.18    36.82  <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="1"/>
                               <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".059"
                                rank="2"/>
                               <Attribute name="CUST_MARITAL_STATUS" actualValue="Mabsent"
                                weight=".059" rank="3"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="4"/>
                               <Attribute name="OCCUPATION" actualValue="Prof." weight=".059"
                                rank="5"/>
                               </Details>

 100694   77    41.04    35.96  <Details algorithm="Support Vector Machines">
                                <Attribute name="HOME_THEATER_PACKAGE" actualValue="1"
                                 weight=".059" rank="1"/>
                                <Attribute name="EDUCATION" actualValue="&lt; Bach." weight=".059"
                                 rank="2"/>
                                <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                 rank="3"/>
                                <Attribute name="CUST_ID" actualValue="100694" weight=".059"
                                 rank="4"/>
                                <Attribute name="COUNTRY_NAME" actualValue="United States of
                                 America" weight=".059" rank="5"/>
                                </Details>

 100308  81    45.33    35.67  <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="1"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="2"/>
                               <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".059"
                                rank="3"/>
                               <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".059"
                                rank="4"/>
                               <Attribute name="CUST_GENDER" actualValue="F" weight=".059"
                                rank="5"/>
                               </Details>

 101256  90    54.39    35.61  <Details algorithm="Support Vector Machines">
                               <Attribute name="YRS_RESIDENCE" actualValue="9" weight=".059"
                                rank="1"/>
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="2"/>
```

```
<Attribute name="EDUCATION" actualValue="&lt; Bach." weight=".059"
 rank="3"/>
<Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
 rank="4"/>
<Attribute name="COUNTRY_NAME" actualValue="United States of
 America" weight=".059" rank="5"/>
</Details>
```

PREDICTION_PROBABILITY

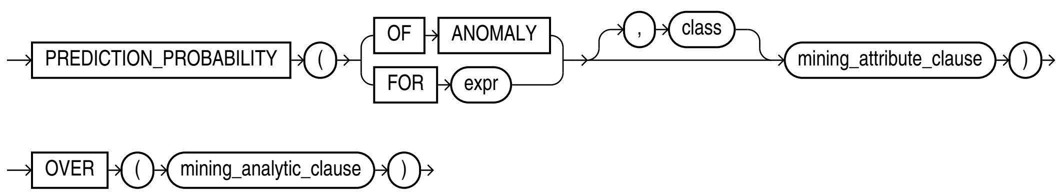
Syntax

***prediction_probability*::=**

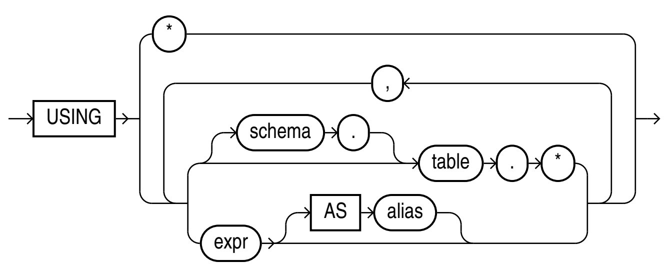


Analytic Syntax

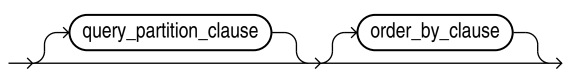
***prediction_prob_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

PREDICTION_PROBABILITY returns a probability for each row in the selection. The probability refers to the highest probability class or to the specified *class*. The data type of the returned probability is BINARY_DOUBLE.

PREDICTION_PROBABILITY can perform classification or anomaly detection. For classification, the returned probability refers to a predicted target class. For anomaly detection, the returned probability refers to a classification of 1 (for typical rows) or 0 (for anomalous rows).

You can use PREDICTION_PROBABILITY in conjunction with the PREDICTION function to obtain the prediction and the probability of the prediction.

Syntax Choice

PREDICTION_PROBABILITY can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a model that performs classification or anomaly detection.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause::=*".)
  - For classification, specify FOR *expr*, where *expr* is an expression that identifies a target column that has a character data type.
  - For anomaly detection, specify the keywords OF ANOMALY.

The syntax of the PREDICTION_PROBABILITY function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See "*mining_attribute_clause::=*".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about predictive data mining.

> **Note:**
>
> The following examples are excerpted from the Data Mining sample programs. For information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

Example

The following example returns the 10 customers living in Italy who are most likely to use an affinity card.

```
SELECT cust_id FROM (
   SELECT cust_id
   FROM mining_data_apply_v
   WHERE country_name = 'Italy'
   ORDER BY PREDICTION_PROBABILITY(DT_SH_Clas_sample, 1 USING *)
      DESC, cust_id)
   WHERE rownum < 11;

   CUST_ID
----------
    100081
    100179
    100185
    100324
    100344
    100554
    100662
    100733
    101250
    101306
```

Analytic Example

This example identifies rows that are most atypical in the data in `mining_data_one_class_v`. Each type of marital status is considered separately so that the most anomalous rows per marital status group are returned.

The query returns three attributes that have the most influence on the determination of anomalous rows. The `PARTITION BY` clause causes separate models to be built and applied for each marital status. Because there is only one record with status Mabsent, no model is created for that partition (and no details are provided).

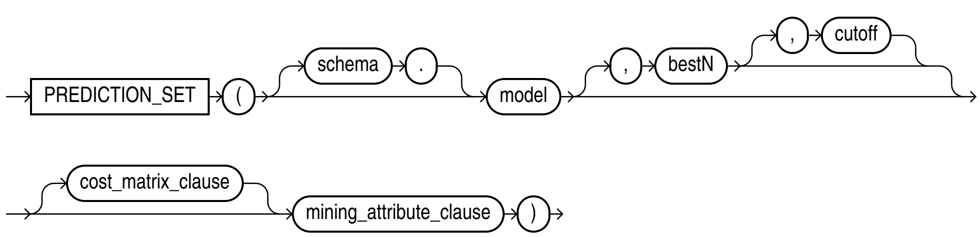
```
SELECT cust_id, cust_marital_status, rank_anom, anom_det FROM
    (SELECT cust_id, cust_marital_status, anom_det,
            rank() OVER (PARTITION BY CUST_MARITAL_STATUS
                         ORDER BY ANOM_PROB DESC,cust_id) rank_anom FROM
     (SELECT cust_id, cust_marital_status,
            PREDICTION_PROBABILITY(OF ANOMALY, 0 USING *)
              OVER (PARTITION BY CUST_MARITAL_STATUS) anom_prob,
            PREDICTION_DETAILS(OF ANOMALY, 0, 3 USING *)
              OVER (PARTITION BY CUST_MARITAL_STATUS) anom_det
     FROM mining_data_one_class_v
    ))
   WHERE rank_anom < 3 order by 2, 3;

CUST_ID CUST_MARITAL_STATUS RANK_ANOM  ANOM_DET
------- ------------------- ----------
```

```
-----------------------------------------------------------
102366   Divorc.             1          <Details algorithm="Support Vector Machines" class="0">
                                        <Attribute name="COUNTRY_NAME" actualValue="United Kingdom"
                                         weight=".069" rank="1"/>
                                        <Attribute name="AGE" actualValue="28" weight=".013"
                                         rank="2"/>
                                        <Attribute name="YRS_RESIDENCE" actualValue="4"
                                         weight=".006" rank="3"/>
                                        </Details>

101817   Divorc.             2          <Details algorithm="Support Vector Machines" class="0">
                                        <Attribute name="YRS_RESIDENCE" actualValue="8"
                                         weight=".018" rank="1"/>
                                        <Attribute name="EDUCATION" actualValue="PhD" weight=".007"
                                         rank="2"/>
                                        <Attribute name="CUST_INCOME_LEVEL" actualValue="K:
                                         250\,000 - 299\,999" weight=".006" rank="3"/>
                                        </Details>

101713   Mabsent             1

101790   Married             1          <Details algorithm="Support Vector Machines" class="0">
                                        <Attribute name="COUNTRY_NAME" actualValue="Canada"
                                         weight=".063" rank="1"/>
                                        <Attribute name="EDUCATION" actualValue="7th-8th"
                                         weight=".011" rank="2"/>
                                        <Attribute name="HOUSEHOLD_SIZE" actualValue="4-5"
                                         weight=".011" rank="3"/>
                                        </Details>
. . .
```

PREDICTION_SET

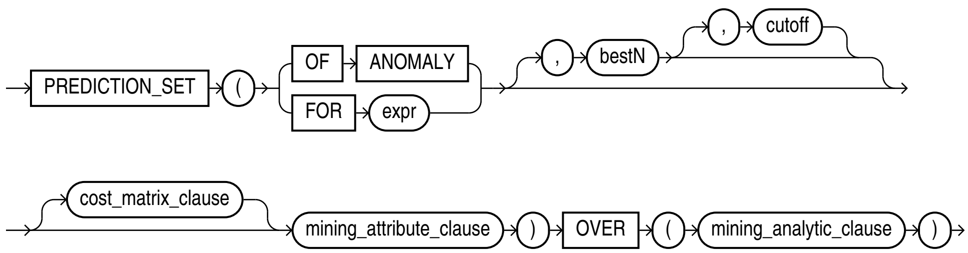
Syntax

***prediction_set*::=**

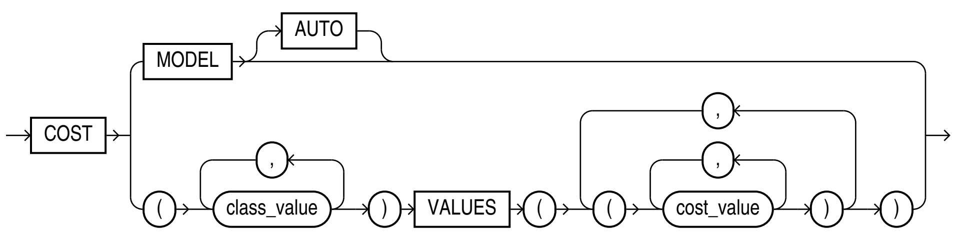


Analytic Syntax

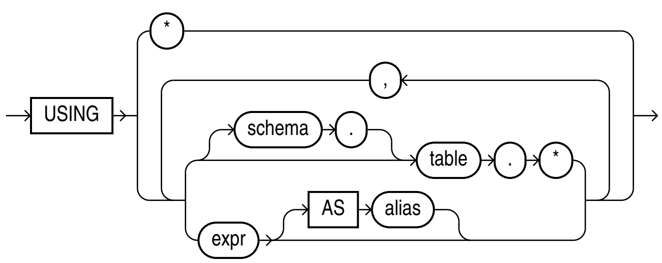
***prediction_set_analytic*::=**



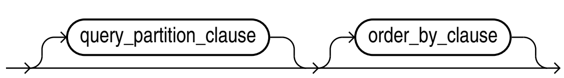
***cost_matrix_clause*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::-**



See Also:

"Analytic Functions " for information on the syntax, semantics, and restrictions of `mining_analytic_clause`

Purpose

PREDICTION_SET returns a set of predictions with either probabilities or costs for each row in the selection. The return value is a varray of objects with field names PREDICTION_ID and PROBABILITY or COST. The prediction identifier is an Oracle NUMBER; the probability and cost fields are BINARY_DOUBLE.

PREDICTION_SET can perform classification or anomaly detection. For classification, the return value refers to a predicted target class. For anomaly detection, the return value refers to a classification of 1 (for typical rows) or 0 (for anomalous rows).

bestN and cutoff

You can specify *bestN* and *cutoff* to limit the number of predictions returned by the function. By default, both *bestN* and *cutoff* are null and all predictions are returned.

- *bestN* is the *N* predictions that are either the most probable or the least costly. If multiple predictions share the *N*th probability or cost, then the function chooses one of them.
- *cutoff* is a value threshold. Only predictions with probability greater than or equal to *cutoff*, or with cost less than or equal to *cutoff*, are returned. To filter by *cutoff* only, specify NULL for *bestN*. If the function uses a *cost_matrix_clause* with COST MODEL AUTO, then *cutoff* is ignored.

You can specify *bestN* with *cutoff* to return up to the *N* most probable predictions that are greater than or equal to *cutoff*. If costs are used, specify *bestN* with *cutoff* to return up to the *N* least costly predictions that are less than or equal to cutoff.

cost_matrix_clause

You can specify *cost_matrix_clause* as a biasing factor for minimizing the most harmful kinds of misclassifications. *cost_matrix_clause* behaves as described for "PREDICTION_COST ".

Syntax Choice

PREDICTION_SET can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a model that performs classification or anomaly detection.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)
  - For classification, specify FOR *expr*, where *expr* is an expression that identifies a target column that has a character data type.
  - For anomaly detection, specify the keywords OF ANOMALY.

The syntax of the PREDICTION_SET function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The `mining_attribute_clause` behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Data Mining User's Guide* for information about scoring.
> - *Oracle Data Mining Concepts* for information about predictive data mining.

> **Note:**
>
> The following example is excerpted from the Data Mining sample programs. For more information about the sample programs, see Appendix A in *Oracle Data Mining User's Guide*.

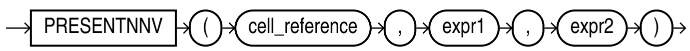
Example

This example lists the probability and cost that customers with ID less than 100006 will use an affinity card. This example has a binary target, but such a query is also useful for multiclass classification such as low, medium, and high.

```
SELECT T.cust_id, S.prediction, S.probability, S.cost
  FROM (SELECT cust_id,
               PREDICTION_SET(dt_sh_clas_sample COST MODEL USING *) pset
          FROM mining_data_apply_v
         WHERE cust_id < 100006) T,
       TABLE(T.pset) S
ORDER BY cust_id, S.prediction;

   CUST_ID PREDICTION  PROBABILITY         COST
---------- ---------- ------------ ------------
    100001          0   .966183575   .270531401
    100001          1   .033816425   .966183575
    100002          0   .740384615  2.076923077
    100002          1   .259615385   .740384615
    100003          0   .909090909   .727272727
    100003          1   .090909091   .909090909
    100004          0   .909090909   .727272727
    100004          1   .090909091   .909090909
    100005          0   .272357724  5.821138211
    100005          1   .727642276   .272357724
```

PRESENTNNV

Syntax



Purpose

The `PRESENTNNV` function can be used only in the *`model_clause`* of the `SELECT` statement and then only on the right-hand side of a model rule. It returns *`expr1`* when *`cell_reference`* exists prior to the execution of the *`model_clause`* and is not null when `PRESENTNNV` is evaluated. Otherwise it returns *`expr2`*. This function differs from `NVL2` in that `NVL2` evaluates the data at the time it is executed, rather than evaluating the data as it was prior to the execution of the *`model_clause`*.

> **See Also:**
>
> - *model_clause* and "Model Expressions" for the syntax and semantics
> - NVL2 for comparison
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PRESENTNNV` when it is a character value

Examples

In the following example, if a row containing sales for the Mouse Pad for the year 2002 exists, and the sales value is not null, then the sales value remains unchanged. If the row exists and the sales value is null, then the sales value is set to 10. If the row does not exist, then the row is created with the sales value set to 10.

```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    ( s['Mouse Pad', 2002] =
        PRESENTNNV(s['Mouse Pad', 2002], s['Mouse Pad', 2002], 10)
    )
  ORDER BY country, prod, year;
```

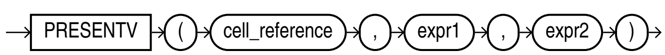
```
COUNTRY       PROD                                      YEAR          S
----------    -----------------------------------   --------  ---------
France        Mouse Pad                                 1998    2509.42
France        Mouse Pad                                 1999    3678.69
France        Mouse Pad                                 2000    3000.72
France        Mouse Pad                                 2001    3269.09
```

```
France          Mouse Pad                                       2002          10
France          Standard Mouse                                  1998     2390.83
France          Standard Mouse                                  1999     2280.45
France          Standard Mouse                                  2000     1274.31
France          Standard Mouse                                  2001     2164.54
Germany         Mouse Pad                                       1998     5827.87
Germany         Mouse Pad                                       1999     8346.44
Germany         Mouse Pad                                       2000     7375.46
Germany         Mouse Pad                                       2001     9535.08
Germany         Mouse Pad                                       2002          10
Germany         Standard Mouse                                  1998     7116.11
Germany         Standard Mouse                                  1999     6263.14
Germany         Standard Mouse                                  2000     2637.31
Germany         Standard Mouse                                  2001     6456.13

18 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to "Examples" to create this view.

PRESENTV

Syntax



Purpose

The `PRESENTV` function can be used only within the *`model_clause`* of the `SELECT` statement and then only on the right-hand side of a model rule. It returns *`expr1`* when, prior to the execution of the *`model_clause`*, *`cell_reference`* exists. Otherwise it returns *`expr2`*.

> **See Also:**
>
> - *model_clause* and "Model Expressions" for the syntax and semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PRESENTV` when it is a character value

Examples

In the following example, if a row containing sales for the Mouse Pad for the year 2000 exists, then the sales value for the Mouse Pad for the year 2001 is set to the sales value for the Mouse Pad for the year 2000. If the row does not exist, then a row is created with the sales value for the Mouse Pad for year 20001 set to 0.

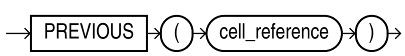
```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
```

```
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s['Mouse Pad', 2001] =
        PRESENTV(s['Mouse Pad', 2000], s['Mouse Pad', 2000], 0)
    )
  ORDER BY country, prod, year;

COUNTRY       PROD                                     YEAR          S
----------    -----------------------------------    --------  ---------
France        Mouse Pad                                1998    2509.42
France        Mouse Pad                                1999    3678.69
France        Mouse Pad                                2000    3000.72
France        Mouse Pad                                2001    3000.72
France        Standard Mouse                           1998    2390.83
France        Standard Mouse                           1999    2280.45
France        Standard Mouse                           2000    1274.31
France        Standard Mouse                           2001    2164.54
Germany       Mouse Pad                                1998    5827.87
Germany       Mouse Pad                                1999    8346.44
Germany       Mouse Pad                                2000    7375.46
Germany       Mouse Pad                                2001    7375.46
Germany       Standard Mouse                           1998    7116.11
Germany       Standard Mouse                           1999    6263.14
Germany       Standard Mouse                           2000    2637.31
Germany       Standard Mouse                           2001    6456.13

16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to "The MODEL clause: Examples" to create this view.

PREVIOUS

Syntax



Purpose

The `PREVIOUS` function can be used only in the *`model_clause`* of the `SELECT` statement and then only in the `ITERATE` ... [`UNTIL`] clause of the *`model_rules_clause`*. It returns the value of *`cell_reference`* at the beginning of each iteration.

> **See Also:**
>
> - *model_clause* and "Model Expressions" for the syntax and semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PREVIOUS` when it is a character value

Examples

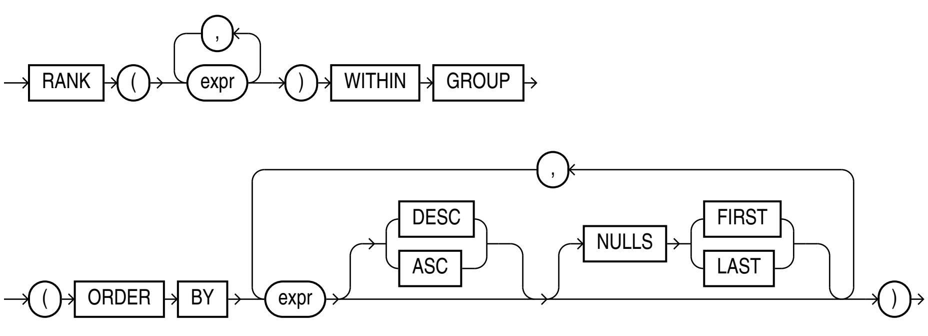
The following example repeats the rules, up to 1000 times, until the difference between the values of `cur_val` at the beginning and at the end of an iteration is less than one:

```
SELECT dim_col, cur_val, num_of_iterations
  FROM (SELECT 1 AS dim_col, 10 AS cur_val FROM dual)
  MODEL
    DIMENSION BY (dim_col)
    MEASURES (cur_val, 0 num_of_iterations)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES ITERATE (1000) UNTIL (PREVIOUS(cur_val[1]) - cur_val[1] < 1)
    (
      cur_val[1] = cur_val[1]/2,
      num_of_iterations[1] = num_of_iterations[1] + 1
    );

   DIM_COL    CUR_VAL NUM_OF_ITERATIONS
---------- ---------- -----------------
         1       .625                 4
```

RANK

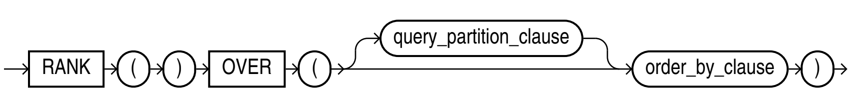
Aggregate Syntax

***rank_aggregate*::=**



Analytic Syntax

***rank_analytic*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

RANK calculates the rank of a value in a group of values. The return type is NUMBER.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Rows with equal values for the ranking criteria receive the same rank. Oracle Database then adds the number of tied rows to the tied rank to calculate the next rank. Therefore, the ranks may not be consecutive numbers. This function is useful for top-N and bottom-N reporting.

- As an aggregate function, RANK calculates the rank of a hypothetical row identified by the arguments of the function with respect to a given sort specification. The arguments of the function must all evaluate to constant expressions within each aggregate group, because they identify a single row within each group. The constant argument expressions and the expressions in the ORDER BY clause of the aggregate match by position. Therefore, the number of arguments must be the same and their types must be compatible.
- As an analytic function, RANK computes the rank of each row returned from a query with respect to the other rows returned by the query, based on the values of the *value_exprs* in the *order_by_clause*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation RANK uses to compare character values for the ORDER BY clause

Aggregate Example

The following example calculates the rank of a hypothetical employee in the sample table hr.employees with a salary of $15,500 and a commission of 5%:

```
SELECT RANK(15500, .05) WITHIN GROUP
   (ORDER BY salary, commission_pct) "Rank"
   FROM employees;

      Rank
----------
       105
```

Similarly, the following query returns the rank for a $15,500 salary among the employee salaries:

```
SELECT RANK(15500) WITHIN GROUP
   (ORDER BY salary DESC) "Rank of 15500"
   FROM employees;

Rank of 15500
--------------
             4
```

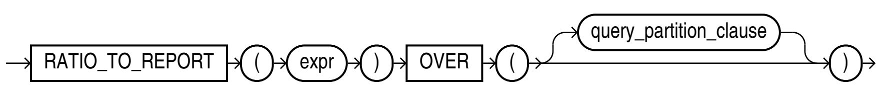
Analytic Example

The following statement ranks the employees in the sample `hr` schema in department 60 based on their salaries. Identical salary values receive the same rank and cause nonconsecutive ranks. Compare this example with the analytic example for DENSE_RANK .

```
SELECT department_id, last_name, salary,
       RANK() OVER (PARTITION BY department_id ORDER BY salary) RANK
  FROM employees WHERE department_id = 60
  ORDER BY RANK, last_name;

DEPARTMENT_ID LAST_NAME                     SALARY       RANK
------------- ------------------------- ---------- ----------
           60 Lorentz                         4200          1
           60 Austin                          4800          2
           60 Pataballa                       4800          2
           60 Ernst                           6000          4
           60 Hunold                          9000          5
```

RATIO_TO_REPORT

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *`expr`*

Purpose

`RATIO_TO_REPORT` is an analytic function. It computes the ratio of a value to the sum of a set of values. If *`expr`* evaluates to null, then the ratio-to-report value also evaluates to null.

The set of values is determined by the *`query_partition_clause`*. If you omit that clause, then the ratio-to-report is computed over all rows returned by the query.

You cannot nest analytic functions by using `RATIO_TO_REPORT` or any other analytic function for *`expr`*. However, you can use other built-in function expressions for *`expr`*. Refer to "About SQL Expressions " for information on valid forms of *`expr`*.

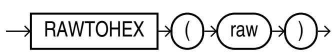
Examples

The following example calculates the ratio-to-report value of each purchasing clerk's salary to the total of all purchasing clerks' salaries:

```
SELECT last_name, salary, RATIO_TO_REPORT(salary) OVER () AS rr
   FROM employees
   WHERE job_id = 'PU_CLERK'
   ORDER BY last_name, salary, rr;

LAST_NAME                     SALARY         RR
------------------------- ---------- ----------
Baida                           2900 .208633094
Colmenares                      2500 .179856115
Himuro                          2600  .18705036
Khoo                            3100 .223021583
Tobias                          2800 .201438849
```

RAWTOHEX

Syntax



Purpose

`RAWTOHEX` converts *`raw`* to a character value containing its hexadecimal representation.

As a SQL built-in function, `RAWTOHEX` accepts an argument of any scalar data type other than `LONG`, `LONG RAW`, `CLOB`, `NCLOB`, `BLOB`, or `BFILE`. If the argument is of a data type other than `RAW`, then this function converts the argument value, which is represented using some number of data bytes, into a `RAW` value with the same number of data bytes. The data itself is not modified in any way, but the data type is recast to a `RAW` data type.

This function returns a `VARCHAR2` value with the hexadecimal representation of bytes that make up the value of *`raw`*. Each byte is represented by two hexadecimal digits.

> **Note:**
>
> `RAWTOHEX` functions differently when used as a PL/SQL built-in function. Refer to *Oracle Database Development Guide* for more information.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `RAWTOHEX`

Examples

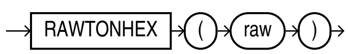
The following hypothetical example returns the hexadecimal equivalent of a `RAW` column value:

```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM graphics;

Graphics
--------
7D
```

> **See Also:**
>
> "RAW and LONG RAW Data Types " and HEXTORAW

RAWTONHEX

Syntax



Purpose

`RAWTONHEX` converts *`raw`* to a character value containing its hexadecimal representation. `RAWTONHEX(`*`raw`*`)` is equivalent to `TO_NCHAR(RAWTOHEX(`*`raw`*`))`. The value returned is always in the national character set.

> **Note:**
>
> `RAWTONHEX` functions differently when used as a PL/SQL built-in function. Refer to *Oracle Database Development Guide* for more information.

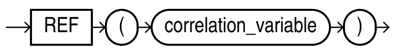
Examples

The following hypothetical example returns the hexadecimal equivalent of a `RAW` column value:

```
SELECT RAWTONHEX(raw_column),
   DUMP ( RAWTONHEX (raw_column) ) "DUMP"
   FROM graphics;

RAWTONHEX(RA)            DUMP
----------------------- ------------------------------
7D                      Typ=1 Len=4: 0,55,0,68
```

REF

Syntax



Purpose

`REF` takes as its argument a correlation variable (table alias) associated with a row of an object table or an object view. A `REF` value is returned for the object instance that is bound to the variable or row.

Examples

The sample schema `oe` contains a type called `cust_address_typ`, described as follows:

```
Attribute                      Type
----------------------------- ----------------
STREET_ADDRESS                     VARCHAR2(40)
POSTAL_CODE                        VARCHAR2(10)
CITY                               VARCHAR2(30)
STATE_PROVINCE                     VARCHAR2(10)
COUNTRY_ID                              CHAR(2)
```

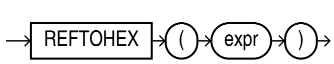
The following example creates a table based on the sample type `oe.cust_address_typ`, inserts a row into the table, and retrieves a `REF` value for the object instance of the type in the addresses table:

```
CREATE TABLE addresses OF cust_address_typ;

INSERT INTO addresses VALUES (
   '123 First Street', '4GF H1J', 'Our Town', 'Ourcounty', 'US');

SELECT REF(e) FROM addresses e;

REF(E)
-----------------------------------------------------------------------------------
00002802097CD1261E51925B60E0340800208254367CD1261E51905B60E034080020825436010101820000
```

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for information on `REF`s

REFTOHEX

Syntax



Purpose

REFTOHEX converts argument *expr* to a character value containing its hexadecimal equivalent. *expr* must return a REF.

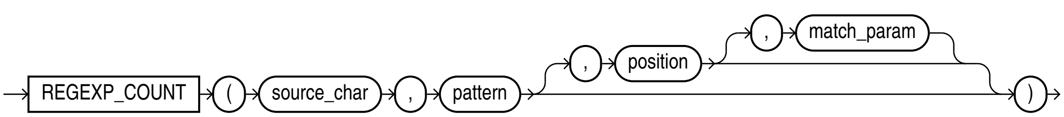
Examples

The sample schema oe contains a warehouse_typ. The following example builds on that type to illustrate how to convert the REF value of a column to a character value containing its hexadecimal equivalent:

```
CREATE TABLE warehouse_table OF warehouse_typ
   (PRIMARY KEY (warehouse_id));

CREATE TABLE location_table
   (location_number NUMBER, building REF warehouse_typ
   SCOPE IS warehouse_table);

INSERT INTO warehouse_table VALUES (1, 'Downtown', 99);

INSERT INTO location_table SELECT 10, REF(w) FROM warehouse_table w;

SELECT REFTOHEX(building) FROM location_table;

REFTOHEX(BUILDING)
--------------------------------------------------------------------------
0000220208859B5E9255C31760E034080020825436859B5E9255C21760E034080020825436
```

REGEXP_COUNT

Syntax



Purpose

REGEXP_COUNT complements the functionality of the REGEXP_INSTR function by returning the number of times a pattern occurs in a source string. The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the number of occurrences of *pattern*. If no match is found, then the function returns 0.

- *source_char* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- *pattern* is the regular expression. It is usually a text literal and can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. It can contain up to 512 bytes. If the data type of *pattern* is different from the data type of *source_char*, then Oracle Database converts *pattern* to the data type of *source_char*.

`REGEXP_COUNT` ignores subexpression parentheses in *`pattern`*. For example, the pattern `'(123(45))'` is equivalent to `'12345'`. For a listing of the operators you can specify in *`pattern`*, refer to Oracle Regular Expression Support.

- *`position`* is a positive integer indicating the character of *`source_char`* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *`source_char`*. After finding the first occurrence of *`pattern`*, the database searches for a second occurrence beginning with the first character following the first occurrence.
- *`match_param`* is a character expression of the data type `VARCHAR2` or `CHAR` that lets you change the default matching behavior of the function.

  The value of *`match_param`* can include one or more of the following characters:

  - `'i'` specifies case-insensitive matching, even if the determined collation of the condition is case-sensitive.
  - `'c'` specifies case-sensitive and accent-sensitive matching, even if the determined collation of the condition is case-insensitive or accent-insensitive.
  - `'n'` allows the period (.), which is the match-any-character character, to match the newline character. If you omit this parameter, then the period does not match the newline character.
  - `'m'` treats the source string as multiple lines. Oracle interprets the caret (`^`) and dollar sign (`$`) as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, then Oracle treats the source string as a single line.
  - `'x'` ignores whitespace characters. By default, whitespace characters match themselves.

  If the value of *`match_param`* contains multiple contradictory characters, then Oracle uses the last character. For example, if you specify `'ic'`, then Oracle uses case-sensitive and accent-sensitive matching. If the value contains a character other than those shown above, then Oracle returns an error.

  If you omit *`match_param`*, then:

  - The default case and accent sensitivity are determined by the determined collation of the `REGEXP_COUNT` function.
  - A period (.) does not match the newline character.
  - The source string is treated as a single line.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `REGEXP_COUNT` uses to compare characters from *`source_char`* with characters from *`pattern`*

Examples

The following example shows that subexpressions parentheses in pattern are ignored:

```
SELECT REGEXP_COUNT('123123123123123', '(12)3', 1, 'i') REGEXP_COUNT
   FROM DUAL;
```

```
REGEXP_COUNT
------------
           5
```

In the following example, the function begins to evaluate the source string at the third character, so skips over the first occurrence of pattern:

```
SELECT REGEXP_COUNT('123123123123', '123', 3, 'i') COUNT FROM DUAL;

     COUNT
----------
         3
```

REGEXP_COUNT simple matching: Examples

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters:

```
select regexp_count('ABC123', '[A-Z]'), regexp_count('A1B2C3', '[A-Z]') from
dual;

REGEXP_COUNT('ABC123','[A-Z]') REGEXP_COUNT('A1B2C3','[A-Z]')
------------------------------ ------------------------------
                 3                     3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number:

```
select regexp_count('ABC123', '[A-Z][0-9]'), regexp_count('A1B2C3', '[A-Z]
[0-9]') from dual;

REGEXP_COUNT('ABC123','[A-Z][0-9]') REGEXP_COUNT('A1B2C3','[A-Z][0-9]')
----------------------------------- -----------------------------------
                  1                       3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number only at the beginning of the string:

```
select regexp_count('ABC123', '[A-Z][0-9]'), regexp_count('A1B2C3', '[A-Z]
[0-9]') from dual;

REGEXP_COUNT('ABC123','^[A-Z][0-9]') REGEXP_COUNT('A1B2C3','^[A-Z][0-9]')
------------------------------------ ------------------------------------
                   0                     1
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by two digits of number only contained within the string:

```
select regexp_count('ABC123', '[A-Z][0-9]{2}'), regexp_count('A1B2C3', '[A-Z]
[0-9]{2}') from dual;

REGEXP_COUNT('ABC123','[A-Z][0-9]{2}') REGEXP_COUNT('A1B2C3','[A-Z][0-9]{2}')
-------------------------------------- --------------------------------------
                     1                         0
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number within the first two occurrences from the beginning of the string:

```
select regexp_count('ABC123', '([A-Z][0-9]){2}'), regexp_count('A1B2C3', '([A-Z][0-9])
{2}') from dual;

REGEXP_COUNT('ABC123','([A-Z][0-9]){2}') REGEXP_COUNT('A1B2C3','([A-Z][0-9]){2}')
---------------------------------------- ----------------------------------------
                                       0                                        1
```

> **Live SQL:**
>
> View and run related examples on Oracle Live SQL at *REGEXP_COUNT simple matching*

REGEXP_COUNT advanced matching: Examples

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters:

```
select regexp_count('ABC123', '[A-Z]') Match_char_ABC_count,
regexp_count('A1B2C3', '[A-Z]') Match_char_ABC_count from dual;

MATCH_CHAR_ABC_COUNT MATCH_CHAR_ABC_COUNT
-------------------- --------------------
           3             3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number:

```
select regexp_count('ABC123', '[A-Z][0-9]') Match_string_C1_count,
regexp_count('A1B2C3', '[A-Z][0-9]')  Match_strings_A1_B2_C3_count from dual;

MATCH_STRING_C1_COUNT MATCH_STRINGS_A1_B2_C3_COUNT
--------------------- ----------------------------
            1                  3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number only at the beginning of the string:

```
select regexp_count('ABC123A5', '^[A-Z][0-9]') Char_num_like_A1_at_start,
regexp_count('A1B2C3', '^[A-Z][0-9]') Char_num_like_A1_at_start from dual;

CHAR_NUM_LIKE_A1_AT_START CHAR_NUM_LIKE_A1_AT_START
------------------------- -------------------------
            0              1
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by two digits of number only contained within the string:

```
select regexp_count('ABC123', '[A-Z][0-9]{2}') Char_num_like_A12_anywhere,
regexp_count('A1B2C34', '[A-Z][0-9]{2}') Char_num_like_A12_anywhere from dual;

CHAR_NUM_LIKE_A12_ANYWHERE CHAR_NUM_LIKE_A12_ANYWHERE
-------------------------- --------------------------
             1                1
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number within the first two occurrences from the beginning of the string:

```
select regexp_count('ABC12D3', '([A-Z][0-9]){2}') Char_num_within_2_places,
regexp_count('A1B2C3', '([A-Z][0-9]){2}') Char_num_within_2_places from dual;

CHAR_NUM_WITHIN_2_PLACES CHAR_NUM_WITHIN_2_PLACES
------------------------ ------------------------
               0             1
```

Live SQL:

View and run related examples on Oracle Live SQL at *REGEXP_COUNT advanced matching*

REGEXP_COUNT case-sensitive matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20));

INSERT INTO regexp_temp (empName) VALUES ('John Doe');
INSERT INTO regexp_temp (empName) VALUES ('Jane Doe');
```

In the following example, the statement queries the employee name column and searches for the lowercase of character ‘E’:

```
SELECT empName, REGEXP_COUNT(empName, 'e', 1, 'c') "CASE_SENSITIVE_E" From
regexp_temp;

EMPNAME          CASE_SENSITIVE_E
-------------------- ----------------
John Doe                1
Jane Doe                2
```

In the following example, the statement queries the employee name column and searches for the lowercase of character ‘O’:

```
SELECT empName, REGEXP_COUNT(empName, 'o', 1, 'c') "CASE_SENSITIVE_O" From
regexp_temp;

EMPNAME          CASE_SENSITIVE_O
-------------------- ----------------
John Doe                2
Jane Doe                1
```

In the following example, the statement queries the employee name column and searches for the lowercase or uppercase of character ‘E’:

```
SELECT empName, REGEXP_COUNT(empName, 'E', 1, 'i') "CASE_INSENSITIVE_E" From
regexp_temp;

EMPNAME          CASE_INSENSITIVE_E
-------------------- ------------------
John Doe                  1
Jane Doe                  2
```

In the following example, the statement queries the employee name column and searches for the lowercase of string 'DO':

```
SELECT empName, REGEXP_COUNT(empName, 'do', 1, 'i') "CASE_INSENSITIVE_STRING" From
regexp_temp;

EMPNAME              CASE_INSENSITIVE_STRING
-------------------- -----------------------
John Doe                       1
Jane Doe                       1
```

In the following example, the statement queries the employee name column and searches for the lowercase or uppercase of string 'AN':

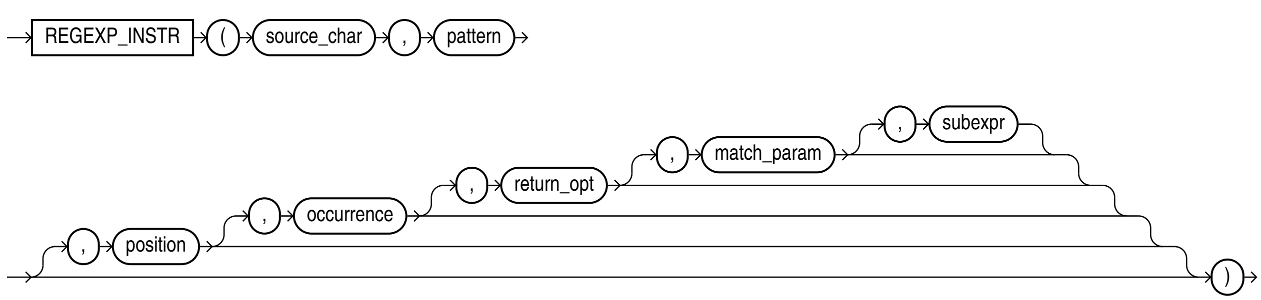
```
SELECT empName, REGEXP_COUNT(empName, 'an', 1, 'c') "CASE_SENSITIVE_STRING" From
regexp_temp;

EMPNAME              CASE_SENSITIVE_STRING
-------------------- ---------------------
John Doe                     0
Jane Doe                     1
```

Live SQL:

View and run related examples on Oracle Live SQL at *REGEXP_COUNT case-sensitive matching*

REGEXP_INSTR

Syntax



Purpose

REGEXP_INSTR extends the functionality of the INSTR function by letting you search a string for a regular expression pattern. The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the beginning or ending position of the matched substring, depending on the value of the *return_option* argument. If no match is found, then the function returns 0.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

- *source_char* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- *pattern* is the regular expression. It is usually a text literal and can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. It can contain up to 512 bytes. If the data type of *pattern* is different from the data type of *source_char*, then Oracle Database converts *pattern* to the data type of *source_char*. For a listing of the operators you can specify in *pattern*, refer to Oracle Regular Expression Support.
- *position* is a positive integer indicating the character of *source_char* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *source_char*.
- *occurrence* is a positive integer indicating which occurrence of *pattern* in *source_char* Oracle should search for. The default is 1, meaning that Oracle searches for the first occurrence of *pattern*. If *occurrence* is greater than 1, then the database searches for the second occurrence beginning with the first character following the first occurrence of *pattern*, and so forth. This behavior is different from the INSTR function, which begins its search for the second occurrence at the second character of the first occurrence.
- *return_option* lets you specify what Oracle should return in relation to the occurrence:
  - If you specify 0, then Oracle returns the position of the first character of the occurrence. This is the default.
  - If you specify 1, then Oracle returns the position of the character following the occurrence.
- *match_param* is a character expression of the data type VARCHAR2 or CHAR that lets you change the default matching behavior of the function. The behavior of this parameter is the same for this function as for REGEXP_COUNT. Refer to REGEXP_COUNT for detailed information.
- For a *pattern* with subexpressions, *subexpr* is an integer from 0 to 9 indicating which subexpression in *pattern* is the target of the function. The *subexpr* is a fragment of pattern enclosed in parentheses. Subexpressions can be nested. Subexpressions are numbered in order in which their left parentheses appear in pattern. For example, consider the following expression:

  ```
  0123(((abc)(de)f)ghi)45(678)
  ```

  This expression has five subexpressions in the following order: "abcdefghi" followed by "abcdef", "abc", "de" and "678".

  If *subexpr* is zero, then the position of the entire substring that matches the *pattern* is returned. If *subexpr* is greater than zero, then the position of the substring fragment that corresponds to subexpression number *subexpr* in the matched substring is returned. If *pattern* does not have at least *subexpr* subexpressions, the function returns zero. A null *subexpr* value returns NULL. The default value for *subexpr* is zero.

See Also:

- INSTR and REGEXP_SUBSTR
- REGEXP_REPLACE and REGEXP_LIKE Condition
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `REGEXP_INSTR` uses to compare characters from *`source_char`* with characters from *`pattern`*

Examples

The following example examines the string, looking for occurrences of one or more non-blank characters. Oracle begins searching at the first character in the string and returns the starting position (default) of the sixth occurrence of one or more non-blank characters.

```
SELECT
  REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA',
               '[^ ]+', 1, 6) "REGEXP_INSTR"
  FROM DUAL;

REGEXP_INSTR
------------
          37
```

The following example examines the string, looking for occurrences of words beginning with `s`, `r`, or `p`, regardless of case, followed by any six alphabetic characters. Oracle begins searching at the third character in the string and returns the position in the string of the character following the second occurrence of a seven-letter word beginning with `s`, `r`, or `p`, regardless of case.

```
SELECT
  REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA',
               '[s|r|p][[:alpha:]]{6}', 3, 2, 1, 'i') "REGEXP_INSTR"
  FROM DUAL;

REGEXP_INSTR
------------
          28
```

The following examples use the *`subexpr`* argument to search for a particular subexpression in *`pattern`*. The first statement returns the position in the source string of the first character in the first subexpression, which is '123':

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))', 1, 1, 0, 'i', 1)
"REGEXP_INSTR" FROM DUAL;

REGEXP_INSTR
-------------------
1
```

The next statement returns the position in the source string of the first character in the second subexpression, which is '45678':

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))', 1, 1, 0, 'i', 2)
"REGEXP_INSTR" FROM DUAL;
```

```
REGEXP_INSTR
-------------------
4
```

The next statement returns the position in the source string of the first character in the fourth subexpression, which is '78':

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))', 1, 1, 0, 'i', 4)
"REGEXP_INSTR" FROM DUAL;

REGEXP_INSTR
-------------------
7
```

REGEXP_INSTR pattern matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20), emailID varchar2(20));

INSERT INTO regexp_temp (empName, emailID) VALUES ('John Doe',
'johndoe@example.com');
INSERT INTO regexp_temp (empName, emailID) VALUES ('Jane Doe', 'janedoe');
```

In the following example, the statement queries the email column and searches for valid email addresses:

```
SELECT emailID, REGEXP_INSTR(emailID, '\w+@\w+(\.\w+)+') "IS_A_VALID_EMAIL" FROM
regexp_temp;

EMAILID          IS_A_VALID_EMAIL
-------------------- ----------------
johndoe@example.com            1
example.com                0
```

In the following example, the statement queries the email column and returns the count of valid email addresses:

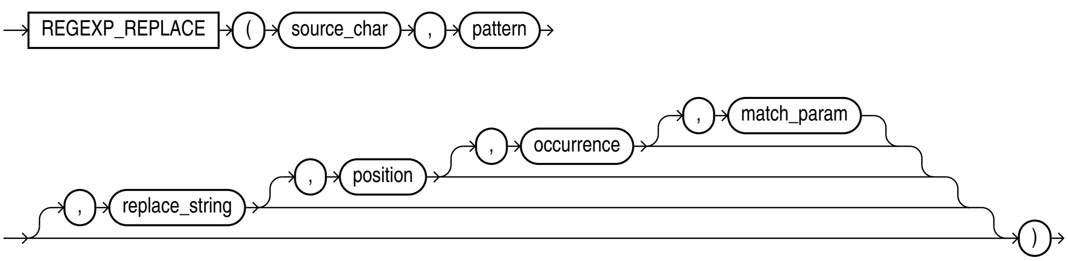
```
EMPNAME        Valid Email            FIELD_WITH_VALID_EMAIL
--------    -------------------    ----------------------
John Doe    johndoe@example.com    1
Jane Doe
```

Live SQL:

View and run related examples on Oracle Live SQL at *REGEXP_INSTR pattern matching*

REGEXP_REPLACE

Syntax



Purpose

REGEXP_REPLACE extends the functionality of the REPLACE function by letting you search a string for a regular expression pattern. By default, the function returns *source_char* with every occurrence of the regular expression pattern replaced with *replace_string*. The string returned is in the same character set as *source_char*. The function returns VARCHAR2 if the first argument is not a LOB and returns CLOB if the first argument is a LOB.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

- *source_char* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB.
- *pattern* is the regular expression. It is usually a text literal and can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. It can contain up to 512 bytes. If the data type of *pattern* is different from the data type of *source_char*, then Oracle Database converts *pattern* to the data type of *source_char*. For a listing of the operators you can specify in *pattern*, refer to Oracle Regular Expression Support.
- *replace_string* can be of any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. If *replace_string* is a CLOB or NCLOB, then Oracle truncates *replace_string* to 32K. The *replace_string* can contain up to 500 backreferences to subexpressions in the form \n, where n is a number from 1 to 9. If you want to include a backslash (\) in *replace_string*, then you must precede it with the escape character, which is also a backslash. For example, to replace \2 you would enter \\2. For more information on backreference expressions, refer to the notes to " Oracle Regular Expression Support", Table D-1.
- *position* is a positive integer indicating the character of *source_char* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *source_char*.
- *occurrence* is a nonnegative integer indicating the occurrence of the replace operation:
  - If you specify 0, then Oracle replaces all occurrences of the match.
  - If you specify a positive integer *n*, then Oracle replaces the *n*th occurrence.

  If *occurrence* is greater than 1, then the database searches for the second occurrence beginning with the first character following the first occurrence of *pattern*, and so forth.

This behavior is different from the `INSTR` function, which begins its search for the second occurrence at the second character of the first occurrence.

- *`match_param`* is a character expression of the data type `VARCHAR2` or `CHAR` that lets you change the default matching behavior of the function. The behavior of this parameter is the same for this function as for `REGEXP_COUNT`. Refer to REGEXP_COUNT for detailed information.

> **See Also:**
>
> - REPLACE
> - REGEXP_INSTR , REGEXP_SUBSTR , and REGEXP_LIKE Condition
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `REGEXP_REPLACE` uses to compare characters from *`source_char`* with characters from *`pattern`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example examines `phone_number`, looking for the pattern *xxx*.*xxx*.*xxxx*. Oracle reformats this pattern with (*xxx*) *xxx*-*xxxx*.

```
SELECT
  REGEXP_REPLACE(phone_number,
                 '([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})',
                 '(\1) \2-\3') "REGEXP_REPLACE"
  FROM employees
  ORDER BY "REGEXP_REPLACE";

REGEXP_REPLACE
--------------------------------------------------------------------------------
(515) 123-4444
(515) 123-4567
(515) 123-4568
(515) 123-4569
(515) 123-5555
. . .
```

The following example examines `country_name`. Oracle puts a space after each non-null character in the string.

```
SELECT
  REGEXP_REPLACE(country_name, '(.)', '\1 ') "REGEXP_REPLACE"
  FROM countries;

REGEXP_REPLACE
--------------------------------------------------------------------------------
A r g e n t i n a
A u s t r a l i a
B e l g i u m
B r a z i l
C a n a d a
. . .
```

The following example examines the string, looking for two or more spaces. Oracle replaces each occurrence of two or more spaces with a single space.

```
SELECT
  REGEXP_REPLACE('500   Oracle     Parkway,    Redwood  Shores, CA',
                 '( ){2,}', ' ') "REGEXP_REPLACE"
  FROM DUAL;

REGEXP_REPLACE
--------------------------------------
500 Oracle Parkway, Redwood Shores, CA
```

REGEXP_REPLACE pattern matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20), emailID varchar2(20));

INSERT INTO regexp_temp (empName, emailID) VALUES ('John Doe', 'johndoe@example.com');
INSERT INTO regexp_temp (empName, emailID) VALUES ('Jane Doe', 'janedoe@example.com');
```

The following statement replaces the string ‘Jane’ with ‘John’:

```
SELECT empName, REGEXP_REPLACE (empName, 'Jane', 'John') "STRING_REPLACE" FROM
regexp_temp;

EMPNAME        STRING_REPLACE
--------    --------------
John Doe    John Doe
Jane Doe    John Doe
```

The following statement replaces the string ‘John’ with ‘Jane’:

```
SELECT empName, REGEXP_REPLACE (empName, 'Jane', 'John') "STRING_REPLACE" FROM
regexp_temp;

EMPNAME        STRING_REPLACE
--------    --------------
John Doe    Jane Doe
Jane Doe    Jane Doe
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_REPLACE - Pattern Matching*

REGEXP_REPLACE: Examples

The following statement replaces all the numbers in a string:

```
WITH strings AS (
  SELECT 'abc123' s FROM dual union all
  SELECT '123abc' s FROM dual union all
  SELECT 'a1b2c3' s FROM dual
)
  SELECT s "STRING", regexp_replace(s, '[0-9]', '') "MODIFIED_STRING"
  FROM strings;
```

```
 STRING               MODIFIED_STRING
-------------------- --------------------
abc123               abc
123abc               abc
a1b2c3               abc
```

The following statement replaces the first numeric occurrence in a string:

```
WITH strings AS (
  SELECT 'abc123' s from DUAL union all
  SELECT '123abc' s from DUAL union all
  SELECT 'a1b2c3' s from DUAL
)
  SELECT s "STRING", REGEXP_REPLACE(s, '[0-9]', '', 1, 1)
"MODIFIED_STRING"
  FROM   strings;

 STRING               MODIFIED_STRING
-------------------- --------------------
abc123               abc23
123abc               23abc
a1b2c3               ab2c3
```

The following statement replaces the second numeric occurrence in a string:

```
WITH strings AS (
  SELECT 'abc123' s from DUAL union all
  SELECT '123abc' s from DUAL union all
  SELECT 'a1b2c3' s from DUAL
)
  SELECT s "STRING", REGEXP_REPLACE(s, '[0-9]', '', 1, 2)
"MODIFIED_STRING"
  FROM   strings;

STRING               MODIFIED_STRING
-------------------- --------------------
abc123               abc13
123abc               13abc
a1b2c3               a1bc3
```

The following statement replaces multiple spaces in a string with a single space:

```
WITH strings AS (
  SELECT 'Hello  World' s FROM dual union all
  SELECT 'Hello        World' s FROM dual union all
  SELECT 'Hello,   World  !' s FROM dual
)
  SELECT s "STRING", regexp_replace(s, ' {2,}', ' ')
"MODIFIED_STRING"
  FROM   strings;

 STRING               MODIFIED_STRING
```

```
-------------------- --------------------
Hello  World         Hello World
Hello        World   Hello World
Hello,   World  !    Hello, World !
```

The following statement converts camel case strings to a string containing lower case words separated by an underscore:

```
WITH strings as (
  SELECT 'AddressLine1' s FROM dual union all
  SELECT 'ZipCode' s FROM dual union all
  SELECT 'Country' s FROM dual
)
  SELECT s "STRING",
         lower(regexp_replace(s, '([A-Z0-9])', '_\1', 2)) "MODIFIED_STRING"
  FROM strings;

  STRING               MODIFIED_STRING
-------------------- --------------------
AddressLine1         address_line_1
ZipCode              zip_code
Country              country
```

The following statement converts the format of a date:

```
WITH date_strings AS (
  SELECT  '2015-01-01' d from dual union all
  SELECT '2000-12-31' d from dual union all
  SELECT '900-01-01' d from dual
)
  SELECT d "STRING",
         regexp_replace(d, '([[:digit:]]+)-([[:digit:]]{2})-([[:digit:]]
{2})', '\3.\2.\1') "MODIFIED_STRING"
  FROM date_strings;

  STRING               MODIFIED_STRING
-------------------- --------------------
2015-01-01           01.01.2015
2000-12-31           31.12.2000
900-01-01            01.01.900
```

The following statement replaces all the letters in a string with ‘1’:

```
WITH strings as (
  SELECT 'NEW YORK' s FROM dual union all
  SELECT 'New York' s FROM dual union all
  SELECT 'new york' s FROM dual
)
  SELECT s "STRING",
        regexp_replace(s, '[a-z]', '1', 1, 0, 'i') "CASE_INSENSITIVE",
        regexp_replace(s, '[a-z]', '1', 1, 0, 'c') "CASE_SENSITIVE",
        regexp_replace(s, '[a-zA-Z]', '1', 1, 0, 'c')
"CASE_SENSITIVE_MATCHING"
```

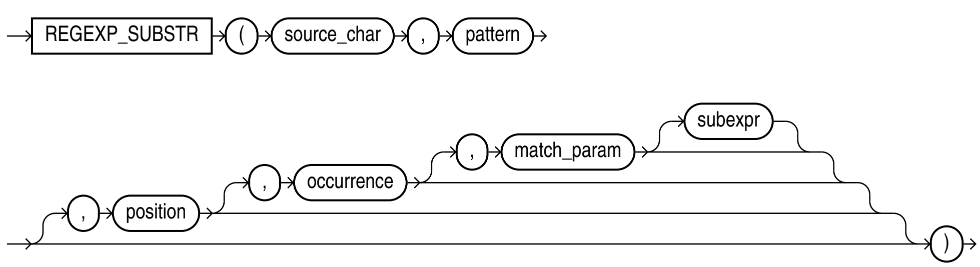
```
  FROM  strings;

  STRING     CASE_INSEN CASE_SENSI CASE_SENSI
---------- ---------- ---------- ----------
NEW YORK   111 1111   NEW YORK   111 1111
New York   111 1111   N11 Y111   111 1111
new york   111 1111   111 1111   111 1111
```

Live SQL:

View and run a related example on Oracle Live SQL at *REGEXP_REPLACE*

REGEXP_SUBSTR

Syntax



Purpose

`REGEXP_SUBSTR` extends the functionality of the `SUBSTR` function by letting you search a string for a regular expression pattern. It is also similar to `REGEXP_INSTR`, but instead of returning the position of the substring, it returns the substring itself. This function is useful if you need the contents of a match string but not its position in the source string. The function returns the string as `VARCHAR2` or `CLOB` data in the same character set as *`source_char`*.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

- *`source_char`* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.
- *`pattern`* is the regular expression. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *`pattern`* is different from the data type of *`source_char`*, then Oracle Database converts *`pattern`* to the data type of *`source_char`*. For a listing of the operators you can specify in *`pattern`*, refer to Oracle Regular Expression Support.

- *position* is a positive integer indicating the character of *source_char* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *source_char*.
- *occurrence* is a positive integer indicating which occurrence of *pattern* in *source_char* Oracle should search for. The default is 1, meaning that Oracle searches for the first occurrence of *pattern*.

  If *occurrence* is greater than 1, then the database searches for the second occurrence beginning with the first character following the first occurrence of *pattern*, and so forth. This behavior is different from the SUBSTR function, which begins its search for the second occurrence at the second character of the first occurrence.
- *match_param* is a character expression of the data type VARCHAR2 or CHAR that lets you change the default matching behavior of the function. The behavior of this parameter is the same for this function as for REGEXP_COUNT. Refer to REGEXP_COUNT for detailed information.
- For a *pattern* with subexpressions, *subexpr* is a nonnegative integer from 0 to 9 indicating which subexpression in *pattern* is to be returned by the function. This parameter has the same semantics that it has for the REGEXP_INSTR function. Refer to REGEXP_INSTR for more information.

> **See Also:**
>
> - SUBSTR and REGEXP_INSTR
> - REGEXP_REPLACE , and REGEXP_LIKE Condition
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation REGEXP_SUBSTR uses to compare characters from *source_char* with characters from *pattern*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example examines the string, looking for the first substring bounded by commas. Oracle Database searches for a comma followed by one or more occurrences of non-comma characters followed by a comma. Oracle returns the substring, including the leading and trailing commas.

```
SELECT
  REGEXP_SUBSTR('500 Oracle Parkway, Redwood Shores, CA',
                ',[^,]+,') "REGEXPR_SUBSTR"
  FROM DUAL;

REGEXPR_SUBSTR
-----------------
, Redwood Shores,
```

The following example examines the string, looking for http:// followed by a substring of one or more alphanumeric characters and optionally, a period (.). Oracle searches for a minimum of three and a maximum of four occurrences of this substring between http:// and either a slash (/) or the end of the string.

```
SELECT
  REGEXP_SUBSTR('http://www.example.com/products',
                'http://([[:alnum:]]+\.?){3,4}/?') "REGEXP_SUBSTR"
  FROM DUAL;

REGEXP_SUBSTR
----------------------
http://www.example.com/
```

The next two examples use the *subexpr* argument to return a specific subexpression of *pattern*. The first statement returns the first subexpression in *pattern*:

```
SELECT REGEXP_SUBSTR('1234567890', '(123)(4(56)(78))', 1, 1, 'i', 1)
"REGEXP_SUBSTR" FROM DUAL;

REGEXP_SUBSTR
-------------------
123
```

The next statement returns the fourth subexpression in *pattern*:

```
SELECT REGEXP_SUBSTR('1234567890', '(123)(4(56)(78))', 1, 1, 'i', 4)
"REGEXP_SUBSTR" FROM DUAL;

REGEXP_SUBSTR
-------------------
78
```

REGEXP_SUBSTR pattern matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20), emailID varchar2(20));

INSERT INTO regexp_temp (empName, emailID) VALUES ('John Doe',
'johndoe@example.com');
INSERT INTO regexp_temp (empName, emailID) VALUES ('Jane Doe', 'janedoe');
```

In the following example, the statement queries the email column and searches for valid email addresses:

```
SELECT empName, REGEXP_SUBSTR(emailID, '[[:alnum:]]+\@[[:alnum:]]+\.[[:alnum:]]
+') "Valid Email" FROM regexp_temp;

EMPNAME  Valid Email
-------- -------------------
John Doe johndoe@example.com
Jane Doe
```

In the following example, the statement queries the email column and returns the count of valid email addresses:

```
SELECT empName, REGEXP_SUBSTR(emailID, '[[:alnum:]]+\@[[:alnum:]]+\.[[:alnum:]]
+') "Valid Email", REGEXP_INSTR(emailID, '\w+@\w+(\.\w+)+')
"FIELD_WITH_VALID_EMAIL" FROM regexp_temp;

EMPNAME        Valid Email            FIELD_WITH_VALID_EMAIL
--------    -------------------    ----------------------
John Doe    johndoe@example.com    1
Jane Doe
```

> **Live SQL:**
>
> View and run related examples on Oracle Live SQL at *REGEXP_SUBSTR pattern matching*

In the following example, numbers and alphabets are extracted from a string:

```
with strings as (
  select 'ABC123' str from dual union all
  select 'A1B2C3' str from dual union all
  select '123ABC' str from dual union all
  select '1A2B3C' str from dual
)
  select regexp_substr(str, '[0-9]') First_Occurrence_of_Number,
         regexp_substr(str, '[0-9].*') Num_Followed_by_String,
         regexp_substr(str, '[A-Z][0-9]') Letter_Followed_by_String
  from strings;

FIRST_OCCURRENCE_OF_NUMB NUM_FOLLOWED_BY_STRING   LETTER_FOLLOWED_BY_STRIN
------------------------ ------------------------ ------------------------
1             123              C1
1             1B2C3              A1
1             123ABC
1             1A2B3C           A2
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_SUBSTR - Extract Numbers and Alphabets*

In the following example, passenger names and flight information are extracted from a string:

```
with strings as (
  select 'LHRJFK/010315/JOHNDOE' str from dual union all
  select 'CDGLAX/050515/JANEDOE' str from dual union all
  select 'LAXCDG/220515/JOHNDOE' str from dual union all
  select 'SFOJFK/010615/JANEDOE' str from dual
)
  SELECT regexp_substr(str, '[A-Z]{6}') String_of_6_characters,
         regexp_substr(str, '[0-9]+') First_Matching_Numbers,
         regexp_substr(str, '[A-Z].*$') Letter_by_other_characters,
         regexp_substr(str, '/[A-Z].*$') Slash_letter_and_characters
  FROM strings;

STRING_OF_6_CHARACTERS    FIRST_MATCHING_NUMBERS    LETTER_BY_OTHER_CHARACTERS
SLASH_LETTER_AND_CHARACTERS
----------------------    ----------------------    --------------------------
---------------------------
LHRJFK                    010315                    LHRJFK/010315/
JOHNDOE              /JOHNDOE
CDGLAX                    050515                    CDGLAX/050515/
JANEDOE              /JANEDOE
LAXCDG                    220515                    LAXCDG/220515/
JOHNDOE              /JOHNDOE
```

```
SFOJFK                      010615                      SFOJFK/010615/
JANEDOE                   /JANEDOE
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_SUBSTR - Extract Passenger Names and Flight Information*

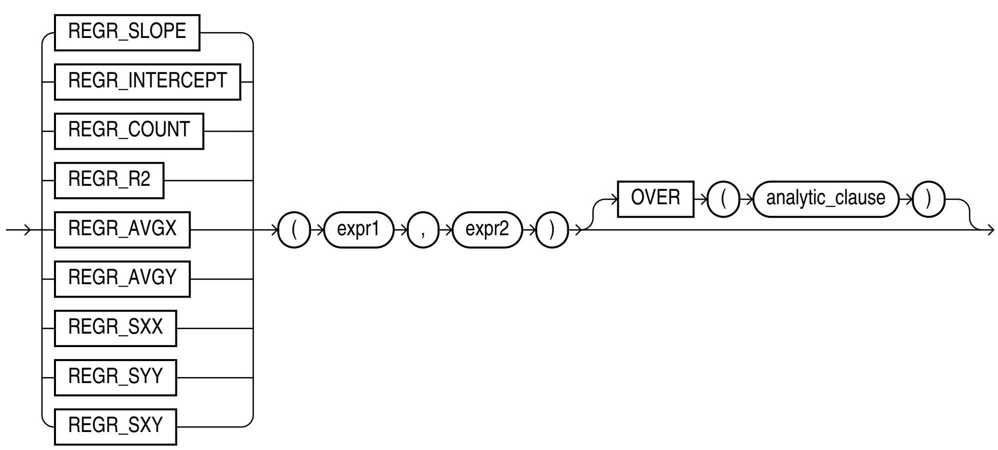
REGR_ (Linear Regression) Functions

The linear regression functions are:

- `REGR_SLOPE`
- `REGR_INTERCEPT`
- `REGR_COUNT`
- `REGR_R2`
- `REGR_AVGX`
- `REGR_AVGY`
- `REGR_SXX`
- `REGR_SYY`
- `REGR_SXY`

Syntax

***linear_regr*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

The linear regression functions fit an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate and analytic functions.

> **See Also:**
>
> "Aggregate Functions " and "About SQL Expressions " for information on valid forms of *expr*

These functions take as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Oracle applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Oracle computes all the regression functions simultaneously during a single pass through the data.

expr1 is interpreted as a value of the dependent variable (a `y` value), and *expr2* is interpreted as a value of the independent variable (an `x` value).

- `REGR_SLOPE` returns the slope of the line. The return value is a numeric data type and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  COVAR_POP(expr1, expr2) / VAR_POP(expr2)
  ```

- `REGR_INTERCEPT` returns the y-intercept of the regression line. The return value is a numeric data type and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  AVG(expr1) - REGR_SLOPE(expr1, expr2) * AVG(expr2)
  ```

- `REGR_COUNT` returns an integer that is the number of non-null number pairs used to fit the regression line.
- `REGR_R2` returns the coefficient of determination (also called R-squared or goodness of fit) for the regression. The return value is a numeric data type and can be null. `VAR_POP`(*expr1*) and `VAR_POP`(*expr2*) are evaluated after the elimination of null pairs. The return values are:

  ```
  NULL if VAR_POP(expr2)  = 0

     1 if VAR_POP(expr1)  = 0 and
          VAR_POP(expr2) != 0
  ```

```
POWER(CORR(expr1,expr),2) if VAR_POP(expr1)  > 0 and
                             VAR_POP(expr2  != 0
```

All of the remaining regression functions return a numeric data type and can be null:

- `REGR_AVGX` evaluates the average of the independent variable (*`expr2`*) of the regression line. It makes the following computation after the elimination of null (*`expr1`*, *`expr2`*) pairs:

  ```
  AVG(expr2)
  ```

- `REGR_AVGY` evaluates the average of the dependent variable (*`expr1`*) of the regression line. It makes the following computation after the elimination of null (*`expr1`*, *`expr2`*) pairs:

  ```
  AVG(expr1)
  ```

`REGR_SXY`, `REGR_SXX`, `REGR_SYY` are auxiliary functions that are used to compute various diagnostic statistics.

- `REGR_SXX` makes the following computation after the elimination of null (*`expr1`*, *`expr2`*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr2)
  ```

- `REGR_SYY` makes the following computation after the elimination of null (*`expr1`*, *`expr2`*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr1)
  ```

- `REGR_SXY` makes the following computation after the elimination of null (*`expr1`*, *`expr2`*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * COVAR_POP(expr1, expr2)
  ```

The following examples are based on the sample tables `sh.sales` and `sh.products`.

General Linear Regression Example

The following example provides a comparison of the various linear regression functions used in their analytic form. The analytic form of these functions can be useful when you want to use regression statistics for calculations such as finding the salary predicted for each employee by the model. The sections that follow on the individual linear regression functions contain examples of the aggregate form of these functions.

```
SELECT job_id, employee_id ID, salary,
REGR_SLOPE(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) slope,
REGR_INTERCEPT(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) intcpt,
REGR_R2(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) rsqr,
REGR_COUNT(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) count,
REGR_AVGX(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) avgx,
REGR_AVGY(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) avgy
   FROM employees
   WHERE department_id in (50, 80)
   ORDER BY job_id, employee_id;

JOB_ID          ID      SALARY SLOPE    INTCPT  RSQR  COUNT        AVGX       AVGY
```

```
---------- ----- ---------- ----- --------- ----- ------ ---------- ---------
SA_MAN       145      14000  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       146      13500  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       147      12000  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       148      11000  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       149      10500  .355 -1707.035  .832      5  12200.000  2626.589
SA_REP       150      10000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       151       9500  .257   404.763  .647     29   8396.552  2561.244
SA_REP       152       9000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       153       8000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       154       7500  .257   404.763  .647     29   8396.552  2561.244
SA_REP       155       7000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       156      10000  .257   404.763  .647     29   8396.552  2561.244
. . .
```

REGR_SLOPE and REGR_INTERCEPT Examples

The following example calculates the slope and regression of the linear regression model for time employed (`SYSDATE` - `hire_date`) and salary using the sample table `hr.employees`. Results are grouped by `job_id`.

```
SELECT job_id,
REGR_SLOPE(SYSDATE-hire_date, salary) slope,
REGR_INTERCEPT(SYSDATE-hire_date, salary) intercept
   FROM employees
   WHERE department_id in (50,80)
   GROUP BY job_id
   ORDER BY job_id;

JOB_ID     SLOPE    INTERCEPT
---------- ----- ------------
SA_MAN      .355 -1707.030762
SA_REP      .257   404.767151
SH_CLERK    .745   159.015293
ST_CLERK    .904   134.409050
ST_MAN      .479  -570.077291
```

REGR_COUNT Examples

The following example calculates the count of by `job_id` for time employed (`SYSDATE` - `hire_date`) and salary using the sample table `hr.employees`. Results are grouped by `job_id`.

```
SELECT job_id,
REGR_COUNT(SYSDATE-hire_date, salary) count
   FROM employees
   WHERE department_id in (30, 50)
   GROUP BY job_id
   ORDER BY job_id, count;

JOB_ID          COUNT
---------- ----------
PU_CLERK            5
PU_MAN              1
SH_CLERK           20
ST_CLERK           20
ST_MAN              5
```

REGR_R2 Examples

The following example calculates the coefficient of determination the linear regression of time employed (`SYSDATE` - `hire_date`) and salary using the sample table `hr.employees`:

```
SELECT job_id,
REGR_R2(SYSDATE-hire_date, salary) Regr_R2
   FROM employees
   WHERE department_id in (80, 50)
   GROUP by job_id
  ORDER BY job_id, Regr_R2;

JOB_ID        REGR_R2
---------- ----------
SA_MAN      .83244748
SA_REP     .647007156
SH_CLERK   .879799698
ST_CLERK   .742808493
ST_MAN      .69418508
```

REGR_AVGY and REGR_AVGX Examples

The following example calculates the average values for time employed (`SYSDATE` - `hire_date`) and salary using the sample table `hr.employees`. Results are grouped by `job_id`:

```
SELECT job_id,
REGR_AVGY(SYSDATE-hire_date, salary) avgy,
REGR_AVGX(SYSDATE-hire_date, salary) avgx
   FROM employees
   WHERE department_id in (30,50)
   GROUP BY job_id
   ORDER BY job_id, avgy, avgx;

JOB_ID           AVGY       AVGX
---------- ---------- ----------
PU_CLERK    2950.3778       2780
PU_MAN      4026.5778      11000
SH_CLERK    2773.0778       3215
ST_CLERK    2872.7278       2785
ST_MAN      3140.1778       7280
```

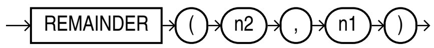
REGR_SXY, REGR_SXX, and REGR_SYY Examples

The following example calculates three types of diagnostic statistics for the linear regression of time employed (`SYSDATE` - `hire_date`) and salary using the sample table `hr.employees`:

```
SELECT job_id,
REGR_SXY(SYSDATE-hire_date, salary) regr_sxy,
REGR_SXX(SYSDATE-hire_date, salary) regr_sxx,
REGR_SYY(SYSDATE-hire_date, salary) regr_syy
   FROM employees
   WHERE department_id in (80, 50)
   GROUP BY job_id
   ORDER BY job_id;

JOB_ID       REGR_SXY    REGR_SXX   REGR_SYY
---------- ---------- ----------- ----------
SA_MAN        3303500   9300000.0    1409642
SA_REP     16819665.5  65489655.2 6676562.55
SH_CLERK      4248650   5705500.0    3596039
ST_CLERK      3531545   3905500.0 4299084.55
ST_MAN        2180460   4548000.0  1505915.2
```

REMAINDER

Syntax



Purpose

`REMAINDER` returns the remainder of *n2* divided by *n1*.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

The `MOD` function is similar to `REMAINDER` except that it uses `FLOOR` in its formula, whereas `REMAINDER` uses `ROUND`. Refer to MOD .

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

- If *n1* = 0 or *n2* = infinity, then Oracle returns
  - An error if the arguments are of type `NUMBER`
  - `NaN` if the arguments are `BINARY_FLOAT` or `BINARY_DOUBLE`.
- If *n1* != 0, then the remainder is *n2* - (*n1***N*) where *N* is the integer nearest *n2*/*n1*. If *n2*/*n1* equals `x.5`, then *N* is the nearest *even* integer.
- If *n2* is a floating-point number, and if the remainder is 0, then the sign of the remainder is the sign of *n2*. Remainders of 0 are unsigned for `NUMBER` values.

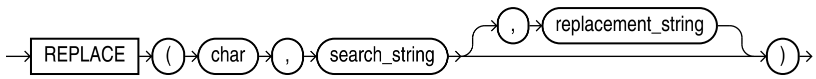
Examples

Using table `float_point_demo`, created for the `TO_BINARY_DOUBLE` "Examples", the following example divides two floating-point numbers and returns the remainder of that operation:

```
SELECT bin_float, bin_double, REMAINDER(bin_float, bin_double)
  FROM float_point_demo;

 BIN_FLOAT BIN_DOUBLE REMAINDER(BIN_FLOAT,BIN_DOUBLE)
---------- ---------- ------------------------------
1.235E+003 1.235E+003                      5.859E-005
```

REPLACE

Syntax



Purpose

REPLACE returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, then all occurrences of *search_string* are removed. If *search_string* is null, then *char* is returned.

Both *search_string* and *replacement_string*, as well as *char*, can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is in the same character set as *char*. The function returns VARCHAR2 if the first argument is not a LOB and returns CLOB if the first argument is a LOB.

REPLACE provides functionality related to that provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE lets you substitute one string for another as well as to remove character strings.

> **See Also:**
>
> - TRANSLATE
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation REPLACE uses to compare characters from *char* with characters from *search_string*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

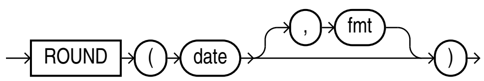
The following example replaces occurrences of J with BL:

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
     FROM DUAL;

Changes
--------------
BLACK and BLUE
```

ROUND (date)

Syntax

***round_date*::=**



Purpose

ROUND returns *date* rounded to the unit specified by the format model *fmt*. This function is not sensitive to the NLS_CALENDAR session parameter. It operates according to the rules of the Gregorian calendar. The value returned is always of data type DATE, even if you specify a different datetime data type for *date*. If you omit *fmt*, then *date* is rounded to the nearest day. The *date* expression must resolve to a DATE value.

> **See Also:**
>
> "ROUND and TRUNC Date Functions" for the permitted format models to use in *fmt*

Examples

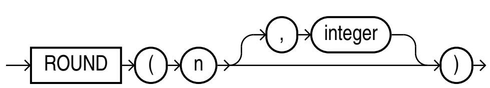
The following example rounds a date to the first day of the following year:

```
SELECT ROUND (TO_DATE ('27-OCT-00'),'YEAR')
   "New Year" FROM DUAL;

New Year
---------
01-JAN-01
```

ROUND (number)

Syntax

***round_number*::=**



Purpose

`ROUND` returns *n* rounded to *integer* places to the right of the decimal point. If you omit *integer*, then *n* is rounded to zero places. If *integer* is negative, then *n* is rounded off to the left of the decimal point.

n can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If you omit *integer*, then the function returns the value `ROUND`(n, 0) in the same data type as the numeric data type of *n*. If you include *integer*, then the function returns `NUMBER`.

`ROUND` is implemented using the following rules:

1. If *n* is 0, then `ROUND` always returns 0 regardless of *integer*.
2. If *n* is negative, then `ROUND`(n, integer) returns -`ROUND`(-n, integer).
3. If *n* is positive, then

   ```
   ROUND(n, integer) = FLOOR(n * POWER(10, integer) + 0.5) * POWER(10, -integer)
   ```

`ROUND` applied to a `NUMBER` value may give a slightly different result from `ROUND` applied to the same value expressed in floating-point. The different results arise from differences in internal representations of `NUMBER` and floating point values. The difference will be 1 in the rounded digit if a difference occurs.

> **See Also:**
>
> - Table 2-8 for more information on implicit conversion
> - "Floating-Point Numbers " for more information on how Oracle Database handles `BINARY_FLOAT` and `BINARY_DOUBLE` values
> - FLOOR and CEIL , TRUNC (number) and MOD for information on functions that perform related operations

Examples

The following example rounds a number to one decimal point:

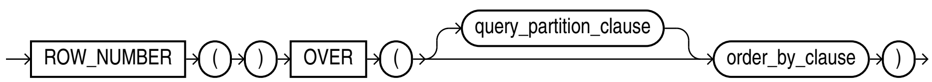
```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2
```

The following example rounds a number one digit to the left of the decimal point:

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;

     Round
----------
        20
```

ROW_NUMBER

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`ROW_NUMBER` is an analytic function. It assigns a unique number to each row to which it is applied (either each row in the partition or each row returned by the query), in the ordered sequence of rows specified in the *`order_by_clause`*, beginning with 1.

By nesting a subquery using `ROW_NUMBER` inside a query that retrieves the `ROW_NUMBER` values for a specified range, you can find a precise subset of rows from the results of the inner query. This use of the function lets you implement top-N, bottom-N, and inner-N reporting. For consistent results, the query must ensure a deterministic sort order.

Examples

The following example finds the three highest paid employees in each department in the `hr.employees` table. Fewer than three rows are returned for departments with fewer than three employees.

```
SELECT department_id, first_name, last_name, salary
FROM
(
  SELECT
    department_id, first_name, last_name, salary,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary desc) rn
  FROM employees
)
WHERE rn <= 3
ORDER BY department_id, salary DESC, last_name;
```

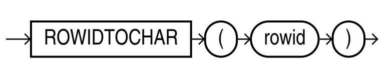
The following example is a join query on the `sh.sales` table. It finds the sales amounts in 2000 of the five top-selling products in 1999 and compares the difference between 2000 and 1999. The ten top-selling products are calculated within each distribution channel.

```
SELECT sales_2000.channel_desc, sales_2000.prod_name,
       sales_2000.amt amt_2000,  top_5_prods_1999_year.amt amt_1999,
       sales_2000.amt  - top_5_prods_1999_year.amt amt_diff
FROM
/* The first subquery finds the 5 top-selling products per channel in year 1999. */
  (SELECT channel_desc, prod_name, amt
   FROM
   (
     SELECT channel_desc, prod_name, sum(amount_sold) amt,
```

```
       ROW_NUMBER () OVER (PARTITION BY channel_desc
                           ORDER BY SUM(amount_sold) DESC) rn
     FROM sales, times, channels, products
     WHERE sales.time_id = times.time_id
       AND times.calendar_year = 1999
       AND channels.channel_id = sales.channel_id
       AND products.prod_id = sales.prod_id
     GROUP BY channel_desc, prod_name
   )
   WHERE rn <= 5
  ) top_5_prods_1999_year,
/* The next subquery finds sales per product and per channel in 2000. */
  (SELECT channel_desc, prod_name, sum(amount_sold) amt
     FROM sales, times, channels, products
     WHERE sales.time_id = times.time_id
       AND times.calendar_year = 2000
       AND channels.channel_id = sales.channel_id
       AND products.prod_id = sales.prod_id
     GROUP BY channel_desc, prod_name
  ) sales_2000
WHERE sales_2000.channel_desc = top_5_prods_1999_year.channel_desc
  AND sales_2000.prod_name = top_5_prods_1999_year.prod_name
ORDER BY sales_2000.channel_desc, sales_2000.prod_name
;
CHANNEL_DESC    PROD_NAME                                          AMT_2000   AMT_1999   AMT_DIFF
--------------- --------------==-------------------------------- ---------- ---------- ----------
Direct Sales     17" LCD w/built-in HDTV Tuner                     628855.7 1163645.78 -534790.08
Direct Sales     Envoy 256MB - 40GB                               502938.54  843377.88 -340439.34
Direct Sales     Envoy Ambassador                                2259566.96 1770349.25  489217.71
Direct Sales     Home Theatre Package with DVD-Audio/Video Play  1235674.15 1260791.44  -25117.29
Direct Sales     Mini DV Camcorder with 3.5" Swivel LCD           775851.87 1326302.51 -550450.64
Internet         17" LCD w/built-in HDTV Tuner                     31707.48   160974.7 -129267.22
Internet         8.3 Minitower Speaker                            404090.32  155235.25  248855.07
Internet         Envoy 256MB - 40GB                                28293.87  154072.02 -125778.15
Internet         Home Theatre Package with DVD-Audio/Video Play   155405.54  153175.04     2230.5
Internet         Mini DV Camcorder with 3.5" Swivel LCD            39726.23  189921.97 -150195.74
Partners         17" LCD w/built-in HDTV Tuner                    269973.97  325504.75  -55530.78
Partners         Envoy Ambassador                                1213063.59  614857.93  598205.66
Partners         Home Theatre Package with DVD-Audio/Video Play   700266.58  520166.26  180100.32
Partners         Mini DV Camcorder with 3.5" Swivel LCD           404265.85  520544.11 -116278.26
Partners         Unix/Windows 1-user pack                         374002.51  340123.02   33879.49

15 rows selected.
```

ROWIDTOCHAR

Syntax



Purpose

`ROWIDTOCHAR` converts a rowid value to `VARCHAR2` data type. The result of this conversion is always 18 characters long.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of ROWIDTOCHAR

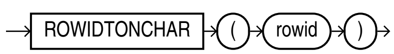
Examples

The following example converts a rowid value in the employees table to a character value. (Results vary for each build of the sample database.)

```
SELECT ROWID FROM employees
   WHERE ROWIDTOCHAR(ROWID) LIKE '%JAAB%'
   ORDER BY ROWID;

ROWID
------------------
AAAFfIAAFAAAABSAAb
```

ROWIDTONCHAR

Syntax



Purpose

ROWIDTONCHAR converts a rowid value to NVARCHAR2 data type. The result of this conversion is always in the national character set and is 18 characters long.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of ROWIDTONCHAR

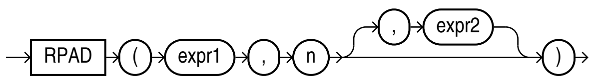
Examples

The following example converts a rowid value to an NVARCHAR2 string:

```
SELECT LENGTHB( ROWIDTONCHAR(ROWID) ) Length, ROWIDTONCHAR(ROWID)
   FROM employees
   ORDER BY length;

    LENGTH ROWIDTONCHAR(ROWID
---------- ------------------
        36 AAAL52AAFAAAABSABD
        36 AAAL52AAFAAAABSABV
. . .
```

RPAD

Syntax



Purpose

`RPAD` returns *`expr1`*, right-padded to length *`n`* characters with *`expr2`*, replicated as many times as necessary. This function is useful for formatting the output of a query.

Both *`expr1`* and *`expr2`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`expr1`* is a character data type, `NVARCHAR2` if *`expr1`* is a national character data type, and a LOB if *`expr1`* is a LOB data type. The string returned is in the same character set as *`expr1`*. The argument *`n`* must be a `NUMBER` integer or a value that can be implicitly converted to a `NUMBER` integer.

`expr1` cannot be null. If you do not specify *`expr2`*, then it defaults to a single blank. If *`expr1`* is longer than *`n`*, then this function returns the portion of *`expr1`* that fits in *`n`*.

The argument *`n`* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `RPAD`

Examples

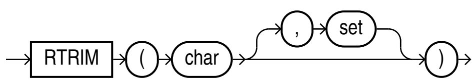
The following example creates a simple chart of salary amounts by padding a single space with asterisks:

```
SELECT last_name, RPAD(' ', salary/1000/1, '*') "Salary"
   FROM employees
   WHERE department_id = 80
   ORDER BY last_name, "Salary";

LAST_NAME                 Salary
------------------------- ---------------
Abel                       **********
Ande                       *****
Banda                      *****
Bates                      ******
Bernstein                  ********
Bloom                      *********
```

```
Cambrault                  **********
Cambrault                  ******
Doran                      ******
Errazuriz                  ***********
Fox                        ********
Greene                     ********
Hall                       ********
Hutton                     *******
Johnson                    *****
King                       *********
. . .
```

RTRIM

Syntax



Purpose

`RTRIM` removes from the right end of *`char`* all of the characters that appear in *`set`*. This function is useful for formatting the output of a query.

If you do not specify *`set`*, then it defaults to a single blank. `RTRIM` works similarly to `LTRIM`.

Both *`char`* and *`set`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`char`* is a character data type, `NVARCHAR2` if *`char`* is a national character data type, and a LOB if *`char`* is a LOB data type.

> **See Also:**
>
> - LTRIM
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `RTRIM` uses to compare characters from *`set`* with characters from *`char`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

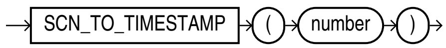
Examples

The following example trims all the right-most occurrences of less than sign (<), greater than sign (>) , and equal sign (=) from a string:

```
SELECT RTRIM('<=====>BROWNING<=====>', '<>=') "RTRIM Example"
  FROM DUAL;

RTRIM Example
---------------
<=====>BROWNING
```

SCN_TO_TIMESTAMP

Syntax



Purpose

`SCN_TO_TIMESTAMP` takes as an argument a number that evaluates to a system change number (SCN), and returns the approximate timestamp associated with that SCN. The returned value is of `TIMESTAMP` data type. This function is useful any time you want to know the timestamp associated with an SCN. For example, it can be used in conjunction with the `ORA_ROWSCN` pseudocolumn to associate a timestamp with the most recent change to a row.

> **Notes:**
>
> - The usual precision of the result value is 3 seconds.
> - The association between an SCN and a timestamp when the SCN is generated is remembered by the database for a limited period of time. This period is the maximum of the auto-tuned undo retention period, if the database runs in the Automatic Undo Management mode, and the retention times of all flashback archives in the database, but no less than 120 hours. The time for the association to become obsolete elapses only when the database is open. An error is returned if the SCN specified for the argument to `SCN_TO_TIMESTAMP` is too old.

> **See Also:**
>
> ORA_ROWSCN Pseudocolumn and TIMESTAMP_TO_SCN

Examples

The following example uses the `ORA_ROWSCN` pseudocolumn to determine the system change number of the last update to a row and uses `SCN_TO_TIMESTAMP` to convert that SCN to a timestamp:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM employees
   WHERE employee_id = 188;
```

You could use such a query to convert a system change number to a timestamp for use in an Oracle Flashback Query:

```
SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
      3800
```

```
UPDATE employees SET salary = salary*10 WHERE employee_id = 188;
COMMIT;

SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
     38000

SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM employees
   WHERE employee_id = 188;
SCN_TO_TIMESTAMP(ORA_ROWSCN)
---------------------------------------------------------------------------
28-AUG-03 01.58.01.000000000 PM

FLASHBACK TABLE employees TO TIMESTAMP
   TO_TIMESTAMP('28-AUG-03 01.00.00.000000000 PM');

SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
      3800
```

SESSIONTIMEZONE

Syntax

→ SESSIONTIMEZONE →

Purpose

SESSIONTIMEZONE returns the time zone of the current session. The return type is a time zone offset (a character type in the format '[+|-]TZH:TZM') or a time zone region name, depending on how the user specified the session time zone value in the most recent ALTER SESSION statement.

> **Note:**
>
> The default client session time zone is an offset even if the client operating system uses a named time zone. If you want the default session time zone to use a named time zone, then set the ORA_SDTZ variable in the client environment to an Oracle time zone region name. Refer to *Oracle Database Globalization Support Guide* for more information on this variable.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of SESSIONTIMEZONE

Examples

The following example returns the time zone of the current session:

```
SELECT SESSIONTIMEZONE FROM DUAL;

SESSION
-------
-08:00
```

SET

Syntax

Purpose

`SET` converts a nested table into a set by eliminating duplicates. The function returns a nested table whose elements are distinct from one another. The returned nested table is of the same type as the input nested table.

The element types of the nested table must be comparable. Refer to "Comparison Conditions " for information on the comparability of nonscalar types.

Examples

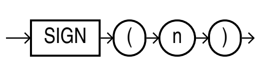
The following example selects from the `customers_demo` table the unique elements of the `cust_address_ntab` nested table column:

```
SELECT customer_id, SET(cust_address_ntab) address
  FROM customers_demo
  ORDER BY customer_id;


CUSTOMER_ID ADDRESS(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
----------- ------------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'))
        102 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'))
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
        104 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'))
        105 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
. . .
```

The preceding example requires the table `customers_demo` and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table column.

SIGN

Syntax



Purpose

`SIGN` returns the sign of *n*. This function takes as an argument any numeric data type, or any nonnumeric data type that can be implicitly converted to `NUMBER`, and returns `NUMBER`.

For value of `NUMBER` type, the sign is:

- -1 if *n*<0
- 0 if *n*=0
- 1 if *n*>0

For binary floating-point numbers (`BINARY_FLOAT` and `BINARY_DOUBLE`), this function returns the sign bit of the number. The sign bit is:

- -1 if *n*<0
- +1 if *n*>=0 or *n*=`NaN`

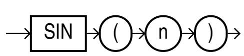
Examples

The following example indicates that the argument of the function (`-15`) is <0:

```
SELECT SIGN(-15) "Sign" FROM DUAL;

      Sign
----------
        -1
```

SIN

Syntax



Purpose

`SIN` returns the sine of *n* (an angle expressed in radians).

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

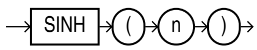
Examples

The following example returns the sine of 30 degrees:

```
SELECT SIN(30 * 3.14159265359/180)
 "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax



Purpose

`SINH` returns the hyperbolic sine of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

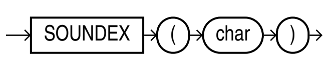
Examples

The following example returns the hyperbolic sine of 1:

```
SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;

Hyperbolic sine of 1
--------------------
          1.17520119
```

SOUNDEX

Syntax



Purpose

`SOUNDEX` returns a character string containing the phonetic representation of *`char`*. This function lets you compare words that are spelled differently, but sound alike in English.

The phonetic representation is defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

1. Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
2. Assign numbers to the remaining letters (after the first) as follows:

```
b, f, p, v = 1
c, g, j, k, q, s, x, z = 2
d, t = 3
l = 4
m, n = 5
r = 6
```

3. If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, then retain the first letter and omit rest of all the adjacent letters with same number.
4. Return the first four bytes padded with 0.

char can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The return value is the same data type as *char*.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of SOUNDEX

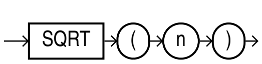
Examples

The following example returns the employees whose last names are a phonetic representation of "Smyth":

```
SELECT last_name, first_name
     FROM hr.employees
     WHERE SOUNDEX(last_name)
         = SOUNDEX('SMYTHE')
     ORDER BY last_name, first_name;

LAST_NAME  FIRST_NAME
---------- ----------
Smith      Lindsey
Smith      William
```

SQRT

Syntax



Purpose

`SQRT` returns the square root of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

- If *n* resolves to a `NUMBER`, then the value *n* cannot be negative. `SQRT` returns a real number.
- If *n* resolves to a binary floating-point number (`BINARY_FLOAT` or `BINARY_DOUBLE`):
  - If *n* >= 0, then the result is positive.
  - If *n* = -0, then the result is -0.
  - If *n* < 0, then the result is `NaN`.

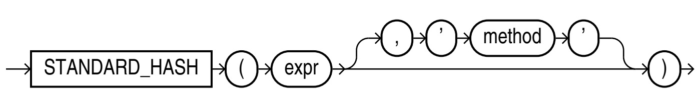
Examples

The following example returns the square root of 26:

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
5.09901951
```

STANDARD_HASH

Syntax



Purpose

`STANDARD_HASH` computes a hash value for a given expression using one of several hash algorithms that are defined and standardized by the National Institute of Standards and Technology. This function is useful for performing authentication and maintaining data integrity in security applications such as digital signatures, checksums, and fingerprinting.

You can use the `STANDARD_HASH` function to create an index on an extended data type column. Refer to "Creating an Index on an Extended Data Type Column" for more information.

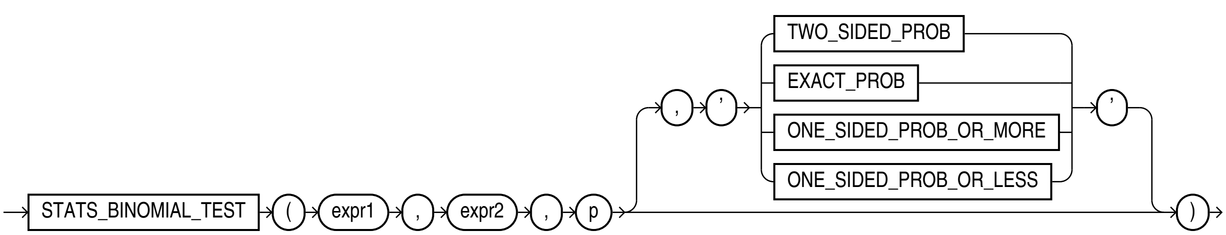
- The *expr* argument determines the data for which you want Oracle Database to compute a hash value. There are no restrictions on the length of data represented by *expr*, which commonly resolves to a column name. The *expr* cannot be a `LONG` or LOB type. It cannot be a user-defined object type. All other data types are supported for *expr*.
- The optional *method* argument lets you specify the name of the hash algorithm to be used. Valid algorithms are `SHA1`, `SHA256`, `SHA384`, `SHA512`, and `MD5`. If you omit this argument, then `SHA1` is used.

The function returns a `RAW` value.

> **Note:**
>
> The `STANDARD_HASH` function is not identical to the one used internally by Oracle Database for hash partitioning.

STATS_BINOMIAL_TEST

Syntax



Purpose

`STATS_BINOMIAL_TEST` is an exact probability test used for dichotomous variables, where only two possible values exist. It tests the difference between a sample proportion and a given proportion. The sample size in such tests is usually small.

This function takes three required arguments: *expr1* is the sample being examined, *expr2* contains the values for which the proportion is expected to be, and *p* is a proportion to test against. The optional fourth argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-3. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the fourth argument, then the default is `'TWO_SIDED_PROB'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_BINOMIAL_TEST`

Table 7-3 STATS_BINOMIAL Return Values

Argument	Return Value Meaning
`'TWO_SIDED_PROB'`	The probability that the given population proportion, *p*, could result in the observed proportion or a more extreme one.
`'EXACT_PROB'`	The probability that the given population proportion, *p*, could result in exactly the observed proportion.
`'ONE_SIDED_PROB_OR_MORE'`	The probability that the given population proportion, *p*, could result in the observed proportion or a larger one.
`'ONE_SIDED_PROB_OR_LESS'`	The probability that the given population proportion, *p*, could result in the observed proportion or a smaller one.

`'EXACT_PROB'` gives the probability of getting exactly proportion *p*. In cases where you want to test whether the proportion found in the sample is significantly different from a 50-50 split, *p* would normally be 0.50. If you want to test only whether the proportion is different, then use the return value `'TWO_SIDED_PROB'`. If your test is whether the proportion is more than the value of *expr2*, then use the return value `'ONE_SIDED_PROB_OR_MORE'`. If the test is to determine whether the proportion of *expr2* is less, then use the return value `'ONE_SIDED_PROB_OR_LESS'`.

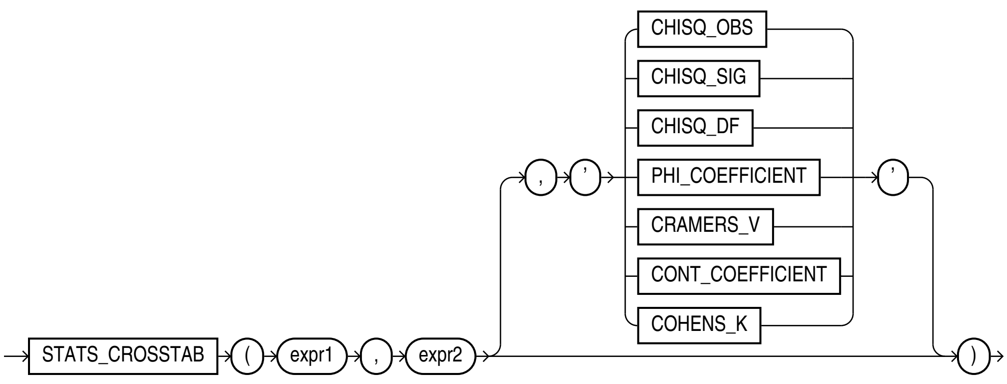
STATS_BINOMIAL_TEST Example

The following example determines the probability that reality exactly matches the number of men observed under the assumption that 69% of the population is composed of men:

```
SELECT AVG(DECODE(cust_gender, 'M', 1, 0)) real_proportion,
       STATS_BINOMIAL_TEST
         (cust_gender, 'M', 0.68, 'EXACT_PROB') exact,
       STATS_BINOMIAL_TEST
         (cust_gender, 'M', 0.68, 'ONE_SIDED_PROB_OR_LESS') prob_or_less
  FROM sh.customers;
```

STATS_CROSSTAB

Syntax



Purpose

Crosstabulation (commonly called crosstab) is a method used to analyze two nominal variables. The `STATS_CROSSTAB` function takes two required arguments: *`expr1`* and *`expr2`* are the two variables being analyzed. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-4. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'CHISQ_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_CROSSTAB`

Table 7-4 STATS_CROSSTAB Return Values

Argument	Return Value Meaning
`'CHISQ_OBS'`	Observed value of chi-squared
`'CHISQ_SIG'`	Significance of observed chi-squared
`'CHISQ_DF'`	Degree of freedom for chi-squared
`'PHI_COEFFICIENT'`	Phi coefficient
`'CRAMERS_V'`	Cramer's V statistic
`'CONT_COEFFICIENT'`	Contingency coefficient
`'COHENS_K'`	Cohen's kappa

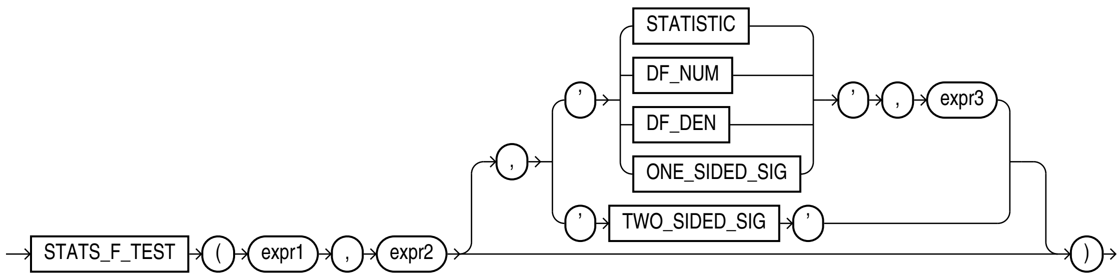
STATS_CROSSTAB Example

The following example determines the strength of the association between gender and income level:

```
SELECT STATS_CROSSTAB
         (cust_gender, cust_income_level, 'CHISQ_OBS') chi_squared,
       STATS_CROSSTAB
         (cust_gender, cust_income_level, 'CHISQ_SIG') p_value,
       STATS_CROSSTAB
         (cust_gender, cust_income_level, 'PHI_COEFFICIENT') phi_coefficient
  FROM sh.customers;

CHI_SQUARED    P_VALUE PHI_COEFFICIENT
----------- ---------- ---------------
 251.690705 1.2364E-47      .067367056
```

STATS_F_TEST

Syntax



Purpose

`STATS_F_TEST` tests whether two variances are significantly different. The observed value of *f* is the ratio of one variance to the other, so values very different from 1 usually indicate significant differences.

This function takes two required arguments: *`expr1`* is the grouping or independent variable and *`expr2`* is the sample of values. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-5. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_F_TEST`

Table 7-5 STATS_F_TEST Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *f*
`'DF_NUM'`	Degree of freedom for the numerator
`'DF_DEN'`	Degree of freedom for the denominator
`'ONE_SIDED_SIG'`	One-tailed significance of *f*
`'TWO_SIDED_SIG'`	Two-tailed significance of *f*

The one-tailed significance is always in relation to the upper tail. The final argument, *`expr3`*, indicates which of the two groups specified by *`expr1`* is the high value or numerator (the value whose rejection region is the upper tail).

The observed value of *f* is the ratio of the variance of one group to the variance of the second group. The significance of the observed value of *f* is the probability that the variances are different just by chance—a number between 0 and 1. A small value for the significance indicates that the variances are significantly different. The degree of freedom for each of the variances is the number of observations in the sample minus 1.

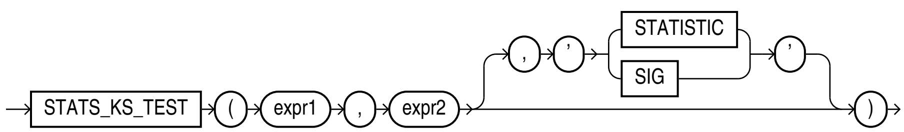
STATS_F_TEST Example

The following example determines whether the variance in credit limit between men and women is significantly different. The results, a p_value not close to zero, and an f_statistic close to 1, indicate that the difference between credit limits for men and women are not significant.

```
SELECT VARIANCE(DECODE(cust_gender, 'M', cust_credit_limit, null)) var_men,
       VARIANCE(DECODE(cust_gender, 'F', cust_credit_limit, null)) var_women,
       STATS_F_TEST(cust_gender, cust_credit_limit, 'STATISTIC', 'F') f_statistic,
       STATS_F_TEST(cust_gender, cust_credit_limit) two_sided_p_value
  FROM sh.customers;

   VAR_MEN  VAR_WOMEN F_STATISTIC TWO_SIDED_P_VALUE
---------- ---------- ----------- -----------------
12879896.7   13046865  1.01296348         .311928071
```

STATS_KS_TEST

Syntax



Purpose

`STATS_KS_TEST` is a Kolmogorov-Smirnov function that compares two samples to test whether they are from the same population or from populations that have the same distribution. It does not assume that the population from which the samples were taken is normally distributed.

This function takes two required arguments: *`expr1`* classifies the data into the two samples and *`expr2`* contains the values for each of the samples. If *`expr1`* classifies the data into only one sample or into more than two samples, then an error is raised. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-6. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_KS_TEST`

Table 7-6 STATS_KS_TEST Return Values

Argument	Return Value Meaning
'STATISTIC'	Observed value of *D*
'SIG'	Significance of *D*

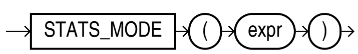
STATS_KS_TEST Example

Using the Kolmogorov Smirnov test, the following example determines whether the distribution of sales between men and women is due to chance:

```
SELECT stats_ks_test(cust_gender, amount_sold, 'STATISTIC') ks_statistic,
       stats_ks_test(cust_gender, amount_sold) p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id;

KS_STATISTIC    P_VALUE
------------ ----------
  .003841396 .004080006
```

STATS_MODE

Syntax



Purpose

STATS_MODE takes as its argument a set of values and returns the value that occurs with the greatest frequency. If more than one mode exists, then Oracle Database chooses one and returns only that one value.

To obtain multiple modes (if multiple modes exist), you must use a combination of other functions, as shown in the hypothetical query:

```
SELECT x FROM (SELECT x, COUNT(x) AS cnt1
   FROM t GROUP BY x)
   WHERE cnt1 =
      (SELECT MAX(cnt2) FROM (SELECT COUNT(x) AS cnt2 FROM t GROUP BY x));
```

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation STATS_MODE uses to compare character values for *expr* , and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following example returns the mode of salary per department in the `hr.employees` table:

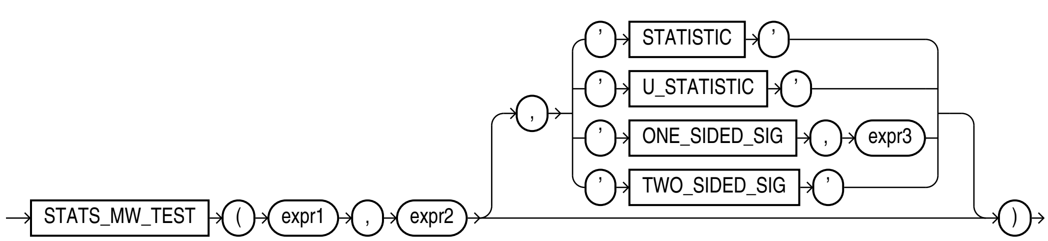
```
SELECT department_id, STATS_MODE(salary) FROM employees
   GROUP BY department_id
   ORDER BY department_id, stats_mode(salary);

DEPARTMENT_ID STATS_MODE(SALARY)
------------- ------------------
           10               4400
           20               6000
           30               2500
           40               6500
           50               2500
           60               4800
           70              10000
           80               9500
           90              17000
          100               6900
          110               8300
                            7000
```

If you need to retrieve all of the modes (in cases with multiple modes), you can do so using a combination of other functions, as shown in the next example:

```
SELECT commission_pct FROM
   (SELECT commission_pct, COUNT(commission_pct) AS cnt1 FROM employees
      GROUP BY commission_pct)
   WHERE cnt1 =
      (SELECT MAX (cnt2) FROM
         (SELECT COUNT(commission_pct) AS cnt2
         FROM employees GROUP BY commission_pct))
   ORDER BY commission_pct;

COMMISSION_PCT
--------------
            .2
            .3
```

STATS_MW_TEST

Syntax



Purpose

A Mann Whitney test compares two independent samples to test the null hypothesis that two populations have the same distribution function against the alternative hypothesis that the two distribution functions are different.

The `STATS_MW_TEST` does not assume that the differences between the samples are normally distributed, as do the `STATS_T_TEST_*` functions. This function takes two required arguments: *`expr1`* classifies the data into groups and *`expr2`* contains the values for each of the groups. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-7. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_MW_TEST`

Table 7-7 STATS_MW_TEST Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *Z*
`'U_STATISTIC'`	The observed value of *U*
`'ONE_SIDED_SIG'`	One-tailed significance of *Z*
`'TWO_SIDED_SIG'`	Two-tailed significance of *Z*

The significance of the observed value of *Z* or *U* is the probability that the variances are different just by chance—a number between 0 and 1. A small value for the significance indicates that the variances are significantly different. The degree of freedom for each of the variances is the number of observations in the sample minus 1.

The one-tailed significance is always in relation to the upper tail. The final argument, *`expr3`*, indicates which of the two groups specified by expr1 is the high value (the value whose rejection region is the upper tail).

`STATS_MW_TEST` computes the probability that the samples are from the same distribution by checking the differences in the sums of the ranks of the values. If the samples come from the same distribution, then the sums should be close in value.

STATS_MW_TEST Example

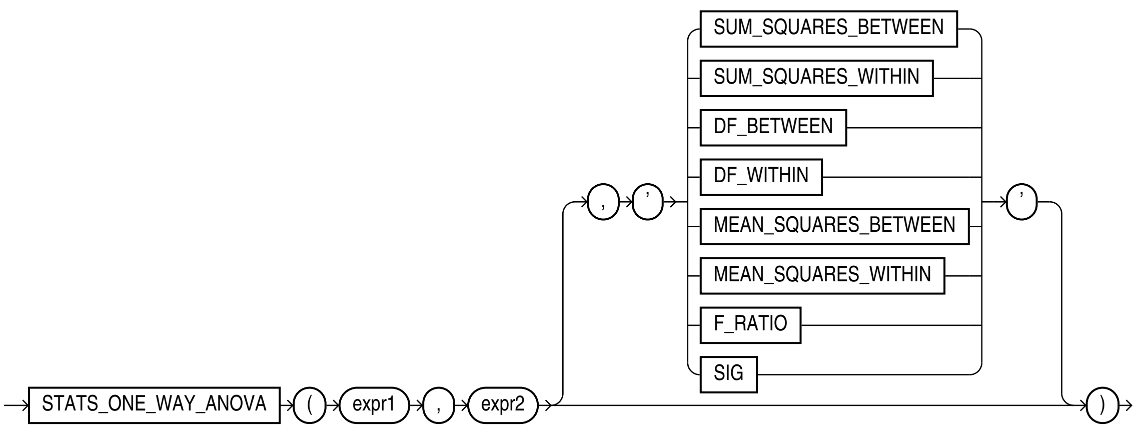
Using the Mann Whitney test, the following example determines whether the distribution of sales between men and women is due to chance:

```
SELECT STATS_MW_TEST
         (cust_gender, amount_sold, 'STATISTIC') z_statistic,
       STATS_MW_TEST
         (cust_gender, amount_sold, 'ONE_SIDED_SIG', 'F') one_sided_p_value
```

```
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id;

Z_STATISTIC ONE_SIDED_P_VALUE
----------- -----------------
 -1.4011509        .080584471
```

STATS_ONE_WAY_ANOVA

Syntax



Purpose

The one-way analysis of variance function (`STATS_ONE_WAY_ANOVA`) tests differences in means (for groups or variables) for statistical significance by comparing two different estimates of variance. One estimate is based on the variances within each group or category. This is known as the **mean squares within** or **mean square error**. The other estimate is based on the variances among the means of the groups. This is known as the **mean squares between**. If the means of the groups are significantly different, then the mean squares between will be larger than expected and will not match the mean squares within. If the mean squares of the groups are consistent, then the two variance estimates will be about the same.

`STATS_ONE_WAY_ANOVA` takes two required arguments: *`expr1`* is an independent or grouping variable that divides the data into a set of groups and *`expr2`* is a dependent variable (a numeric expression) containing the values corresponding to each member of a group. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-8. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_ONE_WAY_ANOVA`

Table 7-8 STATS_ONE_WAY_ANOVA Return Values

Argument	Return Value Meaning
`'SUM_SQUARES_BETEEN'`	Sum of squares between groups
`'SUM_SQUARES_WITHIN'`	Sum of squares within groups
`'DF_BETWEEN'`	Degree of freedom between groups
`'DF_WITHIN'`	Degree of freedom within groups
`'MEAN_SQUARES_BETWEEN'`	Mean squares between groups
`'MEAN_SQUARES_WITHIN'`	Mean squares within groups
`'F_RATIO'`	Ratio of the mean squares between to the mean squares within (MSB/MSW)
`'SIG'`	Significance

The significance of one-way analysis of variance is determined by obtaining the one-tailed significance of an *f*-test on the ratio of the mean squares between and the mean squares within. The *f*-test should use one-tailed significance, because the mean squares between can be only equal to or larger than the mean squares within. Therefore, the significance returned by `STATS_ONE_WAY_ANOVA` is the probability that the differences between the groups happened by chance—a number between 0 and 1. The smaller the number, the greater the significance of the difference between the groups. Refer to the STATS_F_TEST for information on performing an *f*-test.

STATS_ONE_WAY_ANOVA Example

The following example determines the significance of the differences in mean sales within an income level and differences in mean sales between income levels. The results, p_values close to zero, indicate that, for both men and women, the difference in the amount of goods sold across different income levels is significant.

```
SELECT cust_gender,
       STATS_ONE_WAY_ANOVA(cust_income_level, amount_sold, 'F_RATIO') f_ratio,
       STATS_ONE_WAY_ANOVA(cust_income_level, amount_sold, 'SIG') p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id
  GROUP BY cust_gender
  ORDER BY cust_gender;

C    F_RATIO    P_VALUE
- ---------- ----------
F 5.59536943 4.7840E-09
M  9.2865001 6.7139E-17
```

STATS_T_TEST_*

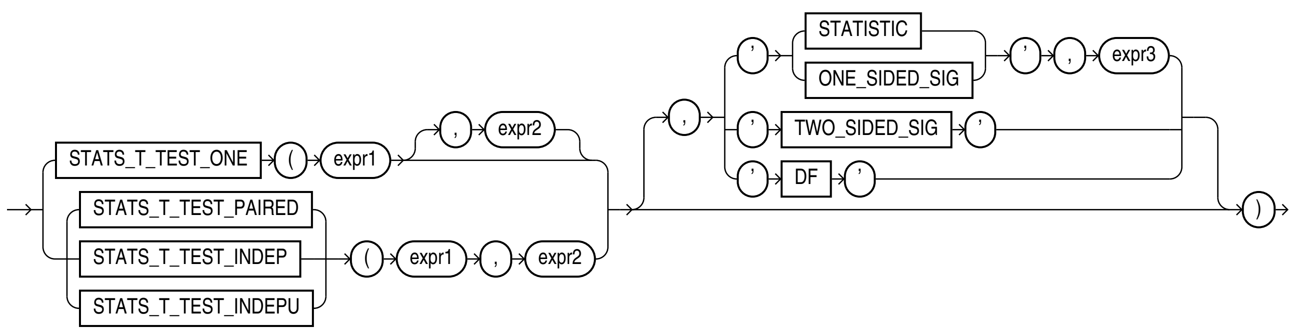
The *t*-test functions are:

- `STATS_T_TEST_ONE`: A one-sample *t*-test
- `STATS_T_TEST_PAIRED`: A two-sample, paired *t*-test (also known as a crossed *t*-test)

- `STATS_T_TEST_INDEP`: A *t*-test of two independent groups with the same variance (pooled variances)
- `STATS_T_TEST_INDEPU`: A *t*-test of two independent groups with unequal variance (unpooled variances)

Syntax

stats_t_test::=



Purpose

The *t*-test measures the significance of a difference of means. You can use it to compare the means of two groups or the means of one group with a constant. Each *t*-test function takes two expression arguments, although the second expression is optional for the one-sample function (`STATS_T_TEST_ONE`). Each *t*-test function takes an optional third argument, which lets you specify the meaning of the `NUMBER` value returned by the function, as shown in Table 7-9. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `STATS_T_TEST_*` functions

Table 7-9 STATS_T_TEST_* Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *t*
`'DF'`	Degree of freedom
`'ONE_SIDED_SIG'`	One-tailed significance of *t*
`'TWO_SIDED_SIG'`	Two-tailed significance of *t*

The two independent `STATS_T_TEST_*` functions can take a fourth argument (*`expr3`*) if the third argument is specified as `'STATISTIC'` or `'ONE_SIDED_SIG'`. In this case, *`expr3`* indicates which value of *`expr1`* is the high value, or the value whose rejection region is the upper tail.

The significance of the observed value of *t* is the probability that the value of *t* would have been obtained by chance—a number between 0 and 1. The smaller the value, the more significant the difference between the means. One-sided significance is always respect to the upper tail. For one-sample and paired *t*-test, the high value is the first expression. For independent *t*-test, the high value is the one specified by `expr3`.

The degree of freedom depends on the type of *t*-test that resulted in the observed value of *t*. For example, for a one-sample *t*-test (`STATS_T_TEST_ONE`), the degree of freedom is the number of observations in the sample minus 1.

STATS_T_TEST_ONE

In the `STATS_T_TEST_ONE` function, `expr1` is the sample and `expr2` is the constant mean against which the sample mean is compared. For this *t*-test only, `expr2` is optional; the constant mean defaults to 0. This function obtains the value of *t* by dividing the difference between the sample mean and the known mean by the standard error of the mean (rather than the standard error of the difference of the means, as for `STATS_T_TEST_PAIRED`).

STATS_T_TEST_ONE Example

The following example determines the significance of the difference between the average list price and the constant value 60:

```
SELECT AVG(prod_list_price) group_mean,
       STATS_T_TEST_ONE(prod_list_price, 60, 'STATISTIC') t_observed,
       STATS_T_TEST_ONE(prod_list_price, 60) two_sided_p_value
  FROM sh.products;

GROUP_MEAN T_OBSERVED TWO_SIDED_P_VALUE
---------- ---------- -----------------
139.545556 2.32107746        .023158537
```

STATS_T_TEST_PAIRED

In the `STATS_T_TEST_PAIRED` function, `expr1` and `expr2` are the two samples whose means are being compared. This function obtains the value of *t* by dividing the difference between the sample means by the standard error of the difference of the means (rather than the standard error of the mean, as for `STATS_T_TEST_ONE`).

STATS_T_TEST_INDEP and STATS_T_TEST_INDEPU

In the `STATS_T_TEST_INDEP` and `STATS_T_TEST_INDEPU` functions, `expr1` is the grouping column and `expr2` is the sample of values. The pooled variances version (`STATS_T_TEST_INDEP`) tests whether the means are the same or different for two distributions that have similar variances. The unpooled variances version (`STATS_T_TEST_INDEPU`) tests whether the means are the same or different even if the two distributions are known to have significantly different variances.

Before using these functions, it is advisable to determine whether the variances of the samples are significantly different. If they are, then the data may come from distributions with different shapes, and the difference of the means may not be very useful. You can perform an *f*-test to determine the difference of the variances. If they are not significantly different, use `STATS_T_TEST_INDEP`. If they are significantly different, use STATS_T_TEST_INDEPU. Refer to STATS_F_TEST for information on performing an *f*-test.

STATS_T_TEST_INDEP Example

The following example determines the significance of the difference between the average sales to men and women where the distributions are assumed to have similar (pooled) variances:

```
SELECT SUBSTR(cust_income_level, 1, 22) income_level,
      AVG(DECODE(cust_gender, 'M', amount_sold, null)) sold_to_men,
      AVG(DECODE(cust_gender, 'F', amount_sold, null)) sold_to_women,
      STATS_T_TEST_INDEP(cust_gender, amount_sold, 'STATISTIC', 'F') t_observed,
      STATS_T_TEST_INDEP(cust_gender, amount_sold) two_sided_p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id
  GROUP BY ROLLUP(cust_income_level)
  ORDER BY income_level, sold_to_men, sold_to_women, t_observed;

INCOME_LEVEL           SOLD_TO_MEN SOLD_TO_WOMEN T_OBSERVED TWO_SIDED_P_VALUE
---------------------- ----------- ------------- ---------- -----------------
A: Below 30,000          105.28349    99.4281447 -1.9880629        .046811482
B: 30,000 - 49,999       102.59651    109.829642 3.04330875        .002341053
C: 50,000 - 69,999      105.627588    110.127931 2.36148671        .018204221
D: 70,000 - 89,999      106.630299     110.47287 2.28496443        .022316997
E: 90,000 - 109,999     103.396741    101.610416 -1.2544577        .209677823
F: 110,000 - 129,999     106.76476    105.981312 -.60444998        .545545304
G: 130,000 - 149,999    108.877532     107.31377 -.85298245        .393671218
H: 150,000 - 169,999    110.987258    107.152191 -1.9062363        .056622983
I: 170,000 - 189,999    102.808238     107.43556 2.18477851        .028908566
J: 190,000 - 249,999    108.040564    115.343356 2.58313425        .009794516
K: 250,000 - 299,999    112.377993    108.196097 -1.4107871        .158316973
L: 300,000 and above    120.970235    112.216342 -2.0642868        .039003862
                        107.121845     113.80441 .686144393        .492670059
                        106.663769    107.276386 1.08013499        .280082357
14 rows selected.
```

STATS_T_TEST_INDEPU Example

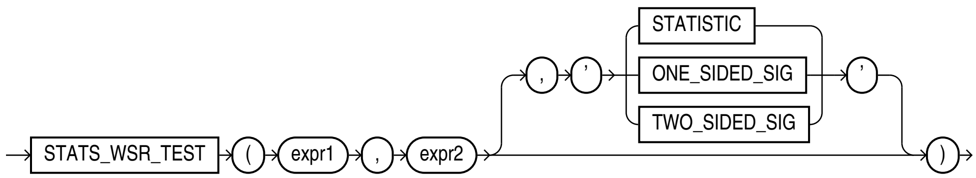
The following example determines the significance of the difference between the average sales to men and women where the distributions are known to have significantly different (unpooled) variances:

```
SELECT SUBSTR(cust_income_level, 1, 22) income_level,
       AVG(DECODE(cust_gender, 'M', amount_sold, null)) sold_to_men,
       AVG(DECODE(cust_gender, 'F', amount_sold, null)) sold_to_women,
       STATS_T_TEST_INDEPU(cust_gender, amount_sold, 'STATISTIC', 'F') t_observed,
       STATS_T_TEST_INDEPU(cust_gender, amount_sold) two_sided_p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id
  GROUP BY ROLLUP(cust_income_level)
  ORDER BY income_level, sold_to_men, sold_to_women, t_observed;

INCOME_LEVEL           SOLD_TO_MEN SOLD_TO_WOMEN T_OBSERVED TWO_SIDED_P_VALUE
---------------------- ----------- ------------- ---------- -----------------
A: Below 30,000          105.28349    99.4281447 -2.0542592        .039964704
B: 30,000 - 49,999       102.59651    109.829642 2.96922332        .002987742
C: 50,000 - 69,999      105.627588    110.127931  2.3496854        .018792277
D: 70,000 - 89,999      106.630299     110.47287 2.26839281        .023307831
E: 90,000 - 109,999     103.396741    101.610416 -1.2603509        .207545662
F: 110,000 - 129,999     106.76476    105.981312 -.60580011        .544648553
```

```
G: 130,000 - 149,999    108.877532     107.31377 -.85219781         .394107755
H: 150,000 - 169,999    110.987258    107.152191 -1.9451486         .051762624
I: 170,000 - 189,999    102.808238     107.43556 2.14966921         .031587875
J: 190,000 - 249,999    108.040564    115.343356 2.54749867         .010854966
K: 250,000 - 299,999    112.377993    108.196097 -1.4115514         .158091676
L: 300,000 and above    120.970235    112.216342 -2.0726194         .038225611
                        107.121845     113.80441 .689462437         .490595765
                        106.663769    107.276386 1.07853782         .280794207
14 rows selected.
```

STATS_WSR_TEST

Syntax



Purpose

STATS_WSR_TEST is a Wilcoxon Signed Ranks test of paired samples to determine whether the median of the differences between the samples is significantly different from zero. The absolute values of the differences are ordered and assigned ranks. Then the null hypothesis states that the sum of the ranks of the positive differences is equal to the sum of the ranks of the negative differences.

This function takes two required arguments: *expr1* and *expr2* are the two samples being analyzed. The optional third argument lets you specify the meaning of the NUMBER value returned by this function, as shown in Table 7-10. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is 'TWO_SIDED_SIG'.

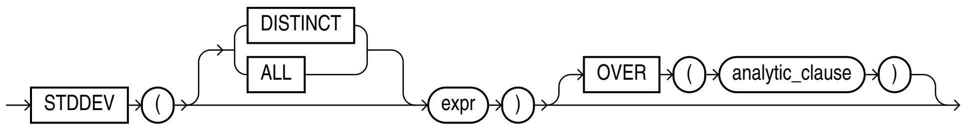
Table 7-10 STATS_WSR_TEST_* Return Values

Argument	Return Value Meaning
'STATISTIC'	The observed value of *Z*
'ONE_SIDED_SIG'	One-tailed significance of *Z*
'TWO_SIDED_SIG'	Two-tailed significance of *Z*

One-sided significance is always with respect to the upper tail. The high value (the value whose rejection region is the upper tail) is *expr1*.

STDDEV

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`STDDEV` returns the sample standard deviation of *`expr`*, a set of numbers. You can use it as both an aggregate and analytic function. It differs from `STDDEV_SAMP` in that `STDDEV` returns zero when it has only 1 row of input data, whereas `STDDEV_SAMP` returns null.

Oracle Database calculates the standard deviation as the square root of the variance defined for the `VARIANCE` aggregate function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

> **See Also:**
>
> - "Aggregate Functions ", VARIANCE , and STDDEV_SAMP
> - "About SQL Expressions " for information on valid forms of *`expr`*

Aggregate Examples

The following example returns the standard deviation of the salaries in the sample `hr.employees` table:

```
SELECT STDDEV(salary) "Deviation"
   FROM employees;
```

```
 Deviation
----------
3909.36575
```

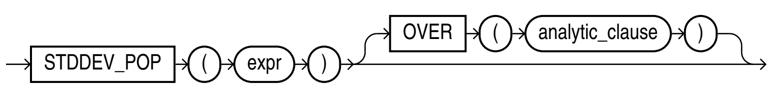
Analytic Examples

The query in the following example returns the cumulative standard deviation of the salaries in Department 80 in the sample table `hr.employees`, ordered by `hire_date`:

```
SELECT last_name, salary,
   STDDEV(salary) OVER (ORDER BY hire_date) "StdDev"
   FROM employees
   WHERE department_id = 30
   ORDER BY last_name, salary, "StdDev";

LAST_NAME                     SALARY     StdDev
------------------------- ---------- ----------
Baida                           2900 4035.26125
Colmenares                      2500 3362.58829
Himuro                          2600  3649.2465
Khoo                            3100 5586.14357
Raphaely                       11000          0
Tobias                          2800  4650.0896
```

STDDEV_POP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`STDDEV_POP` computes the population standard deviation and returns the square root of the population variance. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

This function is the same as the square root of the VAR_POP function. When VAR_POP returns null, this function returns null.

See Also:

- "Aggregate Functions " and VAR_POP
- "About SQL Expressions " for information on valid forms of *expr*

Aggregate Example

The following example returns the population and sample standard deviations of the amount of sales in the sample table sh.sales:

```
SELECT STDDEV_POP(amount_sold) "Pop",
   STDDEV_SAMP(amount_sold) "Samp"
   FROM sales;

       Pop       Samp
---------- ----------
896.355151 896.355592
```

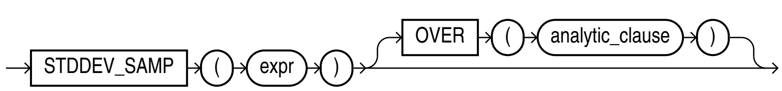
Analytic Example

The following example returns the population standard deviations of salaries in the sample hr.employees table by department:

```
SELECT department_id, last_name, salary,
   STDDEV_POP(salary) OVER (PARTITION BY department_id) AS pop_std
   FROM employees
   ORDER BY department_id, last_name, salary, pop_std;

DEPARTMENT_ID LAST_NAME                     SALARY    POP_STD
------------- ------------------------- ---------- ----------
           10 Whalen                          4400          0
           20 Fay                             6000       3500
           20 Hartstein                      13000       3500
           30 Baida                           2900  3069.6091
. . .
          100 Urman                           7800 1644.18166
          110 Gietz                           8300       1850
          110 Higgins                        12000       1850
              Grant                           7000          0
```

STDDEV_SAMP

Syntax



See Also:

"Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

STDDEV_SAMP computes the cumulative sample standard deviation and returns the square root of the sample variance. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

See Also:

Table 2-8 for more information on implicit conversion

This function is same as the square root of the VAR_SAMP function. When VAR_SAMP returns null, this function returns null.

See Also:

- "Aggregate Functions " and VAR_SAMP
- "About SQL Expressions " for information on valid forms of *expr*

Aggregate Example

Refer to the aggregate example for STDDEV_POP .

Analytic Example

The following example returns the sample standard deviation of salaries in the employees table by department:

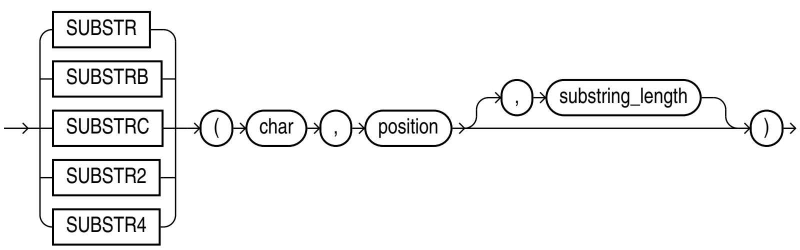
```
SELECT department_id, last_name, hire_date, salary,
   STDDEV_SAMP(salary) OVER (PARTITION BY department_id
      ORDER BY hire_date
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_sdev
   FROM employees
   ORDER BY department_id, last_name, hire_date, salary, cum_sdev;

DEPARTMENT_ID LAST_NAME       HIRE_DATE     SALARY   CUM_SDEV
------------- --------------- --------- ---------- ----------
           10 Whalen          17-SEP-03       4400
           20 Fay             17-AUG-05       6000 4949.74747
           20 Hartstein       17-FEB-04      13000
           30 Baida           24-DEC-05       2900 4035.26125
           30 Colmenares      10-AUG-07       2500 3362.58829
```

```
        30 Himuro           15-NOV-06       2600  3649.2465
        30 Khoo             18-MAY-03       3100 5586.14357
        30 Raphaely         07-DEC-02      11000
. . .
       100 Greenberg        17-AUG-02      12008  2126.9772
       100 Popp             07-DEC-07       6900 1804.13155
       100 Sciarra          30-SEP-05       7700 1929.76233
       100 Urman            07-MAR-06       7800 1788.92504
       110 Gietz            07-JUN-02       8300 2621.95194
       110 Higgins          07-JUN-02      12008
           Grant            24-MAY-07       7000
```

SUBSTR

Syntax

substr::=



Purpose

The SUBSTR functions return a portion of *char*, beginning at character *position*, *substring_length* characters long. SUBSTR calculates lengths using characters as defined by the input character set. SUBSTRB uses bytes instead of characters. SUBSTRC uses Unicode complete characters. SUBSTR2 uses UCS2 code points. SUBSTR4 uses UCS4 code points.

- If *position* is 0, then it is treated as 1.
- If *position* is positive, then Oracle Database counts from the beginning of *char* to find the first character.
- If *position* is negative, then Oracle counts backward from the end of *char*.
- If *substring_length* is omitted, then Oracle returns all characters to the end of *char*. If *substring_length* is less than 1, then Oracle returns null.

char can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The exceptions are SUBSTRC, SUBSTR2, and SUBSTR4, which do not allow *char* to be a CLOB or NCLOB. Both *position* and *substring_length* must be of data type NUMBER, or any data type that can be implicitly converted to NUMBER, and must resolve to an integer. The return value is the same data type as *char*. Floating-point numbers passed as arguments to SUBSTR are automatically converted to integers.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for more information about `SUBSTR` functions and length semantics in different locales
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `SUBSTR`

Examples

The following example returns several specified substrings of "ABCDEFG":

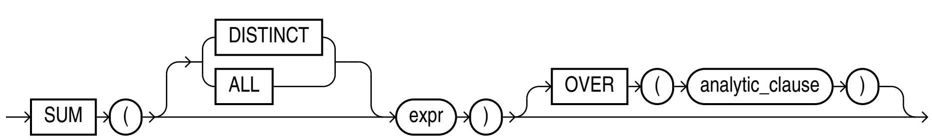
```
SELECT SUBSTR('ABCDEFG',3,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF

SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2) "Substring with bytes"
     FROM DUAL;

Substring with bytes
--------------------
CD
```

SUM

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

SUM returns the sum of values of *expr*. You can use it as an aggregate or analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *expr* and "Aggregate Functions "

Aggregate Example

The following example calculates the sum of all salaries in the sample hr.employees table:

```
SELECT SUM(salary) "Total"
     FROM employees;

     Total
----------
    691400
```

Analytic Example

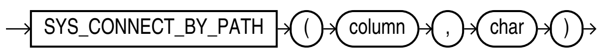
The following example calculates, for each manager in the sample table hr.employees, a cumulative total of salaries of employees who answer to that manager that are equal to or less than the current salary. You can see that Raphaely and Cambrault have the same cumulative total. This is because Raphaely and Cambrault have the identical salaries, so Oracle Database adds together their salary values and applies the same cumulative total to both rows.

```
SELECT manager_id, last_name, salary,
   SUM(salary) OVER (PARTITION BY manager_id ORDER BY salary
   RANGE UNBOUNDED PRECEDING) l_csum
   FROM employees
   ORDER BY manager_id, last_name, salary, l_csum;

MANAGER_ID LAST_NAME                     SALARY     L_CSUM
---------- ------------------------- ---------- ----------
       100 Cambrault                      11000      68900
       100 De Haan                        17000     155400
       100 Errazuriz                      12000      80900
       100 Fripp                           8200      36400
```

```
        100 Hartstein                          13000      93900
        100 Kaufling                            7900      20200
        100 Kochhar                            17000     155400
        100 Mourgos                             5800       5800
        100 Partners                           13500     107400
        100 Raphaely                           11000      68900
        100 Russell                            14000     121400
. . .
        149 Hutton                              8800      39000
        149 Johnson                             6200       6200
        149 Livingston                          8400      21600
        149 Taylor                              8600      30200
        201 Fay                                 6000       6000
        205 Gietz                               8300       8300
            King                               24000      24000
```

SYS_CONNECT_BY_PATH

Syntax



Purpose

`SYS_CONNECT_BY_PATH` is valid only in hierarchical queries. It returns the path of a column value from root to node, with column values separated by *`char`* for each row returned by `CONNECT BY` condition.

Both *`column`* and *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The string returned is of `VARCHAR2` data type and is in the same character set as *`column`*.

> **See Also:**
>
> - "Hierarchical Queries " for more information about hierarchical queries and `CONNECT BY` conditions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `SYS_CONNECT_BY_PATH`

Examples

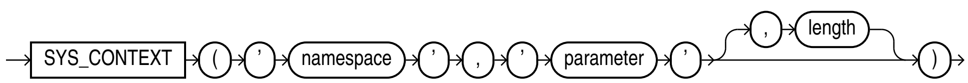
The following example returns the path of employee names from employee `Kochhar` to all employees of `Kochhar` (and their employees):

```
SELECT LPAD(' ', 2*level-1)||SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   START WITH last_name = 'Kochhar'
   CONNECT BY PRIOR employee_id = manager_id;

Path
------------------------------
```

```
      /Kochhar/Greenberg/Chen
      /Kochhar/Greenberg/Faviet
      /Kochhar/Greenberg/Popp
      /Kochhar/Greenberg/Sciarra
      /Kochhar/Greenberg/Urman
      /Kochhar/Higgins/Gietz
    /Kochhar/Baer
    /Kochhar/Greenberg
    /Kochhar/Higgins
    /Kochhar/Mavris
    /Kochhar/Whalen
  /Kochhar
```

SYS_CONTEXT

Syntax



Purpose

SYS_CONTEXT returns the value of *parameter* associated with the context *namespace* at the current instant. You can use this function in both SQL and PL/SQL statements. SYS_CONTEXT must be executed locally.

For *namespace* and *parameter*, you can specify either a string or an expression that resolves to a string designating a namespace or an attribute. If you specify literal arguments for *namespace* and *parameter*, and you are using SYS_CONTEXT explicitly in a SQL statement—rather than in a PL/SQL function that in turn is in mentioned in a SQL statement—then Oracle Database evaluates SYS_CONTEXT only once per SQL statement execution for each call site that invokes the SYS_CONTEXT function.

The context *namespace* must already have been created, and the associated *parameter* and its value must also have been set using the DBMS_SESSION.set_context procedure. The *namespace* must be a valid identifier. The *parameter* name can be any string. It is not case sensitive, but it cannot exceed 30 bytes in length.

The data type of the return value is VARCHAR2. The default maximum size of the return value is 256 bytes. You can override this default by specifying the optional *length* parameter, which must be a NUMBER or a value that can be implicitly converted to NUMBER. The valid range of values is 1 to 4000 bytes. If you specify an invalid value, then Oracle Database ignores it and uses the default.

Oracle provides the following built-in namespaces:

- USERENV - Describes the current session. The predefined parameters of namespace USERENV are listed in Table 7-11.
- SYS_SESSION_ROLES - Indicates whether a specified role is currently enabled for the session. Oracle Database evaluates the SYS_SESSION_ROLES context for the current user, and assumes the defining user's role when it evaluates SYS_SESSION_ROLES within a definer's rights procedure or function. Use invoker's rights, procedures or functions, and/or code based access control (CBAC) as an alternative.

> **See Also:**
>
> - *Oracle Database Security Guide* for information on using the application context feature in your application development
> - CREATE CONTEXT for information on creating user-defined context namespaces
> - *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_SESSION.set_context` procedure
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `SYS_CONTEXT`

Examples

The following statement returns the name of the user who logged onto the database:

```
CONNECT OE
Enter password: password

SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
   FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
---------------------------------------
OE
```

The following example queries the `SESSION_ROLES` data dictionary view to show that `RESOURCE` is the only role currently enabled for the session. It then uses the `SYS_CONTEXT` function to show that the `RESOURCE` role is currently enabled for the session and the `DBA` role is not.

```
CONNECT OE
Enter password: password

SELECT role FROM session_roles;

ROLE
--------
RESOURCE

SELECT SYS_CONTEXT('SYS_SESSION_ROLES', 'RESOURCE')
  FROM DUAL

SYS_CONTEXT('SYS_SESSION_ROLES','RESOURCE')
--------------------------------------
TRUE

SELECT SYS_CONTEXT('SYS_SESSION_ROLES', 'DBA')
  FROM DUAL;

SYS_CONTEXT('SYS_SESSION_ROLES','DBA')
--------------------------------------
FALSE
```

> **Note:**
>
> For simplicity in demonstrating this feature, these examples do not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

The following hypothetical example returns the group number that was set as the value for the attribute `group_no` in the PL/SQL package that was associated with the context `hr_apps` when `hr_apps` was created:

```
SELECT SYS_CONTEXT ('hr_apps', 'group_no') "User Group"
   FROM DUAL;
```

Table 7-11 Predefined Parameters of Namespace USERENV

Parameter	Return Value
`ACTION`	Identifies the position in the module (application name) and is set through the `DBMS_APPLICATION_INFO` package or OCI.
`AUDITED_CURSORID`	Returns the cursor ID of the SQL that triggered the audit. This parameter is not valid in a fine-grained auditing environment. If you specify it in such an environment, then Oracle Database always returns null.
`AUTHENTICATED_IDENTITY`	Returns the identity used in authentication. In the list that follows, the type of user is followed by the value returned: • Kerberos-authenticated enterprise user: kerberos principal name • Kerberos-authenticated external user : kerberos principal name; same as the schema name • SSL-authenticated enterprise user: the DN in the user's PKI certificate • SSL-authenticated external user: the DN in the user's PKI certificate • Password-authenticated enterprise user: nickname; same as the login name • Password-authenticated database user: the database username; same as the schema name • OS-authenticated external user: the external operating system user name • Radius-authenticated external user: the schema name • Proxy with DN : Oracle Internet Directory DN of the client • Proxy with certificate: certificate DN of the client • Proxy with username: database user name if client is a local database user; nickname if client is an enterprise user. • SYSDBA/SYSOPER using Password File: login name • SYSDBA/SYSOPER using OS authentication: operating system user name
`AUTHENTICATION_DATA`	Data being used to authenticate the login user. For X.503 certificate authenticated sessions, this field returns the context of the certificate in HEX2 format. **Note:** You can change the return value of the `AUTHENTICATION_DATA` attribute using the *`length`* parameter of the syntax. Values of up to 4000 are accepted. This is the only attribute of `USERENV` for which Oracle Database implements such a change.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
AUTHENTICATION_METHOD	Returns the method of authentication. In the list that follows, the type of user is followed by the method returned: • Password-authenticated enterprise user, local database user, or user with the SYSDBA or SYSOPER administrative privilege using a password file; proxy with username using password: PASSWORD • Kerberos-authenticated enterprise user or external user (with no administrative privileges): KERBEROS • Kerberos-authenticated enterprise user (with administrative privileges): KERBEROS_GLOBAL • Kerberos-authenticated external user (with administrative privileges): KERBEROS_EXTERNAL • SSL-authenticated enterprise or external user (with no administrative privileges): SSL • SSL-authenticated enterprise user (with administrative privileges): SSL_GLOBAL • SSL-authenticated external user (with administrative privileges): SSL_EXTERNAL • Radius-authenticated external user: RADIUS • OS-authenticated external user or use with the SYSDBA or SYSOPER administrative privilege: OS • Proxy with certificate, DN, or username without using password: NONE • Background process (job queue slave process): JOB • Parallel Query Slave process: PQ_SLAVE For non-administrative connections, you can use IDENTIFICATION_TYPE to distinguish between external and enterprise users when the authentication method is PASSWORD, KERBEROS, or SSL. For administrative connections, AUTHENTICATION_METHOD is sufficient for the PASSWORD, SSL_EXTERNAL, and SSL_GLOBAL authentication methods.
BG_JOB_ID	Job ID of the current session if it was established by an Oracle Database background process. Null if the session was not established by a background process.
CDB_NAME	If queried while connected to a multitenant container database (CDB), returns the name of the CDB. Otherwise, returns null.
CLIENT_IDENTIFIER	Returns an identifier that is set by the application through the DBMS_SESSION.SET_IDENTIFIER procedure, the OCI attribute OCI_ATTR_CLIENT_IDENTIFIER, or Oracle Dynamic Monitoring Service (DMS). This attribute is used by various database components to identify lightweight application users who authenticate as the same database user.
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.
CLIENT_PROGRAM_NAME	The name of the program used for the database session.
CON_ID	If queried while connected to a CDB, returns the current container ID. Otherwise, returns 0.
CON_NAME	If queried while connected to a CDB, returns the current container name. Otherwise, returns the name of the database as specified in the DB_NAME initialization parameter.
CURRENT_BIND	The bind variables for fine-grained auditing. You can specify this attribute only inside the event handler for the fine-grained auditing feature.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
CURRENT_EDITION_ID	The identifier of the current edition.
CURRENT_EDITION_NAME	The name of the current edition.
CURRENT_SCHEMA	The name of the currently active default schema. This value may change during the duration of a session through use of an ALTER SESSION SET CURRENT_SCHEMA statement. This may also change during the duration of a session to reflect the owner of any active definer's rights object. When used directly in the body of a view definition, this returns the default schema used when executing the cursor that is using the view; it does not respect views used in the cursor as being definer's rights. **Note**: Oracle recommends against issuing the SQL statement ALTER SESSION SET CURRENT_SCHEMA from within all types of stored PL/SQL units except logon triggers.
CURRENT_SCHEMAID	Identifier of the currently active default schema.
CURRENT_SQL CURRENT_SQL*n*	CURRENT_SQL returns the first 4K bytes of the current SQL that triggered the fine-grained auditing event. The CURRENT_SQL*n* attributes return subsequent 4K-byte increments, where *n* can be an integer from 1 to 7, inclusive. CURRENT_SQL1 returns bytes 4K to 8K; CURRENT_SQL2 returns bytes 8K to 12K, and so forth. You can specify these attributes only inside the event handler for the fine-grained auditing feature.
CURRENT_SQL_LENGTH	The length of the current SQL statement that triggers fine-grained audit or row-level security (RLS) policy functions or event handlers. You can specify this attribute only inside the event handler for the fine-grained auditing feature.
CURRENT_USER	The name of the database user whose privileges are currently active. This may change during the duration of a database session as Real Application Security sessions are attached or detached, or to reflect the owner of any active definer's rights object. When no definer's rights object is active, CURRENT_USER returns the same value as SESSION_USER. When used directly in the body of a view definition, this returns the user that is executing the cursor that is using the view; it does not respect views used in the cursor as being definer's rights. For enterprise users, returns schema. If a Real Application Security user is currently active, returns user XS$NULL. **See Also:** *Oracle Database 2 Day + Security Guide* for more information on user XS$NULL
CURRENT_USERID	The identifier of the database user whose privileges are currently active.
DATABASE_ROLE	The database role using the SYS_CONTEXT function with the USERENV namespace. The role is one of the following: PRIMARY, PHYSICAL STANDBY, LOGICAL STANDBY, SNAPSHOT STANDBY.
DB_DOMAIN	Domain of the database as specified in the DB_DOMAIN initialization parameter.
DB_NAME	Name of the database in the current context.
DB_SUPPLEMENTAL_LOG_LEVEL	If supplemental logging is enabled, returns a string containing the list of enabled supplemental logging levels. Possible values are: ALL_COLUMN, FOREIGN_KEY, MINIMAL, PRIMARY_KEY, PROCEDURAL, and UNIQUE_INDEX. If supplemental logging is not enabled, returns null.
DB_UNIQUE_NAME	Name of the database as specified in the DB_UNIQUE_NAME initialization parameter.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
`DBLINK_INFO`	Returns the source of a database link session. Specifically, it returns a string of the form: `SOURCE_GLOBAL_NAME=`*`dblink_src_global_name`*`, DBLINK_NAME=`*`dblink_name`*`, SOURCE_AUDIT_SESSIONID=`*`dblink_src_audit_sessionid`* where: • *`dblink_src_global_name`* is the unique global name of the source database • *`dblink_name`* is the name of the database link on the source database • *`dblink_src_audit_sessionid`* is the audit session ID of the session on the source database that initiated the connection to the remote database using *`dblink_name`*
`ENTRYID`	The current audit entry number. The audit entryid sequence is shared between fine-grained audit records and regular audit records. You cannot use this attribute in distributed SQL statements. The correct auditing entry identifier can be seen only through an audit handler for standard or fine-grained audit.
`ENTERPRISE_IDENTITY`	Returns the user's enterprise-wide identity: • For enterprise users: the Oracle Internet Directory DN. • For external users: the external identity (Kerberos principal name, Radius schema names, OS user name, Certificate DN). • For local users and SYSDBA/SYSOPER logins: `NULL`. The value of the attribute differs by proxy method: • For a proxy with DN: the Oracle Internet Directory DN of the client • For a proxy with certificate: the certificate DN of the client for external users; the Oracle Internet Directory DN for global users • For a proxy with username: the Oracle Internet Directory DN if the client is an enterprise users; Null if the client is a local database user.
`FG_JOB_ID`	If queried from within a job that was created using the `DBMS_JOB` package: Returns the job ID of the current session if it was established by a client foreground process. Null if the session was not established by a foreground process. Otherwise: Returns 0.
`GLOBAL_CONTEXT_MEMORY`	Returns the number being used in the System Global Area by the globally accessed context.
`GLOBAL_UID`	Returns the global user ID from Oracle Internet Directory for Enterprise User Security (EUS) logins; returns null for all other logins.
`HOST`	Name of the host machine from which the client has connected.
`IDENTIFICATION_TYPE`	Returns the way the user's schema was created in the database. Specifically, it reflects the `IDENTIFIED` clause in the `CREATE`/`ALTER USER` syntax. In the list that follows, the syntax used during schema creation is followed by the identification type returned: • `IDENTIFIED BY` *`password`*: `LOCAL` • `IDENTIFIED EXTERNALLY`: `EXTERNAL` • `IDENTIFIED GLOBALLY`: `GLOBAL SHARED` • `IDENTIFIED GLOBALLY AS` *`DN`*: `GLOBAL PRIVATE`
`INSTANCE`	The instance identification number of the current instance.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

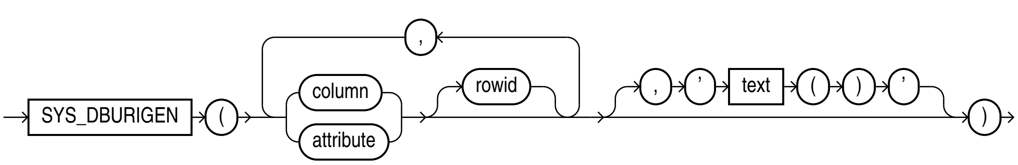
Parameter	Return Value
INSTANCE_NAME	The name of the instance.
IP_ADDRESS	IP address of the machine from which the client is connected. If the client and server are on the same machine and the connection uses IPv6 addressing, then ::1 is returned.
IS_APPLY_SERVER	Returns TRUE if queried from within a SQL Apply server in a logical standby database. Otherwise, returns FALSE.
IS_DG_ROLLING_UPGRADE	Returns TRUE if a rolling upgrade of the database software in a Data Guard configuration, initiated by way of the DBMS_ROLLING package, is active. Otherwise, returns FALSE.
ISDBA	Returns TRUE if the user has been authenticated as having DBA privileges either through the operating system or through a password file.
LANG	The abbreviated name for the language, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	The language and territory currently used by your session, along with the database character set, in this form: language_territory.characterset
MODULE	The application name (module) set through the DBMS_APPLICATION_INFO package or OCI.
NETWORK_PROTOCOL	Network protocol being used for communication, as specified in the 'PROTOCOL=*protocol*' portion of the connect string.
NLS_CALENDAR	The current calendar of the current session.
NLS_CURRENCY	The currency of the current session.
NLS_DATE_FORMAT	The date format for the session.
NLS_DATE_LANGUAGE	The language used for expressing dates.
NLS_SORT	BINARY or the linguistic sort basis.
NLS_TERRITORY	The territory of the current session.
ORACLE_HOME	The full path name for the Oracle home directory.
OS_USER	Operating system user name of the client process that initiated the database session.
PLATFORM_SLASH	The slash character that is used as the file path delimiter for your platform.
POLICY_INVOKER	The invoker of row-level security (RLS) policy functions.
PROXY_ENTERPRISE_IDENTITY	Returns the Oracle Internet Directory DN when the proxy user is an enterprise user.
PROXY_USER	Name of the database user who opened the current session on behalf of SESSION_USER.
PROXY_USERID	Identifier of the database user who opened the current session on behalf of SESSION_USER.
SCHEDULER_JOB	Returns Y if the current session belongs to a foreground job or background job. Otherwise, returns N.
SERVER_HOST	The host name of the machine on which the instance is running.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
SESSION_NAME	The name of the service to which a given session is connected.
SESSION_DEFAULT_COLLATION	The default collation for the session, which is set by the ALTER SESSION SET DEFAULT_COLLATION ... statement.
SESSION_EDITION_ID	The identifier of the session edition.
SESSION_EDITION_NAME	The name of the session edition.
SESSION_USER	The name of the session user (the user who logged on). This may change during the duration of a database session as Real Application Security sessions are attached or detached. For enterprise users, returns the schema. For other users, returns the database user name. If a Real Application Security session is currently attached to the database session, returns user XS$NULL. **See Also:** *Oracle Database 2 Day + Security Guide* for more information on user XS$NULL
SESSION_USERID	The identifier of the session user (the user who logged on).
SESSIONID	The auditing session identifier. You cannot use this attribute in distributed SQL statements.
SID	The session ID.
STATEMENTID	The auditing statement identifier. STATEMENTID represents the number of SQL statements audited in a given session. You cannot use this attribute in distributed SQL statements. The correct auditing statement identifier can be seen only through an audit handler for standard or fine-grained audit.
TERMINAL	The operating system identifier for the client of the current session. In distributed SQL statements, this attribute returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations. (The return length of this parameter may vary by operating system.)
UNIFIED_AUDIT_SESSIONID	If queried while connected to a database that uses unified auditing or mixed mode auditing, returns the unified audit session ID. If queried while connected to a database that uses traditional auditing, returns null.

SYS_DBURIGEN

Syntax



Purpose

SYS_DBURIGen takes as its argument one or more columns or attributes, and optionally a rowid, and generates a URL of data type DBURIType to a particular column or row object. You can then use the URL to retrieve an XML document from the database.

All columns or attributes referenced must reside in the same table. They must perform the function of a primary key. They need not actually match the primary key of the table, but they must reference a unique value. If you specify multiple columns, then all but the final column identify the row in the database, and the last column specified identifies the column within the row.

By default the URL points to a formatted XML document. If you want the URL to point only to the text of the document, then specify the optional '`text()`'.

> **Note:**
>
> In this XML context, the lowercase `text` is a keyword, not a syntactic placeholder.

If the table or view containing the columns or attributes does not have a schema specified in the context of the query, then Oracle Database interprets the table or view name as a public synonym.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for information on the `DBURIType` data type and XML documents in the database

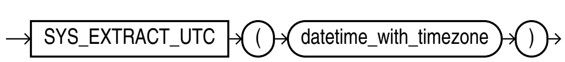
Examples

The following example uses the `SYS_DBURIGen` function to generate a URL of data type `DBURIType` to the `email` column of the row in the sample table `hr.employees` where the `employee_id` = 206:

```
SELECT SYS_DBURIGEN(employee_id, email)
   FROM employees
   WHERE employee_id = 206;

SYS_DBURIGEN(EMPLOYEE_ID,EMAIL)(URL, SPARE)
--------------------------------------------------------------------
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''206'']/EMAIL', NULL)
```

SYS_EXTRACT_UTC

Syntax



Purpose

`SYS_EXTRACT_UTC` extracts the UTC (Coordinated Universal Time—formerly Greenwich Mean Time) from a datetime value with time zone offset or time zone region name. If a time zone is not specified, then the datetime is associated with the session time zone.

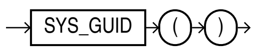
Examples

The following example extracts the UTC from a specified datetime:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2000-03-28 11:30:00.00 -08:00')
   FROM DUAL;

SYS_EXTRACT_UTC(TIMESTAMP'2000-03-2811:30:00.00-08:00')
-----------------------------------------------------------------
28-MAR-00 07.30.00 PM
```

SYS_GUID

Syntax



Purpose

`SYS_GUID` generates and returns a globally unique identifier (`RAW` value) made up of 16 bytes. On most platforms, the generated identifier consists of a host identifier, a process or thread identifier of the process or thread invoking the function, and a nonrepeating value (sequence of bytes) for that process or thread.

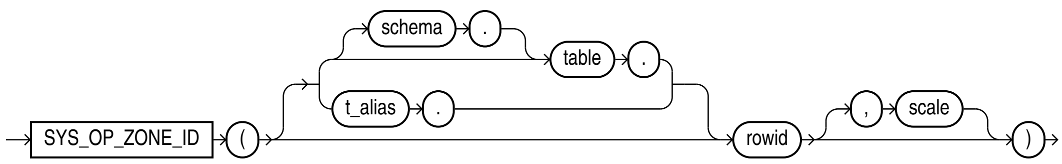
Examples

The following example adds a column to the sample table `hr.locations`, inserts unique identifiers into each row, and returns the 32-character hexadecimal representation of the 16-byte `RAW` value of the global unique identifier:

```
ALTER TABLE locations ADD (uid_col RAW(16));

UPDATE locations SET uid_col = SYS_GUID();

SELECT location_id, uid_col FROM locations
   ORDER BY location_id, uid_col;

LOCATION_ID UID_COL
----------- ----------------------------------------------------------------
       1000 09F686761827CF8AE040578CB20B7491
       1100 09F686761828CF8AE040578CB20B7491
       1200 09F686761829CF8AE040578CB20B7491
       1300 09F68676182ACF8AE040578CB20B7491
       1400 09F68676182BCF8AE040578CB20B7491
       1500 09F68676182CCF8AE040578CB20B7491
. . .
```

SYS_OP_ZONE_ID

Syntax



Purpose

SYS_OP_ZONE_ID takes as its argument a rowid and returns a zone ID. The rowid identifies a row in a table. The zone ID identifies the set of contiguous disk blocks, called the zone, that contains the row. The function returns a NUMBER value.

The SYS_OP_ZONE_ID function is used when creating a zone map with the CREATE MATERIALIZED ZONEMAP statement. You must specify SYS_OP_ZONE_ID in the SELECT and GROUP BY clauses of the defining subquery of the zone map.

For *rowid*, specify the ROWID pseudocolumn of the fact table of the zone map.

Use *schema* and *table* to specify the schema and name of the fact table, or *t_alias* to specify the table alias for the fact table. The specification of these parameters depends on the FROM clause in the defining subquery of the zone map:

- If the FROM clause specifies a table alias for the fact table, then you must also specify the table alias (*t_alias*) in SYS_OP_ZONE_ID.
- If the FROM clause does not specify a table alias for the fact table, then use *table* to specify the name of the fact table. You can use the *schema* qualifier if the fact table is in a schema other than your own. If you omit *schema*, then the database assumes the fact table is in your own schema. If the FROM clause specifies only one table (the fact table) then you need not specify *schema* or *table*.

The optional *scale* parameter represents the scale of the zone map. It is not necessary to specify this parameter because, by default, SYS_OP_ZONE_ID uses the scale of the zone map being created. If you do specify *scale*, then it must match the scale of the zone map being created. Refer to the SCALE clause of CREATE MATERIALIZED ZONEMAP for information on specifying the scale of a zone map.

> **See Also:**
>
> CREATE MATERIALIZED ZONEMAP for more information on creating zone maps

Examples

The following example uses the SYS_OP_ZONE_ID function when creating a basic zone map that tracks the column time_id of the fact table sales. The scale of the zone map is the default value of 10. Therefore, the SYS_OP_ZONE_ID function will default to a scale value of 10.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
AS
  SELECT SYS_OP_ZONE_ID(rowid), MIN(time_id), MAX(time_id)
  FROM sales
  GROUP BY SYS_OP_ZONE_ID(rowid);
```

The following example is similar to the previous example, except that the scale of the zone map being created is specified as 8. Therefore, the `SYS_OP_ZONE_ID` function will default to a scale value of 8.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
SCALE 8
AS
  SELECT SYS_OP_ZONE_ID(rowid), MIN(time_id), MAX(time_id)
  FROM sales
  GROUP BY SYS_OP_ZONE_ID(rowid);
```

The following example returns an error because the scale of the zone map being created is specified as 8, which does not match the *`scale`* argument of 12 specified in the `SYS_OP_ZONE_ID` function.

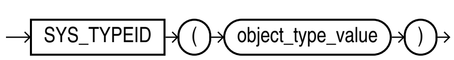
```
CREATE MATERIALIZED ZONEMAP sales_zmap
SCALE 8
AS
  SELECT SYS_OP_ZONE_ID(rowid,12), MIN(time_id), MAX(time_id)
  FROM sales
  GROUP BY SYS_OP_ZONE_ID(rowid,12);
```

The following example creates a join zone map. The fact table is `sales` and the dimension tables are `products` and `customers`. Because the table alias `s` is specified for the fact table in the `FROM` clause, the table alias *`s`* is also specified in the `SYS_OP_ZONE_ID` function.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
AS
  SELECT SYS_OP_ZONE_ID(s.rowid),
         MIN(prod_category), MAX(prod_category),
         MIN(country_id), MAX(country_id)
  FROM sales s, products p, customers c
  WHERE s.prod_id = p.prod_id(+) AND
        s.cust_id = c.cust_id(+)
  GROUP BY SYS_OP_ZONE_ID(s.rowid);
```

SYS_TYPEID

Syntax



Purpose

`SYS_TYPEID` returns the typeid of the most specific type of the operand. This value is used primarily to identify the type-discriminant column underlying a substitutable column. For example, you can use the value returned by `SYS_TYPEID` to build an index on the type-discriminant column.

You can use this function only on object type operands. All final root object types—final types not belonging to a type hierarchy—have a null typeid. Oracle Database assigns to all types belonging to a type hierarchy a unique non-null typeid.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information on typeids

Examples

The following examples use the tables `persons` and `books`, which are created in "Substitutable Table and Column Examples". The first query returns the most specific types of the object instances stored in the `persons` table.

```
SELECT name, SYS_TYPEID(VALUE(p)) "Type_id" FROM persons p;

NAME                      Type_id
------------------------- --------------------------------
Bob                       01
Joe                       02
Tim                       03
```

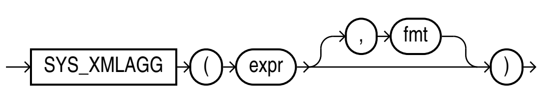
The next query returns the most specific types of authors stored in the table `books`:

```
SELECT b.title, b.author.name, SYS_TYPEID(author)
   "Type_ID" FROM books b;

TITLE                     AUTHOR.NAME          Type_ID
------------------------- -------------------- -------------------
An Autobiography          Bob                  01
Business Rules            Joe                  02
Mixing School and Work    Tim                  03
```

You can use the `SYS_TYPEID` function to create an index on the type-discriminant column of a table. For an example, see "Indexing on Substitutable Columns: Examples".

SYS_XMLAGG

Syntax



Purpose

`SYS_XMLAgg` aggregates all of the XML documents or fragments represented by *`expr`* and produces a single XML document. It adds a new enclosing element with a default name `ROWSET`. If you want to format the XML document differently, then specify *`fmt`*, which is an instance of the `XMLFormat` object.

> **See Also:**
>
> SYS_XMLGEN and "XML Format Model " for using the attributes of the `XMLFormat` type to format `SYS_XMLAgg` results

Examples

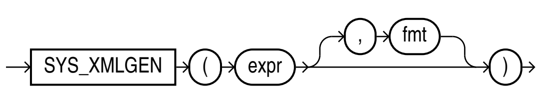
The following example uses the `SYS_XMLGen` function to generate an XML document for each row of the sample table `employees` where the employee's last name begins with the letter R, and then aggregates all of the rows into a single XML document in the default enclosing element `ROWSET`:

```
SELECT SYS_XMLAGG(SYS_XMLGEN(last_name)) XMLAGG
   FROM employees
   WHERE last_name LIKE 'R%'
   ORDER BY xmlagg;

XMLAGG
--------------------------------------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
<LAST_NAME>Rajs</LAST_NAME>
<LAST_NAME>Raphaely</LAST_NAME>
<LAST_NAME>Rogers</LAST_NAME>
<LAST_NAME>Russell</LAST_NAME>
</ROWSET>
```

SYS_XMLGEN

> **Note:**
>
> The `SYS_XMLGen` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the SQL/XML generation functions instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`SYS_XMLGen` takes an expression that evaluates to a particular row and column of the database, and returns an instance of type `XMLType` containing an XML document. The *`expr`* can be a scalar value, a user-defined type, or an `XMLType` instance.

- If *`expr`* is a scalar value, then the function returns an XML element containing the scalar value.

- If *expr* is a type, then the function maps the user-defined type attributes to XML elements.
- If *expr* is an `XMLType` instance, then the function encloses the document in an XML element whose default tag name is `ROW`.

By default the elements of the XML document match the elements of *expr*. For example, if *expr* resolves to a column name, then the enclosing XML element will be the same column name. If you want to format the XML document differently, then specify *fmt*, which is an instance of the `XMLFormat` object.

> **See Also:**
>
> "XML Format Model " for a description of the `XMLFormat` type and how to use its attributes to format `SYS_XMLGen` results

Examples

The following example retrieves the employee email ID from the sample table `oe.employees` where the `employee_id` value is 205, and generates an instance of an `XMLType` containing an XML document with an `EMAIL` element.

```
SELECT SYS_XMLGEN(email)
   FROM employees
   WHERE employee_id = 205;

SYS_XMLGEN(EMAIL)
-------------------------------------------------------------------
<?xml version="1.0"?>
<EMAIL>SHIGGINS</EMAIL>
```

SYSDATE

Syntax

Purpose

`SYSDATE` returns the current date and time set for the operating system on which the database server resides. The data type of the returned value is `DATE`, and the format returned depends on the value of the `NLS_DATE_FORMAT` initialization parameter. The function requires no arguments. In distributed SQL statements, this function returns the date and time set for the operating system of your local database. You cannot use this function in the condition of a `CHECK` constraint.

> **Note:**
>
> The `FIXED_DATE` initialization parameter enables you to set a constant date and time that `SYSDATE` will always return instead of the current date and time. This parameter is useful primarily for testing. Refer to *Oracle Database Reference* for more information on the `FIXED_DATE` initialization parameter.

Examples

The following example returns the current operating system date and time:

```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "NOW"
     FROM DUAL;

NOW
-------------------
04-13-2001 09:45:51
```

SYSTIMESTAMP

Syntax

Purpose

`SYSTIMESTAMP` returns the system date, including fractional seconds and time zone, of the system on which the database resides. The return type is `TIMESTAMP WITH TIME ZONE`.

Examples

The following example returns the system timestamp:

```
SELECT SYSTIMESTAMP FROM DUAL;

SYSTIMESTAMP
------------------------------------------------------------------
28-MAR-00 12.38.55.538741 PM -08:00
```

The following example shows how to explicitly specify fractional seconds:

```
SELECT TO_CHAR(SYSTIMESTAMP, 'SSSSS.FF') FROM DUAL;

TO_CHAR(SYSTIME
---------------
55615.449255
```

The following example returns the current timestamp in a specified time zone:

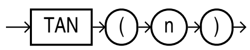
```
SELECT SYSTIMESTAMP AT TIME ZONE 'UTC' FROM DUAL;

SYSTIMESTAMPATTIMEZONE'UTC'
```

```
--------------------------------------------------------------------------
08-07-21 20:39:52,743557 UTC
```

The output format in this example depends on the `NLS_TIMESTAMP_TZ_FORMAT` for the session.

TAN

Syntax



Purpose

`TAN` returns the tangent of *n* (an angle expressed in radians).

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

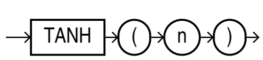
Examples

The following example returns the tangent of 135 degrees:

```
SELECT TAN(135 * 3.14159265359/180)
   "Tangent of 135 degrees"  FROM DUAL;

Tangent of 135 degrees
----------------------
                   - 1
```

TANH

Syntax



Purpose

`TANH` returns the hyperbolic tangent of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns BINARY_DOUBLE. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

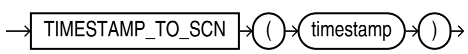
Examples

The following example returns the hyperbolic tangent of .5:

```
SELECT TANH(.5) "Hyperbolic tangent of .5"
   FROM DUAL;

Hyperbolic tangent of .5
------------------------
              .462117157
```

TIMESTAMP_TO_SCN

Syntax



Purpose

TIMESTAMP_TO_SCN takes as an argument a timestamp value and returns the approximate system change number (SCN) associated with that timestamp. The returned value is of data type NUMBER. This function is useful any time you want to know the SCN associated with a particular timestamp.

> **Note:**
>
> The association between an SCN and a timestamp when the SCN is generated is remembered by the database for a limited period of time. This period is the maximum of the auto-tuned undo retention period, if the database runs in the Automatic Undo Management mode, and the retention times of all flashback archives in the database, but no less than 120 hours. The time for the association to become obsolete elapses only when the database is open. An error is returned if the timestamp specified for the argument to TIMESTAMP_TO_SCN is too old.

> **See Also:**
>
> SCN_TO_TIMESTAMP for information on converting SCNs to timestamp

Examples

The following example inserts a row into the `oe.orders` table and then uses `TIMESTAMP_TO_SCN` to determine the system change number of the insert operation. (The actual SCN returned will differ on each system.)

```
INSERT INTO orders (order_id, order_date, customer_id, order_total)
   VALUES (5000, SYSTIMESTAMP, 188, 2345);
1 row created.

COMMIT;
Commit complete.

SELECT TIMESTAMP_TO_SCN(order_date) FROM orders
   WHERE order_id = 5000;

TIMESTAMP_TO_SCN(ORDER_DATE)
----------------------------
                      574100
```

TO_APPROX_COUNT_DISTINCT

Syntax

Purpose

`TO_APPROX_COUNT_DISTINCT` takes as its input a detail containing information about an approximate distinct value count, and converts it to a `NUMBER` value.

For *`detail`*, specify a detail of type `BLOB`, which was created by the `APPROX_COUNT_DISTINCT_DETAIL` function or the `APPROX_COUNT_DISTINCT_AGG` function.

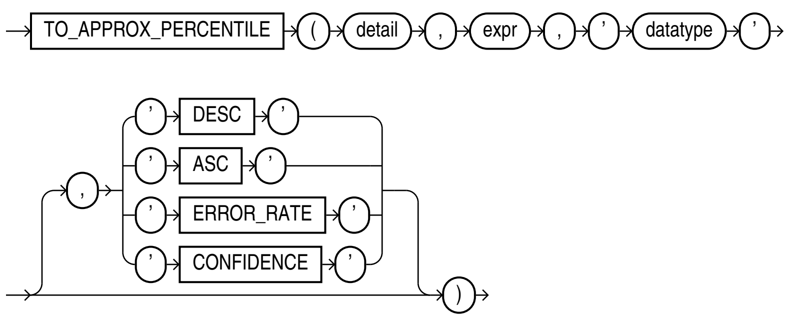
> **See Also:**
>
> - APPROX_COUNT_DISTINCT_DETAIL
> - TO_APPROX_COUNT_DISTINCT

Examples

Refer to TO_APPROX_COUNT_DISTINCT: Examples for examples of using the `TO_APPROX_COUNT_DISTINCT` function in conjunction with the `APPROX_COUNT_DISTINCT_DETAIL` and `APPROX_COUNT_DISTINCT_AGG` functions.

TO_APPROX_PERCENTILE

Syntax



Purpose

TO_APPROX_PERCENTILE takes as its input a detail containing approximate percentile information, a percentile value, and a sort specification, and returns an approximate interpolated value that would fall into that percentile value with respect to the sort specification.

For *detail*, specify a detail of type BLOB, which was created by the APPROX_PERCENTILE_DETAIL function or the APPROX_PERCENTILE_AGG function.

For *expr*, specify a percentile value, which must evaluate to a numeric value between 0 and 1. If you specify the ERROR_RATE or CONFIDENCE clause, then the percentile value does not apply. In this case, for *expr* you must specify null or a numeric value between 0 and 1. However, the value will be ignored.

For *datatype*, specify the data type of the approximate percentile information in the detail. This is the data type of the expression supplied to the APPROX_PERCENTILE_DETAIL function that originated the detail. Valid data types are NUMBER, BINARY_FLOAT, BINARY_DOUBLE, DATE, TIMESTAMP, INTERVAL YEAR TO MONTH, and INTERVAL DAY TO SECOND.

DESC | ASC

Specify the sort specification for the interpolation. Specify DESC for a descending sort order, or ASC for an ascending sort order. ASC is the default.

ERROR_RATE | CONFIDENCE

These clauses let you determine the accuracy of the percentile evaluation of the detail. If you specify one of these clauses, then instead of returning the approximate interpolated value, the function returns a decimal value from 0 to 1, inclusive, which represents one of the following values:

- If you specify ERROR_RATE, then the return value represents the error rate of the percentile evaluation for the detail.
- If you specify CONFIDENCE, then the return value represents the confidence level for the error rate returned when you specify ERROR_RATE.

If you specify ERROR_RATE or CONFIDENCE, then the percentile value *expr* is ignored.

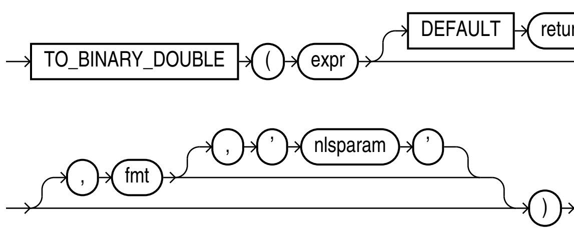
> **See Also:**
>
> - APPROX_PERCENTILE_DETAIL
> - APPROX_PERCENTILE_AGG

Examples

Refer to APPROX_PERCENTILE_AGG: Examples for examples of using the TO_APPROX_PERCENTILE function in conjunction with the APPROX_PERCENTILE_DETAIL and APPROX_PERCENTILE_AGG functions.

TO_BINARY_DOUBLE

Syntax



Purpose

TO_BINARY_DOUBLE converts *expr* to a double-precision floating-point number.

- *expr* can be any expression that evaluates to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE, or null. If *expr* is BINARY_DOUBLE, then the function returns *expr*. If *expr* evaluates to null, then the function returns null. Otherwise, the function converts *expr* to a BINARY_DOUBLE value.
- The optional DEFAULT *return_value* ON CONVERSION ERROR clause allows you to specify the value returned by this function if an error occurs while converting *expr* to BINARY_DOUBLE. This clause has no effect if an error occurs while evaluating *expr*. The *return_value* can be an expression or a bind variable, and must evaluate to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE, or null. The function converts *return_value* to BINARY_DOUBLE in the same way it converts *expr* to BINARY_DOUBLE. If *return_value* cannot be converted to BINARY_DOUBLE, then the function returns an error.
- The optional '*fmt*' and '*nlsparam*' arguments serve the same purpose as for the TO_NUMBER function. If you specify these arguments, then *expr* and *return_value*, if specified, must each be a character string or null. If either is a character string, then the function uses the *fmt* and *nlsparam* arguments to convert the character string to a BINARY_DOUBLE value.

If *expr* or *return_value* evaluate to the following character strings, then the function converts them as follows:

- The case-insensitive string 'INF' is converted to positive infinity.
- The case-insensitive string '-INF' is converted to negative identity.
- The case-insensitive string 'NaN' is converted to NaN (not a number).

You cannot use a floating-point number format element (F, f, D, or d) in a character string *expr*.

Conversions from character strings or NUMBER to BINARY_DOUBLE can be inexact, because the NUMBER and character types use decimal precision to represent the numeric value, and BINARY_DOUBLE uses binary precision.

Conversions from BINARY_FLOAT to BINARY_DOUBLE are exact.

> **See Also:**
>
> TO_CHAR (number) and "Floating-Point Numbers "

Examples

The examples that follow are based on a table with three columns, each with a different numeric data type:

```
CREATE TABLE float_point_demo
  (dec_num NUMBER(10,2), bin_double BINARY_DOUBLE, bin_float BINARY_FLOAT);

INSERT INTO float_point_demo
  VALUES (1234.56,1234.56,1234.56);

SELECT * FROM float_point_demo;

   DEC_NUM BIN_DOUBLE  BIN_FLOAT
---------- ---------- ----------
   1234.56 1.235E+003 1.235E+003
```

The following example converts a value of data type NUMBER to a value of data type BINARY_DOUBLE:

```
SELECT dec_num, TO_BINARY_DOUBLE(dec_num)
  FROM float_point_demo;

   DEC_NUM TO_BINARY_DOUBLE(DEC_NUM)
---------- -------------------------
   1234.56                1.235E+003
```

The following example compares extracted dump information from the dec_num and bin_double columns:

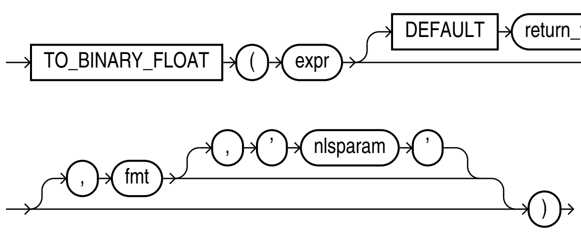
```
SELECT DUMP(dec_num) "Decimal",
   DUMP(bin_double) "Double"
   FROM float_point_demo;

Decimal                     Double
--------------------------- ---------------------------------------------
Typ=2 Len=4: 194,13,35,57   Typ=101 Len=8: 192,147,74,61,112,163,215,10
```

The following example returns the default value of 0 because the specified expression cannot be converted to a BINARY_DOUBLE value:

```
SELECT TO_BINARY_DOUBLE('2oo' DEFAULT 0 ON CONVERSION ERROR) "Value"
  FROM DUAL;

     Value
----------
         0
```

TO_BINARY_FLOAT

Syntax



Purpose

TO_BINARY_FLOAT converts *expr* to a single-precision floating-point number.

- *expr* can be any expression that evaluates to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE, or null. If *expr* is BINARY_FLOAT, then the function returns *expr*. If *expr* evaluates to null, then the function returns null. Otherwise, the function converts *expr* to a BINARY_FLOAT value.
- The optional DEFAULT *return_value* ON CONVERSION ERROR clause allows you to specify the value returned by this function if an error occurs while converting *expr* to BINARY_FLOAT. This clause has no effect if an error occurs while evaluating *expr*. The *return_value* can be an expression or a bind variable, and must evaluate to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE, or null. The function converts *return_value* to BINARY_FLOAT in the same way it converts *expr* to BINARY_FLOAT. If *return_value* cannot be converted to BINARY_FLOAT, then the function returns an error.
- The optional '*fmt*' and '*nlsparam*' arguments serve the same purpose as for the TO_NUMBER function. If you specify these arguments, then *expr* and *return_value*, if specified, must each be a character string or null. If either is a character string, then the function uses the *fmt* and *nlsparam* arguments to convert the character string to a BINARY_FLOAT value.

If *expr* or *return_value* evaluate to the following character strings, then the function converts them as follows:

- The case-insensitive string 'INF' is converted to positive infinity.
- The case-insensitive string '-INF' is converted to negative identity.
- The case-insensitive string 'NaN' is converted to NaN (not a number).

You cannot use a floating-point number format element (`F`, `f`, `D`, or `d`) in a character string *`expr`*.

Conversions from character strings or `NUMBER` to `BINARY_FLOAT` can be inexact, because the `NUMBER` and character types use decimal precision to represent the numeric value and `BINARY_FLOAT` uses binary precision.

Conversions from `BINARY_DOUBLE` to `BINARY_FLOAT` are inexact if the `BINARY_DOUBLE` value uses more bits of precision than supported by the `BINARY_FLOAT`.

> **See Also:**
>
> TO_CHAR (number) and "Floating-Point Numbers "

Examples

Using table `float_point_demo` created for TO_BINARY_DOUBLE , the following example converts a value of data type `NUMBER` to a value of data type `BINARY_FLOAT`:

```
SELECT dec_num, TO_BINARY_FLOAT(dec_num)
  FROM float_point_demo;

   DEC_NUM TO_BINARY_FLOAT(DEC_NUM)
---------- ------------------------
   1234.56                1.235E+003
```

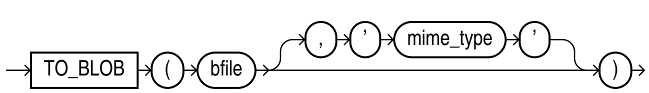
The following example returns the default value of `0` because the specified expression cannot be converted to a `BINARY_FLOAT` value:

```
SELECT TO_BINARY_FLOAT('2oo' DEFAULT 0 ON CONVERSION ERROR) "Value"
  FROM DUAL;

     Value
----------
         0
```

TO_BLOB (bfile)

Syntax

***to_blob_bfile*::=**



Purpose

`TO_BLOB` (bfile) converts a `BFILE` value to a `BLOB` value.

For *`mime_type`*, specify the MIME type to be set on the `BLOB` value returned by this function. If you omit *`mime_type`*, then a MIME type will not be set on the `BLOB` value.

Example

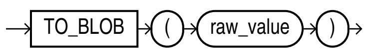
The following hypothetical example returns the BLOB of a BFILE column value media_col in table media_tab. It sets the MIME type to JPEG on the resulting BLOB.

```
SELECT TO_BLOB(media_col, 'JPEG') FROM media_tab;
```

TO_BLOB (raw)

Syntax

to_blob::=



Purpose

TO_BLOB (raw) converts LONG RAW and RAW values to BLOB values.

From within a PL/SQL package, you can use TO_BLOB (raw) to convert RAW and BLOB values to BLOB.

Examples

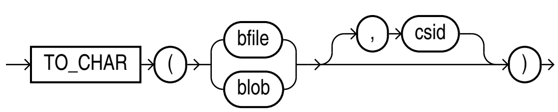
The following hypothetical example returns the BLOB of a RAW column value:

```
SELECT TO_BLOB(raw_column) blob FROM raw_table;

BLOB
-----------------------
00AADD343CDBBD
```

TO_CHAR (bfile|blob)

Syntax

to_char_bfile_blob::=



Purpose

TO_CHAR (bfile|blob) converts BFILE or BLOB data to the database character set. The value returned is always VARCHAR2. If the value returned is too large to fit into the VARCHAR2 data type, then the data is truncated.

For *csid*, specify the character set ID of the BFILE or BLOB data. If the character set of the BFILE or BLOB data is the database character set, then you can specify a value of 0 for *csid*, or omit *csid* altogether.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Example

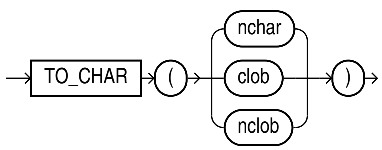
The following hypothetical example takes as its input a BFILE column media_col in table media_tab, which uses the character set with ID 873. The example returns a VARCHAR2 value that uses the database character set.

```
SELECT TO_CHAR(media_col, 873) FROM media_tab;
```

TO_CHAR (character)

Syntax

***to_char_char*::=**



Purpose

TO_CHAR (character) converts NCHAR, NVARCHAR2, CLOB, or NCLOB data to the database character set. The value returned is always VARCHAR2.

When you use this function to convert a character LOB into the database character set, if the LOB value to be converted is larger than the target type, then the database returns an error.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example interprets a simple string as character data:

```
SELECT TO_CHAR('01110') FROM DUAL;

TO_CH
-----
01110
```

Compare this example with the first example for TO_CHAR (number) .

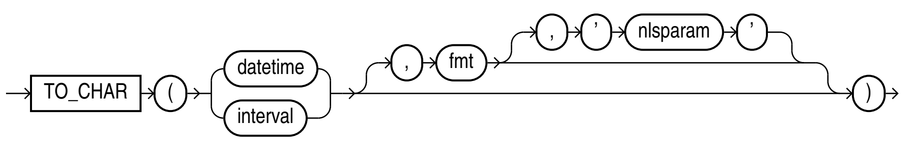
The following example converts some `CLOB` data from the `pm.print_media` table to the database character set:

```
SELECT TO_CHAR(ad_sourcetext) FROM print_media
      WHERE product_id = 2268;

TO_CHAR(AD_SOURCETEXT)
--------------------------------------------------------------------
******************************
TIGER2 2268...Standard Hayes Compatible Modem
Product ID: 2268
The #1 selling modem in the universe! Tiger2's modem includes call management
and Internet voicing. Make real-time full duplex phone calls at the same time
you're online.
**********************************
```

TO_CHAR (datetime)

Syntax

to_char_date::=



Purpose

`TO_CHAR` (datetime) converts a datetime or interval value of `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL DAY TO SECOND`, or `INTERVAL YEAR TO MONTH` data type to a value of `VARCHAR2` data type in the format specified by the date format *`fmt`*. If you omit *`fmt`*, then *`date`* is converted to a `VARCHAR2` value as follows:

- `DATE` values are converted to values in the default date format.
- `TIMESTAMP` and `TIMESTAMP WITH LOCAL TIME ZONE` values are converted to values in the default timestamp format.
- `TIMESTAMP WITH TIME ZONE` values are converted to values in the default timestamp with time zone format.
- Interval values are converted to the numeric representation of the interval literal.

Refer to "Format Models " for information on datetime formats.

The *`'nlsparam'`* argument specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit *'nlsparam'*, then this function uses the default date language for your session.

> **See Also:**
>
> "Security Considerations for Data Conversion"

You can use this function in conjunction with any of the XML functions to generate a date in the database format rather than the XML Schema standard format.

> **See Also:**
>
> - *Oracle XML DB Developer's Guide* for information about formatting of XML dates and timestamps, including examples
> - "XML Functions " for a listing of the XML functions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example uses this table:

```
CREATE TABLE date_tab (
   ts_col      TIMESTAMP,
   tsltz_col   TIMESTAMP WITH LOCAL TIME ZONE,
   tstz_col    TIMESTAMP WITH TIME ZONE);
```

The example shows the results of applying `TO_CHAR` to different `TIMESTAMP` data types. The result for a `TIMESTAMP WITH LOCAL TIME ZONE` column is sensitive to session time zone, whereas the results for the `TIMESTAMP` and `TIMESTAMP WITH TIME ZONE` columns are not sensitive to session time zone:

```
ALTER SESSION SET TIME_ZONE = '-8:00';
INSERT INTO date_tab VALUES (
   TIMESTAMP'1999-12-01 10:00:00',
   TIMESTAMP'1999-12-01 10:00:00',
   TIMESTAMP'1999-12-01 10:00:00');
INSERT INTO date_tab VALUES (
   TIMESTAMP'1999-12-02 10:00:00 -8:00',
   TIMESTAMP'1999-12-02 10:00:00 -8:00',
   TIMESTAMP'1999-12-02 10:00:00 -8:00');

SELECT TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS ts_date,
   TO_CHAR(tstz_col, 'DD-MON-YYYY HH24:MI:SSxFF TZH:TZM') AS tstz_date
   FROM date_tab
   ORDER BY ts_date, tstz_date;

TS_DATE                        TSTZ_DATE
------------------------------ -------------------------------------
01-DEC-1999 10:00:00.000000    01-DEC-1999 10:00:00.000000 -08:00
```

```
02-DEC-1999 10:00:00.000000    02-DEC-1999 10:00:00.000000 -08:00

SELECT SESSIONTIMEZONE,
   TO_CHAR(tsltz_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS tsltz
   FROM date_tab
   ORDER BY sessiontimezone, tsltz;

SESSIONTIM TSLTZ
---------- ------------------------------
-08:00     01-DEC-1999 10:00:00.000000
-08:00     02-DEC-1999 10:00:00.000000

ALTER SESSION SET TIME_ZONE = '-5:00';
SELECT TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS ts_col,
   TO_CHAR(tstz_col, 'DD-MON-YYYY HH24:MI:SSxFF TZH:TZM') AS tstz_col
   FROM date_tab
   ORDER BY ts_col, tstz_col;

TS_COL                         TSTZ_COL
------------------------------ -------------------------------------
01-DEC-1999 10:00:00.000000    01-DEC-1999 10:00:00.000000 -08:00
02-DEC-1999 10:00:00.000000    02-DEC-1999 10:00:00.000000 -08:00

SELECT SESSIONTIMEZONE,
TO_CHAR(tsltz_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS tsltz_col
   FROM date_tab
   ORDER BY sessiontimezone, tsltz_col;
  2    3    4
SESSIONTIM TSLTZ_COL
---------- ------------------------------
-05:00     01-DEC-1999 13:00:00.000000
-05:00     02-DEC-1999 13:00:00.000000
```

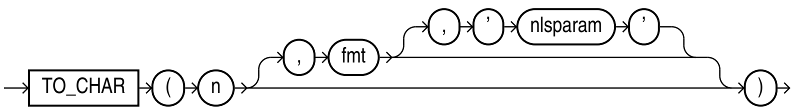
The following example converts an interval literal into a text literal:

```
SELECT TO_CHAR(INTERVAL '123-2' YEAR(3) TO MONTH) FROM DUAL;

TO_CHAR
-------
+123-02
```

TO_CHAR (number)

Syntax

to_char_number::=



Purpose

`TO_CHAR` (number) converts *`n`* to a value of `VARCHAR2` data type, using the optional number format *`fmt`*. The value *`n`* can be of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`. If you omit *`fmt`*, then *`n`* is converted to a `VARCHAR2` value exactly long enough to hold its significant digits.

If *n* is negative, then the sign is applied after the format is applied. Thus `TO_CHAR(-1, '$9')` returns -$1, rather than $-1.

Refer to "Format Models " for information on number formats.

The *'nlsparam'* argument specifies these characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- International currency symbol

This argument can have this form:

```
'NLS_NUMERIC_CHARACTERS = ''dg''
   NLS_CURRENCY = ''text''
   NLS_ISO_CURRENCY = territory '
```

The characters *d* and *g* represent the decimal character and group separator, respectively. They must be different single-byte characters. Within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit *'nlsparam'* or any one of the parameters, then this function uses the default parameter values for your session.

> **See Also:**
>
> - "Security Considerations for Data Conversion"
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following statement uses implicit conversion to combine a string and a number into a number:

```
SELECT TO_CHAR('01110' + 1) FROM DUAL;

TO_C
----
1111
```

Compare this example with the first example for TO_CHAR (character) .

In the next example, the output is blank padded to the left of the currency symbol. In the optional number format fmt, `L` designates local currency symbol and `MI` designates a trailing minus sign. See Table 2-15 for a complete listing of number format elements. The example shows the output in a session in which the session parameter `NLS_TERRITORY` is set to `AMERICA`.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
     FROM DUAL;

Amount
--------------
  $10,000.00-
```

In the next example, NLS_CURRENCY specifies the string to use as the local currency symbol for the L number format element. NLS_NUMERIC_CHARACTERS specifies comma as the character to use as the decimal separator for the D number format element and period as the character to use as the group separator for the G number format element. These characters are expected in many countries, for example in Germany.

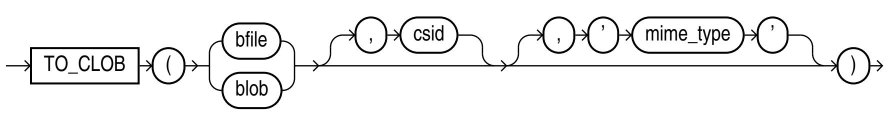
```
SELECT TO_CHAR(-10000,'L99G999D99MI',
   'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_CURRENCY = ''AusDollars'' ') "Amount"
     FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

In the next example, NLS_ISO_CURRENCY instructs the database to use the international currency symbol for the territory of POLAND for the C number format element:

```
SELECT TO_CHAR(-10000,'99G999D99C',
   'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_ISO_CURRENCY=POLAND') "Amount"
     FROM DUAL;

Amount
-----------------
    -10.000,00PLN
```

TO_CLOB (bfile|blob)

Syntax



Purpose

TO_CLOB (bfile|blob) converts BFILE or BLOB data to the database character set and returns the data as a CLOB value.

For *csid*, specify the character set ID of the BFILE or BLOB data. If the character set of the BFILE or BLOB data is the database character set, then you can specify a value of 0 for *csid*, or omit *csid* altogether.

For *mime_type*, specify the MIME type to be set on the CLOB value returned by this function. If you omit *mime_type*, then a MIME type will not be set on the CLOB value.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

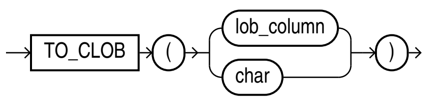
Example

The following hypothetical example returns the `CLOB` of a `BFILE` column value `docu` in table `media_tab`, which uses the character set with ID 873. It sets the MIME type to `text/xml` for the resulting `CLOB`.

```
SELECT TO_CLOB(docu, 873, 'text/xml') FROM media_tab;
```

TO_CLOB (character)

Syntax



Purpose

`TO_CLOB` (character) converts `NCLOB` values in a LOB column or other character strings to `CLOB` values. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. Oracle Database executes this function by converting the underlying LOB data from the national character set to the database character set.

From within a PL/SQL package, you can use the `TO_CLOB` (character) function to convert `RAW`, `CHAR`, `VARCHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB` values to `CLOB` or `NCLOB` values.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

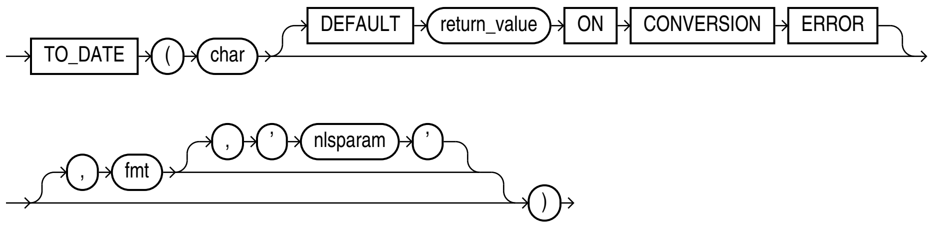
Examples

The following statement converts `NCLOB` data from the sample `pm.print_media` table to `CLOB` and inserts it into a `CLOB` column, replacing existing data in that column.

```
UPDATE PRINT_MEDIA
   SET AD_FINALTEXT = TO_CLOB (AD_FLTEXTN);
```

TO_DATE

Syntax



Purpose

`TO_DATE` converts *`char`* to a value of `DATE` data type.

For *`char`*, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

> **Note:**
>
> This function does not convert data to any of the other datetime data types. For information on other datetime conversions, refer to TO_TIMESTAMP , TO_TIMESTAMP_TZ , TO_DSINTERVAL , and TO_YMINTERVAL .

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting *`char`* to `DATE`. This clause has no effect if an error occurs while evaluating *`char`*. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type, or null. The function converts *`return_value`* to `DATE` using the same method it uses to convert *`char`* to `DATE`. If *`return_value`* cannot be converted to `DATE`, then the function returns an error.

The *`fmt`* is a datetime model format specifying the format of *`char`*. If you omit *`fmt`*, then *`char`* must be in the default date format. The default date format is determined implicitly by the `NLS_TERRITORY` initialization parameter or can be set explicitly by the `NLS_DATE_FORMAT` parameter. If *`fmt`* is `J`, for Julian, then *`char`* must be an integer.

> **Caution:**
>
> It is good practice *always* to specify a format mask (*`fmt`*) with `TO_DATE`, as shown in the examples in the section that follows. When it is used without a format mask, the function is valid *only* if *`char`* uses the same format as is determined by the `NLS_TERRITORY` or `NLS_DATE_FORMAT` parameters. Furthermore, the function may not be stable across databases unless the explicit format mask is specified to avoid dependencies.

The *'nlsparam'* argument specifies the language of the text string that is being converted to a date. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

Do not use the TO_DATE function with a DATE value for the *char* argument. The first two digits of the returned DATE value can differ from the original *char*, depending on *fmt* or the default date format.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Datetime Format Models " and "Data Type Comparison Rules " for more information

Examples

The following example converts a character string into a date:

```
SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;

TO_DATE('
---------
15-JAN-89
```

The value returned reflects the default date format if the NLS_TERRITORY parameter is set to 'AMERICA'. Different NLS_TERRITORY values result in different default date formats:

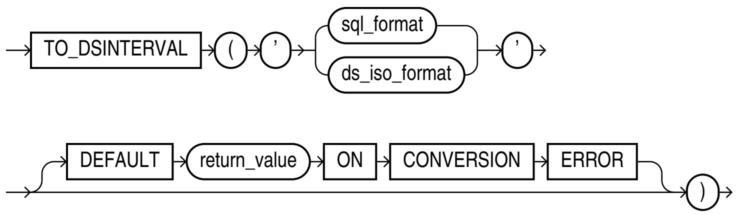
```
ALTER SESSION SET NLS_TERRITORY = 'KOREAN';

SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;

TO_DATE(
--------
89/01/15
```

The following example returns the default value because the specified expression cannot be converted to a DATE value, due to a misspelling of the month:

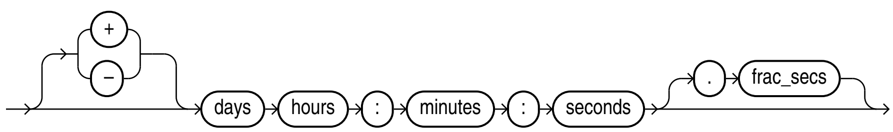
```
SELECT TO_DATE('Febuary 15, 2016, 11:00 A.M.'
       DEFAULT 'January 01, 2016 12:00 A.M.' ON CONVERSION ERROR,
       'Month dd, YYYY, HH:MI A.M.') "Value"
  FROM DUAL;

Value
---------
01-JAN-16
```

TO_DSINTERVAL

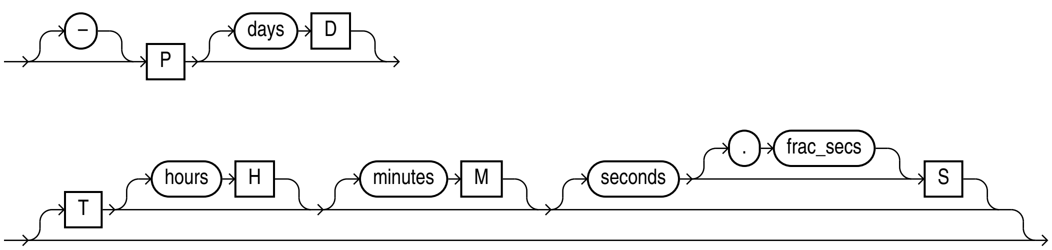
Syntax



***sql_format*::=**



***ds_iso_format*::=**



> **Note:**
>
> In earlier releases, the `TO_DSINTERVAL` function accepted an optional *`nlsparam`* clause. This clause is still accepted for backward compatibility, but has no effect.

Purpose

`TO_DSINTERVAL` converts its argument to a value of `INTERVAL DAY TO SECOND` data type.

For the argument, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

`TO_DSINTERVAL` accepts argument in one of the two formats:

- SQL interval format compatible with the SQL standard (ISO/IEC 9075)

- ISO duration format compatible with the ISO 8601:2004 standard

In the SQL format, *days* is an integer between 0 and 999999999, *hours* is an integer between 0 and 23, and *minutes* and *seconds* are integers between 0 and 59. *frac_secs* is the fractional part of seconds between .0 and .999999999. One or more blanks separate days from hours. Additional blanks are allowed between format elements.

In the ISO format, *days*, *hours*, *minutes* and *seconds* are integers between 0 and 999999999. *frac_secs* is the fractional part of seconds between .0 and .999999999. No blanks are allowed in the value. If you specify `T`, then you must specify at least one of the *hours*, *minutes*, or *seconds* values.

The optional `DEFAULT` *return_value* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting the argument to an `INTERVAL DAY TO SECOND` type. This clause has no effect if an error occurs while evaluating the argument. The *return_value* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type. It can be in either the SQL format or ISO format, and need not be in the same format as the function argument. If *return_value* cannot be converted to an `INTERVAL DAY TO SECOND` type, then the function returns an error.

Examples

The following example uses the SQL format to select from the `hr.employees` table the employees who had worked for the company for at least 100 days on November 1, 2002:

```
SELECT employee_id, last_name FROM employees
   WHERE hire_date + TO_DSINTERVAL('100 00:00:00')
   <= DATE '2002-11-01'
   ORDER BY employee_id;

EMPLOYEE_ID LAST_NAME
----------- ---------------
        102 De Haan
        203 Mavris
        204 Baer
        205 Higgins
        206 Giet
```

The following example uses the ISO format to display the timestamp 100 days and 5 hours after the beginning of the year 2009:

```
SELECT TO_CHAR(TIMESTAMP '2009-01-01 00:00:00' + TO_DSINTERVAL('P100DT05H'),
   'YYYY-MM-DD HH24:MI:SS') "Time Stamp"
     FROM DUAL;

Time Stamp
-------------------
2009-04-11 05:00:00
```

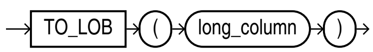
The following example returns the default value because the specified expression cannot be converted to an `INTERVAL DAY TO SECOND` value:

```
SELECT TO_DSINTERVAL('1o 1:02:10'
       DEFAULT '10 8:00:00' ON CONVERSION ERROR) "Value"
  FROM DUAL;
```

```
Value
-----------------------------
+000000010 08:00:00.000000000
```

TO_LOB

Syntax



Purpose

`TO_LOB` converts `LONG` or `LONG` RAW values in the column *`long_column`* to LOB values. You can apply this function only to a `LONG` or `LONG RAW` column, and only in the select list of a subquery in an `INSERT` statement.

Before using this function, you must create a LOB column to receive the converted `LONG` values. To convert `LONG` values, create a `CLOB` column. To convert `LONG` RAW values, create a `BLOB` column.

You cannot use the `TO_LOB` function to convert a `LONG` column to a LOB column in the subquery of a `CREATE TABLE` ... `AS SELECT` statement if you are creating an index-organized table. Instead, create the index-organized table without the `LONG` column, and then use the `TO_LOB` function in an `INSERT` ... `AS SELECT` statement.

You cannot use this function within a PL/SQL package. Instead use the `TO_CLOB` (character) or `TO_BLOB` (raw) functions.

> **See Also:**
>
> - the *`modify_col_properties`* clause of ALTER TABLE for an alternative method of converting `LONG` columns to LOB
> - INSERT for information on the subquery of an `INSERT` statement
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

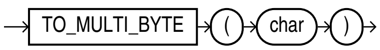
Examples

The following syntax shows how to use the `TO_LOB` function on your `LONG` data in a hypothetical table *`old_table`*:

```
CREATE TABLE new_table (col1, col2, ... lob_col CLOB);
INSERT INTO new_table (select o.col1, o.col2, ... TO_LOB(o.old_long_col)
   FROM old_table o;
```

TO_MULTI_BYTE

Syntax



Purpose

`TO_MULTI_BYTE` returns *`char`* with all of its single-byte characters converted to their corresponding multibyte characters. *`char`* can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is in the same data type as *`char`*.

Any single-byte characters in *`char`* that have no multibyte equivalents appear in the output string as single-byte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `TO_MULTI_BYTE`

Examples

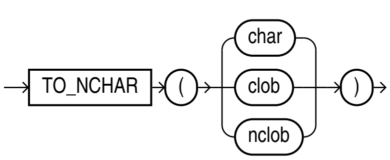
The following example illustrates converting from a single byte `A` to a multibyte `A` in UTF8:

```
SELECT dump(TO_MULTI_BYTE( 'A')) FROM DUAL;

DUMP(TO_MULTI_BYTE('A'))
------------------------
Typ=1 Len=3: 239,188,161
```

TO_NCHAR (character)

Syntax

to_nchar_char::=



Purpose

`TO_NCHAR` (character) converts a character string, `CHAR`, `VARCHAR2`, `CLOB`, or `NCLOB` value to the national character set. The value returned is always `NVARCHAR2`. This function is equivalent to the `TRANSLATE` ... `USING` function with a `USING` clause in the national character set.

> **See Also:**
>
> - "Data Conversion " and TRANSLATE ... USING
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

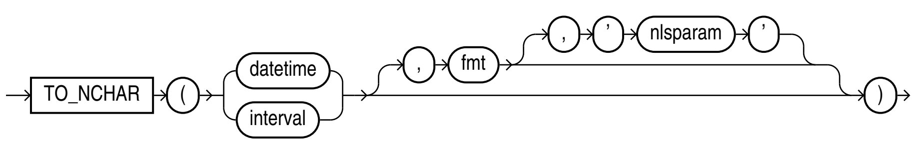
The following example converts `VARCHAR2` data from the `oe.customers` table to the national character set:

```
SELECT TO_NCHAR(cust_last_name) FROM customers
   WHERE customer_id=103;

TO_NCHAR(CUST_LAST_NAME)
--------------------------------------------------
Taylor
```

TO_NCHAR (datetime)

Syntax

***to_nchar_date*::=**



Purpose

`TO_NCHAR` (datetime) converts a datetime or interval value of `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL MONTH TO YEAR`, or `INTERVAL DAY TO SECOND` data type from the database character set to the national character set.

See Also:

- "Security Considerations for Data Conversion"
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

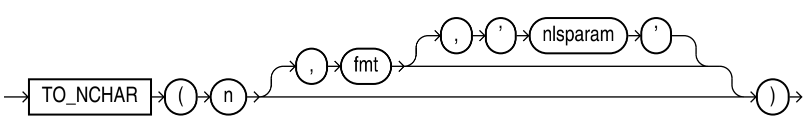
The following example converts the `order_date` of all orders whose status is `9` to the national character set:

```
SELECT TO_NCHAR(ORDER_DATE) AS order_date
   FROM ORDERS
   WHERE ORDER_STATUS > 9
   ORDER BY order_date;

ORDER_DATE
--------------------------------------------------------------------------
06-DEC-99 02.22.34.225609 PM
13-SEP-99 10.19.00.654279 AM
14-SEP-99 09.53.40.223345 AM
26-JUN-00 10.19.43.190089 PM
27-JUN-00 09.53.32.335522 PM
```

TO_NCHAR (number)

Syntax

***to_nchar_number*::=**



Purpose

`TO_NCHAR` (number) converts *`n`* to a string in the national character set. The value *`n`* can be of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`. The function returns a value of the same type as the argument. The optional *`fmt`* and *`'nlsparam'`* corresponding to *`n`* can be of `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL MONTH TO YEAR`, or `INTERVAL DAY TO SECOND` data type.

See Also:

- "Security Considerations for Data Conversion"
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

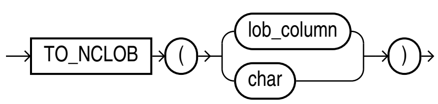
Examples

The following example converts the `customer_id` values from the sample table `oe.orders` to the national character set:

```
SELECT TO_NCHAR(customer_id) "NCHAR_Customer_ID"  FROM orders
   WHERE order_status > 9
   ORDER BY "NCHAR_Customer_ID";

NCHAR_Customer_ID
----------------------------------------
102
103
148
148
149
```

TO_NCLOB

Syntax



Purpose

`TO_NCLOB` converts `CLOB` values in a LOB column or other character strings to `NCLOB` values. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. Oracle Database implements this function by converting the character set of *`char`* from the database character set to the national character set.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

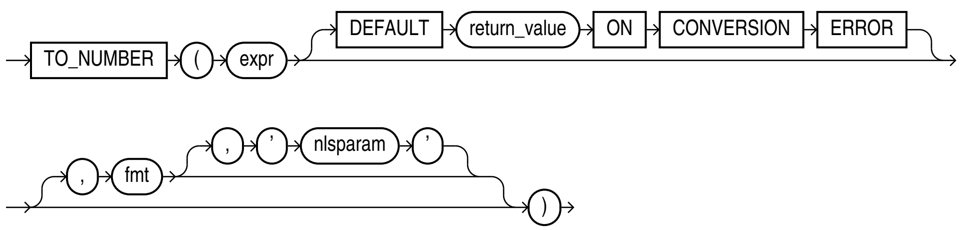
Examples

The following example inserts some character data into an `NCLOB` column of the `pm.print_media` table by first converting the data with the `TO_NCLOB` function:

```
INSERT INTO print_media (product_id, ad_id, ad_fltextn)
   VALUES (3502, 31001,
      TO_NCLOB('Placeholder for new product description'));
```

TO_NUMBER

Syntax



Purpose

`TO_NUMBER` converts *`expr`* to a value of `NUMBER` data type.

`expr` can be any expression that evaluates to a character string of type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`, a numeric value of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`, or null. If *`expr`* is `NUMBER`, then the function returns *`expr`*. If *`expr`* evaluates to null, then the function returns null. Otherwise, the function converts *`expr`* to a `NUMBER` value.

- If you specify an *`expr`* of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type, then you can optionally specify the format model *`fmt`*.
- If you specify an *`expr`* of `BINARY_FLOAT` or `BINARY_DOUBLE` data type, then you cannot specify a format model because a float can be interpreted only by its internal representation.

Refer to "Format Models " for information on number formats.

The *`'nlsparam'`* argument in this function has the same purpose as it does in the `TO_CHAR` function for number conversions. Refer to TO_CHAR (number) for more information.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information.

Examples

The following examples convert character string data into a number:

```
UPDATE employees SET salary = salary +
   TO_NUMBER('100.00', '9G999D99')
   WHERE last_name = 'Perkins';
```

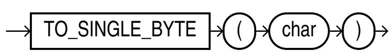
```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY            = ''AusDollars''
   ') "Amount"
     FROM DUAL;

    Amount
----------
      -100
```

The following example returns the default value of `0` because the specified expression cannot be converted to a `NUMBER` value:

```
SELECT TO_NUMBER('2,00' DEFAULT 0 ON CONVERSION ERROR) "Value"
  FROM DUAL;

   Value
--------
       0
```

TO_SINGLE_BYTE

Syntax



Purpose

`TO_SINGLE_BYTE` returns *`char`* with all of its multibyte characters converted to their corresponding single-byte characters. *`char`* can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is in the same data type as *`char`*.

Any multibyte characters in *`char`* that have no single-byte equivalents appear in the output as multibyte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `TO_SINGLE_BYTE`

Examples

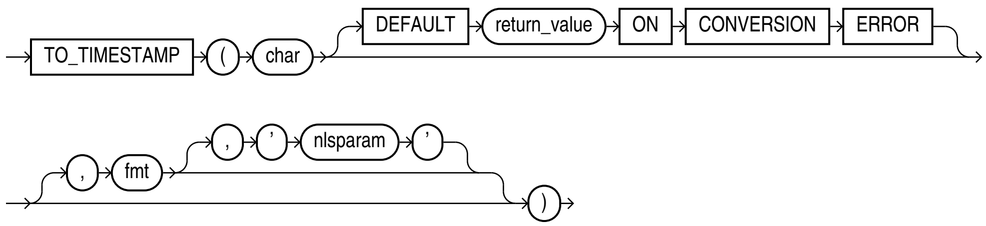
The following example illustrates going from a multibyte `A` in UTF8 to a single byte ASCII `A`:

```
SELECT TO_SINGLE_BYTE( CHR(15711393)) FROM DUAL;
```

T
-
A

TO_TIMESTAMP

Syntax



Purpose

TO_TIMESTAMP converts *char* to a value of TIMESTAMP data type.

For *char*, you can specify any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type.

The optional DEFAULT *return_value* ON CONVERSION ERROR clause allows you to specify the value this function returns if an error occurs while converting *char* to TIMESTAMP. This clause has no effect if an error occurs while evaluating *char*. The *return_value* can be an expression or a bind variable, and it must evaluate to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or null. The function converts *return_value* to TIMESTAMP using the same method it uses to convert *char* to TIMESTAMP. If *return_value* cannot be converted to TIMESTAMP, then the function returns an error.

The optional *fmt* specifies the format of *char*. If you omit *fmt*, then *char* must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. The optional *'nlsparam'* argument has the same purpose in this function as in the TO_CHAR function for date conversion.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information.

Examples

The following example converts a character string to a timestamp. The character string is not in the default TIMESTAMP format, so the format mask must be specified:

```
SELECT TO_TIMESTAMP ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF')
   FROM DUAL;
```

```
TO_TIMESTAMP('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF')
---------------------------------------------------------------------------
10-SEP-02 02.10.10.123000000 PM
```

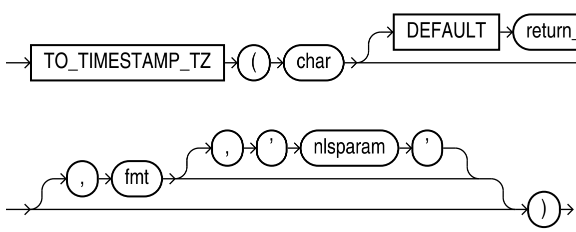
The following example returns the default value of NULL because the specified expression cannot be converted to a `TIMESTAMP` value, due to an invalid month specification:

```
SELECT TO_TIMESTAMP ('10-Sept-02 14:10:10.123000'
       DEFAULT NULL ON CONVERSION ERROR,
       'DD-Mon-RR HH24:MI:SS.FF',
       'NLS_DATE_LANGUAGE = American') "Value"
  FROM DUAL;
```

> **See Also:**
>
> `NLS_TIMESTAMP_FORMAT` initialization parameter for information on the default `TIMESTAMP` format and "Datetime Format Models " for information on specifying the format mask

TO_TIMESTAMP_TZ

Syntax



Purpose

`TO_TIMESTAMP_TZ` converts *`char`* to a value of `TIMESTAMP WITH TIME ZONE` data type.

For *`char`*, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

> **Note:**
>
> This function does not convert character strings to `TIMESTAMP WITH LOCAL TIME ZONE`. To do this, use a `CAST` function, as shown in CAST .

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting *`char`* to `TIMESTAMP WITH TIME ZONE`. This clause has no effect if an error occurs while evaluating *`char`*. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type, or null. The function converts *`return_value`* to

TIMESTAMP WITH TIME ZONE using the same method it uses to convert *char* to TIMESTAMP WITH TIME ZONE. If *return_value* cannot be converted to TIMESTAMP WITH TIME ZONE, then the function returns an error.

The optional *fmt* specifies the format of *char*. If you omit *fmt*, then *char* must be in the default format of the TIMESTAMP WITH TIME ZONE data type. The optional *'nlsparam'* has the same purpose in this function as in the TO_CHAR function for date conversion.

Examples

The following example converts a character string to a value of TIMESTAMP WITH TIME ZONE:

```
SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
   'YYYY-MM-DD HH:MI:SS TZH:TZM') FROM DUAL;

TO_TIMESTAMP_TZ('1999-12-0111:00:00-08:00','YYYY-MM-DDHH:MI:SSTZH:TZM')
--------------------------------------------------------------------
01-DEC-99 11.00.00.000000000 AM -08:00
```

The following example casts a null column in a UNION operation as TIMESTAMP WITH LOCAL TIME ZONE using the sample tables oe.order_items and oe.orders:

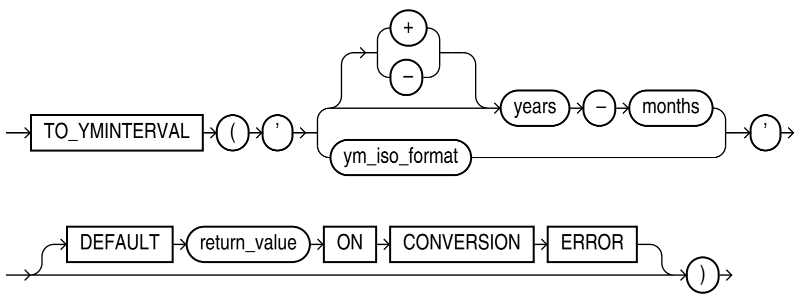
```
SELECT order_id, line_item_id,
   CAST(NULL AS TIMESTAMP WITH LOCAL TIME ZONE) order_date
   FROM order_items
UNION
SELECT order_id, to_number(null), order_date
   FROM orders;

  ORDER_ID LINE_ITEM_ID ORDER_DATE
---------- ------------ -----------------------------------
      2354            1
      2354            2
      2354            3
      2354            4
      2354            5
      2354            6
      2354            7
      2354            8
      2354            9
      2354           10
      2354           11
      2354           12
      2354           13
      2354              14-JUL-00 05.18.23.234567 PM
      2355            1
      2355            2
. . .
```

The following example returns the default value of NULL because the specified expression cannot be converted to a TIMESTAMP WITH TIME ZONE value, due to an invalid month specification:

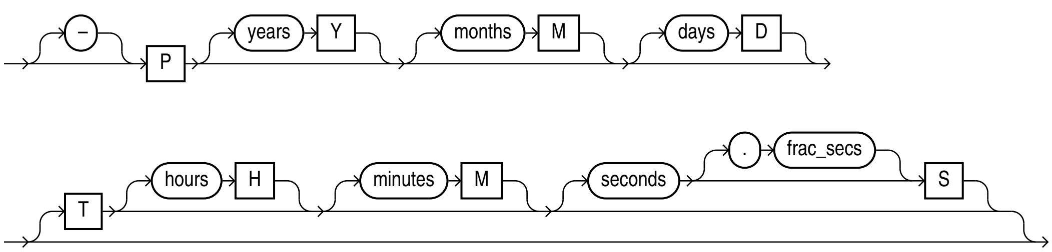
```
SELECT TO_TIMESTAMP_TZ('1999-13-01 11:00:00 -8:00'
       DEFAULT NULL ON CONVERSION ERROR,
       'YYYY-MM-DD HH:MI:SS TZH:TZM') "Value"
  FROM DUAL;
```

TO_YMINTERVAL

Syntax



***ym_iso_format*::=**



Purpose

`TO_YMINTERVAL` converts its argument to a value of `INTERVAL MONTH TO YEAR` data type.

For the argument, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

`TO_YMINTERVAL` accepts argument in one of the two formats:

- SQL interval format compatible with the SQL standard (ISO/IEC 9075)
- ISO duration format compatible with the ISO 8601:2004 standard

In the SQL format, *`years`* is an integer between 0 and 999999999, and *`months`* is an integer between 0 and 11. Additional blanks are allowed between format elements.

In the ISO format, years and months are integers between 0 and 999999999. Days, *`hours`*, *`minutes`*, *`seconds`*, and frac_secs are non-negative integers, and are ignored, if specified. No blanks are allowed in the value. If you specify `T`, then you must specify at least one of the *`hours`*, *`minutes`*, or *`seconds`* values.

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting the argument to an `INTERVAL MONTH TO YEAR` type. This clause has no effect if an error occurs while evaluating the argument. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type. It can be in either the SQL format or ISO format, and need not be in the same format as the function argument. If *return_value* cannot be converted to an `INTERVAL MONTH TO YEAR` type, then the function returns an error.

Examples

The following example calculates for each employee in the sample `hr.employees` table a date one year two months after the hire date:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('01-02') "14 months"
   FROM employees;

HIRE_DATE 14 months
--------- ---------
17-JUN-03 17-AUG-04
21-SEP-05 21-NOV-06
13-JAN-01 13-MAR-02
20-MAY-08 20-JUL-09
21-MAY-07 21-JUL-08

. . .
```

The following example makes the same calculation using the ISO format:

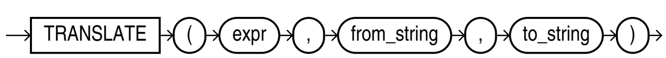
```
SELECT hire_date, hire_date + TO_YMINTERVAL('P1Y2M') FROM employees;
```

The following example returns the default value because the specified expression cannot be converted to an `INTERVAL MONTH TO YEAR` value:

```
SELECT TO_YMINTERVAL('1x-02'
       DEFAULT '00-00' ON CONVERSION ERROR) "Value"
  FROM DUAL;

Value
-------------
+000000000-00
```

TRANSLATE

Syntax



Purpose

`TRANSLATE` returns *expr* with all occurrences of each character in *from_string* replaced by its corresponding character in *to_string*. Characters in *expr* that are not in *from_string* are not replaced. The argument *from_string* can contain more characters than *to_string*. In this case, the extra characters at the end of *from_string* have no corresponding characters in *to_string*. If these extra characters appear in *expr*, then they are removed from the return value.

If a character appears multiple times in *from_string*, then the *to_string* mapping corresponding to the first occurrence is used.

You cannot use an empty string for *to_string* to remove all characters in *from_string* from the return value. Oracle Database interprets the empty string as null, and if this function has a null argument, then it returns null. To remove all characters in *from_string*, concatenate another character to the beginning of *from_string* and specify this character as the *to_string*. For example, TRANSLATE(*expr*, 'x0123456789', 'x') removes all digits from *expr*.

TRANSLATE provides functionality related to that provided by the REPLACE function. REPLACE lets you substitute a single string for another single string, as well as remove character strings. TRANSLATE lets you make several single-character, one-to-one substitutions in one operation.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information and REPLACE
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation TRANSLATE uses to compare characters from *expr* with characters from *from_string*, and for the collation derivation rules, which define the collation assigned to the character return value of TRANSLATE

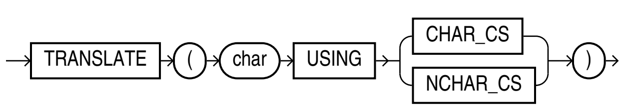
Examples

The following statement translates a book title into a string that could be used (for example) as a filename. The *from_string* contains four characters: a space, asterisk, slash, and apostrophe (with an extra apostrophe as the escape character). The *to_string* contains only three underscores. This leaves the fourth character in the *from_string* without a corresponding replacement, so apostrophes are dropped from the returned value.

```
SELECT TRANSLATE('SQL*Plus User''s Guide', ' */''', '___') FROM DUAL;

TRANSLATE('SQL*PLUSU
--------------------
SQL_Plus_Users_Guide
```

TRANSLATE ... USING

Syntax



Purpose

TRANSLATE ... USING converts *char* into the character set specified for conversions between the database character set and the national character set.

> **Note:**
>
> The TRANSLATE ... USING function is supported primarily for ANSI compatibility. Oracle recommends that you use the TO_CHAR and TO_NCHAR functions, as appropriate, for converting data to the database or national character set. TO_CHAR and TO_NCHAR can take as arguments a greater variety of data types than TRANSLATE ... USING, which accepts only character data.

The *char* argument is the expression to be converted.

- Specifying the USING CHAR_CS argument converts *char* into the database character set. The output data type is VARCHAR2.
- Specifying the USING NCHAR_CS argument converts *char* into the national character set. The output data type is NVARCHAR2.

This function is similar to the Oracle CONVERT function, but must be used instead of CONVERT if either the input or the output data type is being used as NCHAR or NVARCHAR2. If the input contains UCS2 code points or backslash characters (\), then use the UNISTR function.

> **See Also:**
>
> - CONVERT and UNISTR
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of TRANSLATE ... USING

Examples

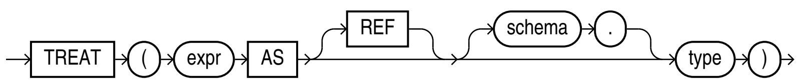
The following statements use data from the sample table oe.product_descriptions to show the use of the TRANSLATE ... USING function:

```
CREATE TABLE translate_tab (char_col  VARCHAR2(100),
                            nchar_col NVARCHAR2(50));
INSERT INTO translate_tab
   SELECT NULL, translated_name
      FROM product_descriptions
      WHERE product_id = 3501;

SELECT * FROM translate_tab;

CHAR_COL             NCHAR_COL
-------------------- --------------------------------------------------
. . .
                     C pre SPNIX4.0 - Sys
                     C pro SPNIX4.0 - Sys
                     C til SPNIX4.0 - Sys
                     C voor SPNIX4.0 - Sys
. . .
```

```
UPDATE translate_tab
   SET char_col = TRANSLATE (nchar_col USING CHAR_CS);

SELECT * FROM translate_tab;

CHAR_COL                  NCHAR_COL
------------------------- -------------------------
. . .
C per a SPNIX4.0 - Sys    C per a SPNIX4.0 - Sys
C pro SPNIX4.0 - Sys      C pro SPNIX4.0 - Sys
C for SPNIX4.0 - Sys      C for SPNIX4.0 - Sys
C til SPNIX4.0 - Sys      C til SPNIX4.0 - Sys
. . .
```

TREAT

Syntax



Purpose

`TREAT` changes the declared type of an expression.

You must have the `EXECUTE` object privilege on *`type`* to use this function.

- *`expr`* and *`type`* must be user-defined object types, excluding top-level collections.
- *`type`* must be some supertype or subtype of the declared type of *`expr`*. If the most specific type of *`expr`* is *`type`* (or some subtype of *`type`*), then `TREAT` returns *`expr`*. If the most specific type of *`expr`* is not *`type`* (or some subtype of *`type`*), then `TREAT` returns `NULL`.
- You can specify `REF` only if the declared type of *`expr`* is a `REF` type.
- If the declared type of *`expr`* is a `REF` to a source type of *`expr`*, then *`type`* must be some subtype or supertype of the source type of *`expr`*. If the most specific type of `DEREF`(*`expr`*) is *`type`* (or a subtype of *`type`*), then `TREAT` returns *`expr`*. If the most specific type of `DEREF`(*`expr`*) is not *`type`* (or a subtype of *`type`*), then `TREAT` returns `NULL`.

> **See Also:**
>
> "Data Type Comparison Rules " for more information

Examples

The following statement uses the table `oe.persons`, which is created in "Substitutable Table and Column Examples". The example retrieves the salary attribute of all people in the `persons` table, the value being null for instances of people that are not employees.

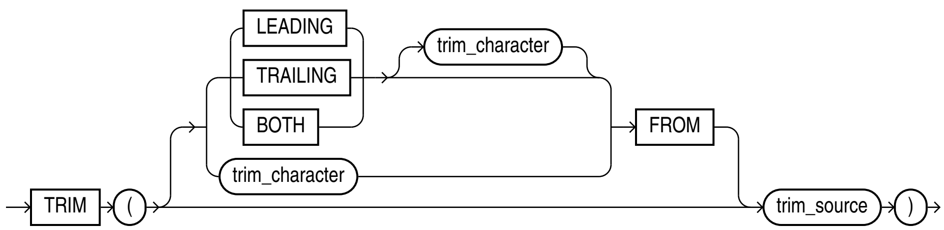
```
SELECT name, TREAT(VALUE(p) AS employee_t).salary salary
   FROM persons p;

NAME                          SALARY
```

```
------------------------- ----------
Bob
Joe                           100000
Tim                             1000
```

You can use the TREAT function to create an index on the subtype attributes of a substitutable column. For an example, see "Indexing on Substitutable Columns: Examples".

TRIM

Syntax



Purpose

TRIM enables you to trim leading or trailing characters (or both) from a character string. If *trim_character* or *trim_source* is a character literal, then you must enclose it in single quotation marks.

- If you specify LEADING, then Oracle Database removes any leading characters equal to *trim_character*.
- If you specify TRAILING, then Oracle removes any trailing characters equal to *trim_character*.
- If you specify BOTH or none of the three, then Oracle removes leading and trailing characters equal to *trim_character*.
- If you do not specify *trim_character*, then the default value is a blank space.
- If you specify only *trim_source*, then Oracle removes leading and trailing blank spaces.
- The function returns a value with data type VARCHAR2. The maximum length of the value is the length of *trim_source*.
- If either *trim_source* or *trim_character* is null, then the TRIM function returns null.

Both *trim_character* and *trim_source* can be VARCHAR2 or any data type that can be implicitly converted to VARCHAR2. The string returned is a VARCHAR2 (NVARCHAR2) data type if *trim_source* is a CHAR or VARCHAR2 (NCHAR or NVARCHAR2) data type, and a CLOB if *trim_source* is a CLOB data type. The return string is in the same character set as *trim_source*.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `TRIM` uses to compare characters from *`trim_character`* with characters from *`trim_source`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

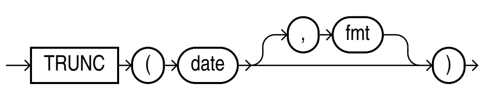
This example trims leading zeros from the hire date of the employees in the `hr` schema:

```
SELECT employee_id,
      TO_CHAR(TRIM(LEADING 0 FROM hire_date))
      FROM employees
      WHERE department_id = 60
      ORDER BY employee_id;

EMPLOYEE_ID TO_CHAR(T
----------- ---------
        103 20-MAY-08
        104 21-MAY-07
        105 25-JUN-05
        106 5-FEB-06
        107 7-FEB-07
```

TRUNC (date)

Syntax

***trunc_date*::=**



Purpose

The `TRUNC` (date) function returns *`date`* with the time portion of the day truncated to the unit specified by the format model *`fmt`*. This function is not sensitive to the `NLS_CALENDAR` session parameter. It operates according to the rules of the Gregorian calendar. The value returned is always of data type `DATE`, even if you specify a different datetime data type for *`date`*. If you omit *`fmt`*, then the default format model '`DD`' is used and the value returned is *`date`* truncated to the day with a time of midnight. Refer to "ROUND and TRUNC Date Functions" for the permitted format models to use in *`fmt`*.

Examples

The following example truncates a date:

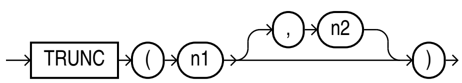
```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
  "New Year" FROM DUAL;

New Year
```

```
---------
01-JAN-92
```

TRUNC (number)

Syntax

***trunc_number*::=**



Purpose

The `TRUNC` (number) function returns *`n1`* truncated to *`n2`* decimal places. If *`n2`* is omitted, then *`n1`* is truncated to 0 places. *`n2`* can be negative to truncate (make zero) *`n2`* digits left of the decimal point.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If you omit *`n2`*, then the function returns the same data type as the numeric data type of the argument. If you include *`n2`*, then the function returns `NUMBER`.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

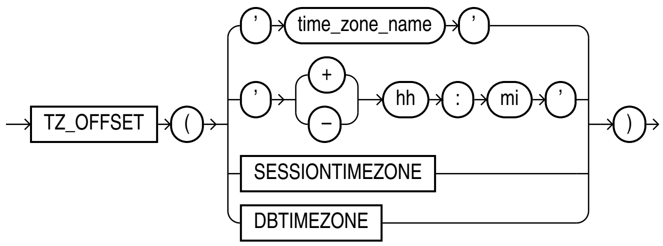
Examples

The following examples truncate numbers:

```
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

  Truncate
----------
      15.7

SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

  Truncate
----------
        10
```

TZ_OFFSET

Syntax



Purpose

`TZ_OFFSET` returns the time zone offset corresponding to the argument based on the date the statement is executed. You can enter a valid time zone region name, a time zone offset from UTC (which simply returns itself), or the keyword `SESSIONTIMEZONE` or `DBTIMEZONE`. For a listing of valid values for *`time_zone_name`*, query the `TZNAME` column of the `V$TIMEZONE_NAMES` dynamic performance view.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

> **See Also:**
>
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `TZ_OFFSET`

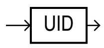
Examples

The following example returns the time zone offset of the US/Eastern time zone from UTC:

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;

TZ_OFFS
-------
-04:00
```

UID

Syntax



Purpose

`UID` returns an integer that uniquely identifies the session user (the user who logged on).

> **See Also:**
>
> USER to learn how Oracle Database determines the session user

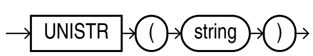
Examples

The following example returns the UID of the session user:

```
SELECT UID FROM DUAL;
```

UNISTR

Syntax



Purpose

`UNISTR` takes as its argument a text literal or an expression that resolves to character data and returns it in the national character set. The national character set of the database can be either AL16UTF16 or UTF8. `UNISTR` provides support for Unicode string literals by letting you specify the Unicode encoding value of characters in the string. This is useful, for example, for inserting data into `NCHAR` columns.

The Unicode encoding value has the form '\xxxx' where 'xxxx' is the hexadecimal value of a character in UCS-2 encoding format. Supplementary characters are encoded as two code units, the first from the high-surrogates range (U+D800 to U+DBFF), and the second from the low-surrogates range (U+DC00 to U+DFFF). To include the backslash in the string itself, precede it with another backslash (\\).

For portability and data preservation, Oracle recommends that in the `UNISTR` string argument you specify only ASCII characters and the Unicode encoding values.

See Also:

- *Oracle Database Globalization Support Guide* for information on Unicode and national character sets
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `UNISTR`

Examples

The following example passes both ASCII characters and Unicode encoding values to the `UNISTR` function, which returns the string in the national character set:

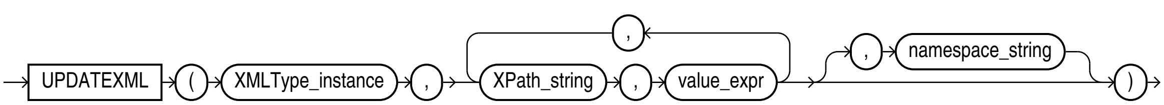
```
SELECT UNISTR('abc\00e5\00f1\00f6') FROM DUAL;

UNISTR
------
abcåñö
```

UPDATEXML

Note:

The `UPDATEXML` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use XQuery Update instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`UPDATEXML` takes as arguments an `XMLType` instance and an XPath-value pair and returns an `XMLType` instance with the updated value. If *`XPath_string`* is an XML element, then the corresponding *`value_expr`* must be an `XMLType` instance. If *`XPath_string`* is an attribute or text node, then the *`value_expr`* can be any scalar data type. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node.

The data types of the target of each *`XPath_string`* and its corresponding *`value_expr`* must match. The optional *`namespace_string`* must resolve to a `VARCHAR2` value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

If you update an XML element to null, then Oracle removes the attributes and children of the element, and the element becomes empty. If you update the text node of an element to null, Oracle removes the text value of the element, and the element itself remains but is empty.

In most cases, this function materializes an XML document in memory and updates the value. However, `UPDATEXML` is optimized for `UPDATE` statements on object-relational columns so that the function updates the value directly in the column. This optimization requires the following conditions:

- The *`XMLType_instance`* must be the same as the column in the `UPDATE` ... `SET` clause.
- The *`XPath_string`* must resolve to scalar content.

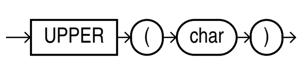
Examples

The following example updates to 4 the number of docks in the San Francisco warehouse in the sample schema `OE`, which has a `warehouse_spec` column of type `XMLType`:

```
SELECT warehouse_name,
   EXTRACT(warehouse_spec, '/Warehouse/Docks')
   "Number of Docks"
   FROM warehouses
   WHERE warehouse_name = 'San Francisco';

WAREHOUSE_NAME       Number of Docks
-------------------- --------------------
San Francisco        <Docks>1</Docks>

UPDATE warehouses SET warehouse_spec =
   UPDATEXML(warehouse_spec,
   '/Warehouse/Docks/text()',4)
   WHERE warehouse_name = 'San Francisco';

1 row updated.

SELECT warehouse_name,
   EXTRACT(warehouse_spec, '/Warehouse/Docks')
   "Number of Docks"
   FROM warehouses
   WHERE warehouse_name = 'San Francisco';

WAREHOUSE_NAME       Number of Docks
-------------------- --------------------
San Francisco        <Docks>4</Docks>
```

UPPER

Syntax



Purpose

UPPER returns *char*, with all letters uppercase. *char* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same data type as *char*. The database sets the case of the characters based on the binary mapping defined for the underlying character set. For linguistic-sensitive uppercase, refer to NLS_UPPER .

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of UPPER

Examples

The following example returns each employee's last name in uppercase:

```
SELECT UPPER(last_name) "Uppercase"
   FROM employees;
```

USER

Syntax

Purpose

USER returns the name of the session user (the user who logged on). This may change during the duration of a database session as Real Application Security sessions are attached or detached. For enterprise users, this function returns the schema. For other users, it returns the database user name. If a Real Application Security session is currently attached to the database session, then it returns user XS$NULL.

This function returns a VARCHAR2 value.

Oracle Database compares values of this function with blank-padded comparison semantics.

In a distributed SQL statement, the UID and USER functions together identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint.

> **See Also:**
>
> - *Oracle Database 2 Day + Security Guide* for more information on user XS$NULL
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of USER

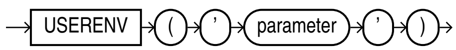
Examples

The following example returns the session user and the user's UID:

```
SELECT USER, UID FROM DUAL;
```

USERENV

Syntax



Purpose

> **Note:**
>
> `USERENV` is a legacy function that is retained for backward compatibility. Oracle recommends that you use the `SYS_CONTEXT` function with the built-in `USERENV` namespace for current functionality. See SYS_CONTEXT for more information.

`USERENV` returns information about the current session. This information can be useful for writing an application-specific audit trail table or for determining the language-specific characters currently used by your session. You cannot use `USERENV` in the condition of a `CHECK` constraint. Table 7-12 describes the values for the *`parameter`* argument.

All calls to `USERENV` return `VARCHAR2` data except for calls with the `SESSIONID`, `SID`, and `ENTRYID` parameters, which return `NUMBER`.

Table 7-12 Parameters of the USERENV Function

Parameter	Return Value
`CLIENT_INFO`	`CLIENT_INFO` returns up to 64 bytes of user session information that can be stored by an application using the `DBMS_APPLICATION_INFO` package. **Caution:** Some commercial applications may be using this context value. Refer to the applicable documentation for those applications to determine what restrictions they may impose on use of this context area. **See Also:** *Oracle Database Security Guide* for more information on application context, CREATE CONTEXT, and SYS_CONTEXT
`ENTRYID`	The current audit entry number. The audit entryid sequence is shared between fine-grained audit records and regular audit records. You cannot use this attribute in distributed SQL statements.
`ISDBA`	`ISDBA` returns '`TRUE`' if the user has been authenticated as having DBA privileges either through the operating system or through a password file.

Table 7-12 (Cont.) Parameters of the USERENV Function

Parameter	Return Value
LANG	LANG returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	LANGUAGE returns the language and territory used by the current session along with the database character set in this form: `language_territory.characterset`
SESSIONID	SESSIONID returns the auditing session identifier. You cannot specify this parameter in distributed SQL statements.
SID	SID returns the session ID.
TERMINAL	TERMINAL returns the operating system identifier for the terminal of the current session. In distributed SQL statements, this parameter returns the identifier for your local session. In a distributed environment, this parameter is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of USERENV

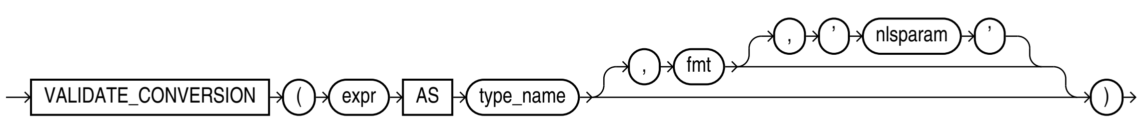
Examples

The following example returns the LANGUAGE parameter of the current session:

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

Language
-----------------------------------
AMERICAN_AMERICA.WE8ISO8859P1
```

VALIDATE_CONVERSION

Syntax



Purpose

VALIDATE_CONVERSION determines whether *expr* can be converted to the specified data type. If *expr* can be successfully converted, then this function returns 1; otherwise, this function returns 0. If *expr* evaluates to null, then this function returns 1. If an error occurs while evaluating *expr*, then this function returns the error.

For *expr*, specify a SQL expression. The acceptable data types for *expr*, and the purpose of the optional *fmt* and *nlsparam* arguments, depend on the data type you specify for *type_name*.

For *type_name*, specify the data type to which you want to convert *expr*. You can specify the following data types:

- BINARY_DOUBLE

  If you specify BINARY_DOUBLE, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_BINARY_DOUBLE function. Refer to TO_BINARY_DOUBLE for more information.

- BINARY_FLOAT

  If you specify BINARY_FLOAT, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_BINARY_FLOAT function. Refer to TO_BINARY_FLOAT for more information.

- DATE

  If you specify DATE, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_DATE function. Refer to TO_DATE for more information.

- INTERVAL DAY TO SECOND

  If you specify INTERVAL DAY TO SECOND, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, and must contain a value in either the SQL interval format or the ISO duration format. The optional *fmt* and *nlsparam* arguments do not apply for this data type. Refer to TO_DSINTERVAL for more information on the SQL interval format and the ISO duration format.

- INTERVAL YEAR TO MONTH

  If you specify INTERVAL YEAR TO MONTH, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, and must contain a value in either the SQL interval format or the ISO duration format. The optional *fmt* and *nlsparam* arguments do not apply for this data type. Refer to TO_YMINTERVAL for more information on the SQL interval format and the ISO duration format.

- NUMBER

  If you specify NUMBER, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_NUMBER function. Refer to TO_NUMBER for more information.

If *expr* is a value of type NUMBER, then the VALIDATE_CONVERSION function verifies that *expr* is a legal numeric value. If *expr* is not a legal numeric value, then the function returns 0. This enables you to identify corrupt numeric values in your database.

- TIMESTAMP

  If you specify TIMESTAMP, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. Refer to TO_TIMESTAMP for more information.

- TIMESTAMP WITH TIME ZONE

  If you specify TIMESTAMP WITH TIME ZONE, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP_TZ function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP WITH TIME ZONE data type, which is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. Refer to TO_TIMESTAMP_TZ for more information.

- TIMESTAMP WITH LOCAL TIME ZONE

  If you specify TIMESTAMP, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. Refer to TO_TIMESTAMP for more information.

Examples

In each of the following statements, the specified value can be successfully converted to the specified data type. Therefore, each of these statements returns a value of 1.

```
SELECT VALIDATE_CONVERSION(1000 AS BINARY_DOUBLE)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('1234.56' AS BINARY_FLOAT)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('July 20, 1969, 20:18' AS DATE,
    'Month dd, YYYY, HH24:MI', 'NLS_DATE_LANGUAGE = American')
  FROM DUAL;

SELECT VALIDATE_CONVERSION('200 00:00:00' AS INTERVAL DAY TO SECOND)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('P1Y2M' AS INTERVAL YEAR TO MONTH)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('$100,00' AS NUMBER,
    '$999D99', 'NLS_NUMERIC_CHARACTERS = '',.''')
  FROM DUAL;

SELECT VALIDATE_CONVERSION('29-Jan-02 17:24:00' AS TIMESTAMP,
    'DD-MON-YY HH24:MI:SS')
  FROM DUAL;
```

```
SELECT VALIDATE_CONVERSION('1999-12-01 11:00:00 -8:00'
    AS TIMESTAMP WITH TIME ZONE, 'YYYY-MM-DD HH:MI:SS TZH:TZM')
  FROM DUAL;

SELECT VALIDATE_CONVERSION('11-May-16 17:30:00'
    AS TIMESTAMP WITH LOCAL TIME ZONE, 'DD-MON-YY HH24:MI:SS')
  FROM DUAL;
```

The following statement returns 0, because the specified value cannot be converted to `BINARY_FLOAT`:

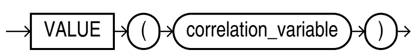
```
SELECT VALIDATE_CONVERSION('$29.99' AS BINARY_FLOAT)
  FROM DUAL;
```

The following statement returns 1, because the specified number format model enables the value to be converted to `BINARY_FLOAT`:

```
SELECT VALIDATE_CONVERSION('$29.99' AS BINARY_FLOAT, '$99D99')
  FROM DUAL;
```

VALUE

Syntax



Purpose

`VALUE` takes as its argument a correlation variable (table alias) associated with a row of an object table and returns object instances stored in the object table. The type of the object instances is the same type as the object table.

Examples

The following example uses the sample table `oe.persons`, which is created in "Substitutable Table and Column Examples":

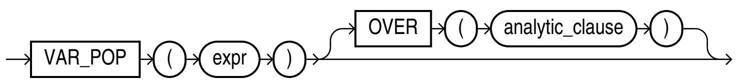
```
SELECT VALUE(p) FROM persons p;

VALUE(P)(NAME, SSN)
-------------------------------------------------------------
PERSON_T('Bob', 1234)
EMPLOYEE_T('Joe', 32456, 12, 100000)
PART_TIME_EMP_T('Tim', 5678, 13, 1000, 20)
```

See Also:

"IS OF *type* Condition " for information on using `IS OF` type conditions with the `VALUE` function

VAR_POP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`VAR_POP` returns the population variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

If the function is applied to an empty set, then it returns null. The function makes the following calculation:

`SUM((expr - (SUM(expr) / COUNT(expr)))`^{2}`) / COUNT(expr)`

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *`expr`* and "Aggregate Functions "

Aggregate Example

The following example returns the population variance of the salaries in the `employees` table:

```
SELECT VAR_POP(salary) FROM employees;

VAR_POP(SALARY)
---------------
     15141964.9
```

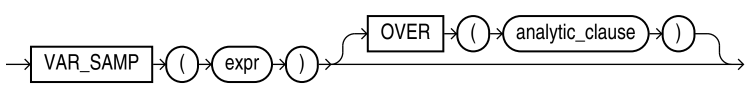
Analytic Example

The following example calculates the cumulative population and sample variances in the `sh.sales` table of the monthly sales in 1998:

```
SELECT t.calendar_month_desc,
   VAR_POP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Pop",
   VAR_SAMP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Samp"
  FROM sales s, times t
  WHERE s.time_id = t.time_id AND t.calendar_year = 1998
  GROUP BY t.calendar_month_desc
  ORDER BY t.calendar_month_desc, "Var_Pop", "Var_Samp";

CALENDAR    Var_Pop   Var_Samp
-------- ---------- ----------
1998-01           0
1998-02  2269111326 4538222653
1998-03  5.5849E+10 8.3774E+10
1998-04  4.8252E+10 6.4336E+10
1998-05  6.0020E+10 7.5025E+10
1998-06  5.4091E+10 6.4909E+10
1998-07  4.7150E+10 5.5009E+10
1998-08  4.1345E+10 4.7252E+10
1998-09  3.9591E+10 4.4540E+10
1998-10  3.9995E+10 4.4439E+10
1998-11  3.6870E+10 4.0558E+10
1998-12  4.0216E+10 4.3872E+10
```

VAR_SAMP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`VAR_SAMP` returns the sample variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

If the function is applied to an empty set, then it returns null. The function makes the following calculation:

```
(SUM(expr - (SUM(expr) / COUNT(expr)))2) / (COUNT(expr) - 1)
```

This function is similar to `VARIANCE`, except that given an input set of one element, `VARIANCE` returns 0 and `VAR_SAMP` returns null.

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *`expr`* and "Aggregate Functions "

Aggregate Example

The following example returns the sample variance of the salaries in the sample `employees` table.

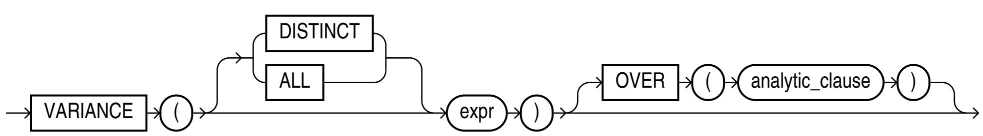
```
SELECT VAR_SAMP(salary) FROM employees;

VAR_SAMP(SALARY)
----------------
      15284813.7
```

Analytic Example

Refer to the analytic example for VAR_POP .

VARIANCE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`VARIANCE` returns the variance of *`expr`*. You can use it as an aggregate or analytic function.

Oracle Database calculates the variance of *`expr`* as follows:

- 0 if the number of rows in *`expr`* = 1
- `VAR_SAMP` if the number of rows in *`expr`* > 1

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion, "About SQL Expressions " for information on valid forms of *`expr`* and "Aggregate Functions "

Aggregate Example

The following example calculates the variance of all salaries in the sample `employees` table:

```
SELECT VARIANCE(salary) "Variance"
   FROM employees;

  Variance
----------
15283140.5
```

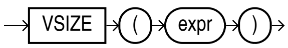
Analytic Example

The following example returns the cumulative variance of salary values in Department 30 ordered by hire date.

```
SELECT last_name, salary, VARIANCE(salary)
      OVER (ORDER BY hire_date) "Variance"
   FROM employees
   WHERE department_id = 30
   ORDER BY last_name, salary, "Variance";

LAST_NAME                     SALARY   Variance
------------------------- ---------- ----------
Baida                           2900 16283333.3
Colmenares                      2500   11307000
Himuro                          2600   13317000
Khoo                            3100   31205000
Raphaely                       11000          0
Tobias                          2800 21623333.3
```

VSIZE

Syntax



Purpose

`VSIZE` returns the number of bytes in the internal representation of *`expr`*. If *`expr`* is null, then this function returns null.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information

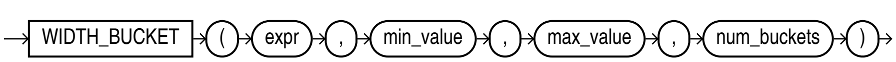
Examples

The following example returns the number of bytes in the `last_name` column of the employees in department 10:

```
SELECT last_name, VSIZE (last_name) "BYTES"
  FROM employees
  WHERE department_id = 10
  ORDER BY employee_id;

LAST_NAME            BYTES
--------------- ----------
Whalen                   6
```

WIDTH_BUCKET

Syntax



Purpose

`WIDTH_BUCKET` lets you construct equiwidth histograms, in which the histogram range is divided into intervals that have identical size. (Compare this function with `NTILE`, which creates equiheight histograms.) Ideally each bucket is a closed-open interval of the real number line. For example, a bucket can be assigned to scores between 10.00 and 19.999 ... to indicate that 10 is included in the interval and 20 is excluded. This is sometimes denoted [10, 20).

For a given expression, `WIDTH_BUCKET` returns the bucket number into which the value of this expression would fall after being evaluated.

- *`expr`* is the expression for which the histogram is being created. This expression must evaluate to a numeric or datetime value or to a value that can be implicitly converted to a numeric or datetime value. If *`expr`* evaluates to null, then the expression returns null.
- *`min_value`* and *`max_value`* are expressions that resolve to the end points of the acceptable range for *`expr`*. Both of these expressions must also evaluate to numeric or datetime values, and neither can evaluate to null.
- *`num_buckets`* is an expression that resolves to a constant indicating the number of buckets. This expression must evaluate to a positive integer.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion

When needed, Oracle Database creates an underflow bucket numbered 0 and an overflow bucket numbered *`num_buckets`*+1. These buckets handle values less than *`min_value`* and more than *`max_value`* and are helpful in checking the reasonableness of endpoints.

Examples

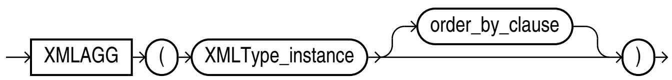
The following example creates a ten-bucket histogram on the `credit_limit` column for customers in Switzerland in the sample table `oe.customers` and returns the bucket number ("Credit Group") for each customer. Customers with credit limits greater than or equal to the maximum value are assigned to the overflow bucket, 11:

```
SELECT customer_id, cust_last_name, credit_limit,
   WIDTH_BUCKET(credit_limit, 100, 5000, 10) "Credit Group"
   FROM customers WHERE nls_territory = 'SWITZERLAND'
   ORDER BY "Credit Group", customer_id, cust_last_name, credit_limit;

CUSTOMER_ID CUST_LAST_NAME       CREDIT_LIMIT Credit Group
----------- -------------------- ------------ ------------
        825 Dreyfuss                      500            1
        826 Barkin                        500            1
        827 Siegel                        500            1
        853 Palin                         400            1
        843 Oates                         700            2
        844 Julius                        700            2
        835 Eastwood                     1200            3
        836 Berenger                     1200            3
        837 Stanton                      1200            3
        840 Elliott                      1400            3
        841 Boyer                        1400            3
        842 Stern                        1400            3
        848 Olmos                        1800            4
        849 Kaurusmdki                   1800            4
        828 Minnelli                     2300            5
        829 Hunter                       2300            5
        850 Finney                       2300            5
        851 Brown                        2300            5
        852 Tanner                       2300            5
```

```
830 Dutt                            3500            7
831 Bel Geddes                      3500            7
832 Spacek                          3500            7
833 Moranis                         3500            7
834 Idle                            3500            7
838 Nicholson                       3500            7
839 Johnson                         3500            7
845 Fawcett                         5000           11
846 Brando                          5000           11
847 Streep                          5000           11
```

XMLAGG

Syntax



Purpose

`XMLAgg` is an aggregate function. It takes a collection of XML fragments and returns an aggregated XML document. Any arguments that return null are dropped from the result.

`XMLAgg` is similar to `SYS_XMLAgg` except that `XMLAgg` returns a collection of nodes but it does not accept formatting using the `XMLFormat` object. Also, `XMLAgg` does not enclose the output in an element tag as does `SYS_XMLAgg`.

Within the *`order_by_clause`*, Oracle Database does not interpret number literals as column positions, as it does in other uses of this clause, but simply as number literals.

> **See Also:**
>
> XMLELEMENT and SYS_XMLAGG

Examples

The following example produces a `Department` element containing `Employee` elements with employee job ID and last name as the contents of the elements:

```
SELECT XMLELEMENT("Department",
   XMLAGG(XMLELEMENT("Employee",
   e.job_id||' '||e.last_name)
   ORDER BY last_name))
   as "Dept_list"
   FROM employees e
   WHERE e.department_id = 30;

Dept_list
-------------------------------------------------------------
<Department>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
```

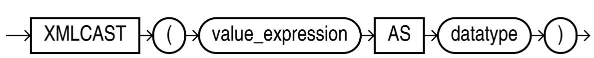
```
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Tobias</Employee>
</Department>
```

The result is a single row, because `XMLAgg` aggregates the rows. You can use the `GROUP BY` clause to group the returned set of rows into multiple groups:

```
SELECT XMLELEMENT("Department",
      XMLAGG(XMLELEMENT("Employee", e.job_id||' '||e.last_name)))
   AS "Dept_list"
   FROM employees e
   GROUP BY e.department_id;

Dept_list
---------------------------------------------------------
<Department>
  <Employee>AD_ASST Whalen</Employee>
</Department>

<Department>
  <Employee>MK_MAN Hartstein</Employee>
  <Employee>MK_REP Fay</Employee>
</Department>

<Department>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_CLERK Tobias</Employee>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
</Department>
. . .
```

XMLCAST

Syntax



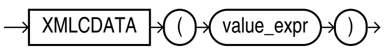
Purpose

`XMLCast` casts *`value_expression`* to the scalar SQL data type specified by *`datatype`*. The *`value_expression`* argument is a SQL expression that is evaluated. The *`datatype`* argument can be of data type `NUMBER`, `VARCHAR2`, `CHAR`, `CLOB`, `BLOB`, `REF XMLTYPE`, and any of the datetime data types.

> **See Also:**
>
> - *Oracle XML DB Developer's Guide* for more information on uses for this function and examples
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `XMLCAST` when it is a character value

XMLCDATA

Syntax



Purpose

`XMLCData` generates a CDATA section by evaluating *`value_expr`*. The *`value_expr`* must resolve to a string. The value returned by the function takes the following form:

```
<![CDATA[string]]>
```

If the resulting value is not a valid XML CDATA section, then the function returns an error.The following conditions apply to `XMLCData`:

- The *`value_expr`* cannot contain the substring `]]>`.
- If *`value_expr`* evaluates to null, then the function returns null.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

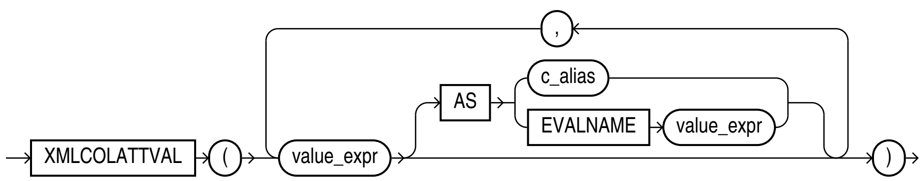
The following statement uses the `DUAL` table to illustrate the syntax of `XMLCData`:

```
SELECT XMLELEMENT("PurchaseOrder",
   XMLAttributes(dummy as "pono"),
   XMLCdata('<!DOCTYPE po_dom_group [
   <!ELEMENT po_dom_group(student_name)*>
   <!ELEMENT po_purch_name (#PCDATA)>
   <!ATTLIST po_name po_no ID #REQUIRED>
   <!ATTLIST po_name trust_1 IDREF #IMPLIED>
   <!ATTLIST po_name trust_2 IDREF #IMPLIED>
   ]>')) "XMLCData" FROM DUAL;

XMLCData
----------------------------------------------------------
<PurchaseOrder pono="X"><![CDATA[
<!DOCTYPE po_dom_group [
```

```
   <!ELEMENT po_dom_group(student_name)*>
   <!ELEMENT po_purch_name (#PCDATA)>
   <!ATTLIST po_name po_no ID #REQUIRED>
   <!ATTLIST po_name trust_1 IDREF #IMPLIED>
   <!ATTLIST po_name trust_2 IDREF #IMPLIED>
   ]>
  ]]>
</PurchaseOrder>
```

XMLCOLATTVAL

Syntax



Purpose

`XMLColAttVal` creates an XML fragment and then expands the resulting XML so that each XML fragment has the name `column` with the attribute `name`.

You can use the `AS` clause to change the value of the `name` attribute to something other than the column name. You can do this by specifying *`c_alias`*, which is a string literal, or by specifying `EVALNAME` *`value_expr`*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the alias. The alias can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`. See "Extended Data Types" for more information.

You must specify a value for *`value_expr`*. If *`value_expr`* is null, then no element is returned.

Restriction on XMLColAttVal

You cannot specify an object type column for *`value_expr`*.

Examples

The following example creates an `Emp` element for a subset of employees, with nested `employee_id`, `last_name`, and `salary` elements as the contents of `Emp`. Each nested element is named `column` and has a `name` attribute with the column name as the attribute value:

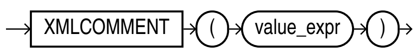
```
SELECT XMLELEMENT("Emp",
   XMLCOLATTVAL(e.employee_id, e.last_name, e.salary)) "Emp Element"
   FROM employees e
   WHERE employee_id = 204;

Emp Element
--------------------------------------------------------------------
<Emp>
  <column name="EMPLOYEE_ID">204</column>
```

```
  <column name="LAST_NAME">Baer</column>
  <column name="SALARY">10000</column>
</Emp>
```

Refer to the example for XMLFOREST to compare the output of these two functions.

XMLCOMMENT

Syntax



Purpose

XMLComment generates an XML comment using an evaluated result of *value_expr*. The *value_expr* must resolve to a string. It cannot contain two consecutive dashes (hyphens). The value returned by the function takes the following form:

```
<!--string-->
```

If *value_expr* resolves to null, then the function returns null.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

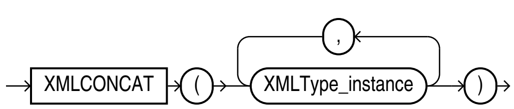
Examples

The following example uses the DUAL table to illustrate the XMLComment syntax:

```
SELECT XMLCOMMENT('OrderAnalysisComp imported, reconfigured, disassembled')
   AS "XMLCOMMENT" FROM DUAL;

XMLCOMMENT
--------------------------------------------------------------------------------
<!--OrderAnalysisComp imported, reconfigured, disassembled-->
```

XMLCONCAT

Syntax



Purpose

XMLConcat takes as input a series of XMLType instances, concatenates the series of elements for each row, and returns the concatenated series. XMLConcat is the inverse of XMLSequence.

Null expressions are dropped from the result. If all the value expressions are null, then the function returns null.

> **See Also:**
>
> XMLSEQUENCE

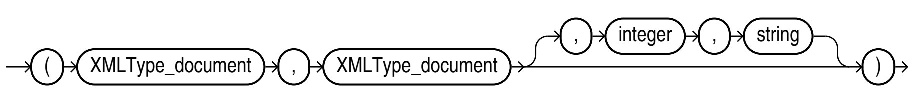
Examples

The following example creates XML elements for the first and last names of a subset of employees, and then concatenates and returns those elements:

```
SELECT XMLCONCAT(XMLELEMENT("First", e.first_name),
   XMLELEMENT("Last", e.last_name)) AS "Result"
   FROM employees e
   WHERE e.employee_id > 202;

Result
----------------------------------------------------------------
<First>Susan</First>
<Last>Mavris</Last>

<First>Hermann</First>
<Last>Baer</Last>

<First>Shelley</First>
<Last>Higgins</Last>

<First>William</First>
<Last>Gietz</Last>

4 rows selected.
```

XMLDIFF

Syntax



Purpose

The `XMLDiff` function is the SQL interface for the XmlDiff C API. This function compares two XML documents and captures the differences in XML conforming to an Xdiff schema. The diff document is returned as an XMLType document.

- For the first two arguments, specify the names of two XMLType documents.
- For the *`integer`*, specify a number representing the hashLevel for a C function XmlDiff. If you do not want hashing, set this argument to 0 or omit it entirely. If you

do not want hashing, but you want to specify flags, then you must set this argument to 0.

- For `string`, specify the flags that control the behavior of the function. These flags are specified by one or more names separated by semicolon. The names are the same as the names of constants for XmlDiff function.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide* for more information on using this function, including examples, and *Oracle Database XML C API Reference* for information on the XML APIs for C

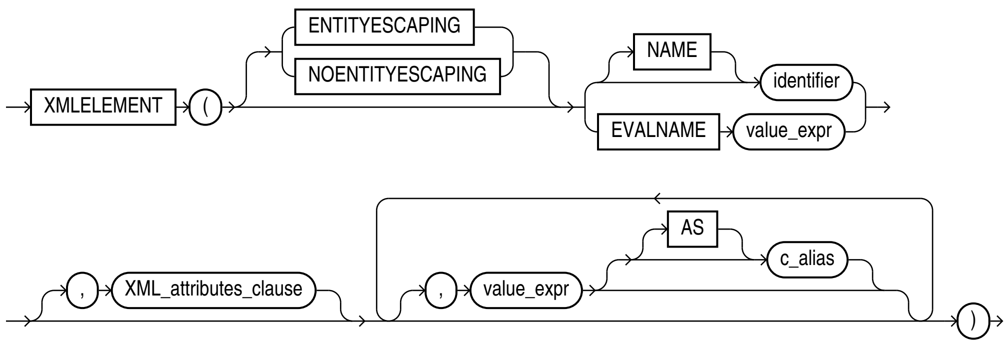
Examples

The following example compares two XML documents and returns the difference as an XMLType document:

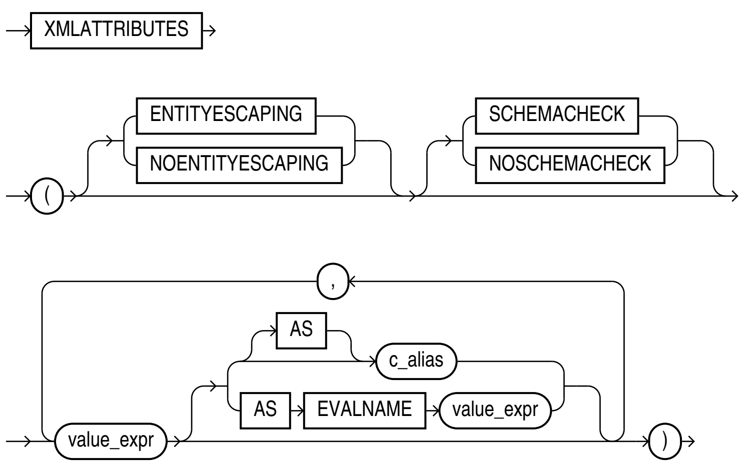
```
SELECT XMLDIFF(
XMLTYPE('<?xml version="1.0"?>
<bk:book xmlns:bk="http://example.com">
   <bk:tr>
        <bk:td>
                <bk:chapter>
                        Chapter 1.
                </bk:chapter>
        </bk:td>
        <bk:td>
                 <bk:chapter>
                        Chapter 2.
                </bk:chapter>
        </bk:td>
   </bk:tr>
</bk:book>'),
XMLTYPE('<?xml version="1.0"?>
<bk:book xmlns:bk="http://example.com">
   <bk:tr>
        <bk:td>
                <bk:chapter>
                        Chapter 1.
                </bk:chapter>
        </bk:td>
        <bk:td/>
   </bk:tr>
</bk:book>')
)
FROM DUAL;
```

XMLELEMENT

Syntax



***XML_attributes_clause*::=**



Purpose

`XMLElement` takes an element name for *`identifier`* or evaluates an element name for `EVALNAME` *`value_expr`*, an optional collection of attributes for the element, and arguments that make up the content of the element. It returns an instance of type `XMLType`. `XMLElement` is similar to `SYS_XMLGen` except that `XMLElement` can include attributes in the XML returned, but it does not accept formatting using the `XMLFormat` object.

The `XMLElement` function is typically nested to produce an XML document with a nested structure, as in the example in the following section.

For an explanation of the `ENTITYESCAPING` and `NONENTITYESCAPING` keywords, refer to *Oracle XML DB Developer's Guide*.

You must specify a value for Oracle Database to use an the enclosing tag. You can do this by specifying *`identifier`*, which is a string literal, or by specifying `EVALNAME` *value_expr*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the identifier. The identifier does not have to be a column name or column reference. It cannot be an expression or null. It can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`.

The objects that make up the element content follow the `XMLATTRIBUTES` keyword. In the *XML_attributes_clause*, if the *value_expr* is null, then no attribute is created for that value expression. The type of *value_expr* cannot be an object type or collection. If you specify an alias for *value_expr* using the `AS` clause, then the *c_alias* or the evaluated value expression (`EVALNAME` *value_expr*) can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`.

> **See Also:**
>
> "Extended Data Types" for more information on `MAX_STRING_SIZE`

For the optional *value_expr* that follows the *XML_attributes_clause* in the diagram:

- If *value_expr* is a scalar expression, then you can omit the `AS` clause, and Oracle uses the column name as the element name.
- If *value_expr* is an object type or collection, then the `AS` clause is mandatory, and Oracle uses the specified *c_alias* as the enclosing tag.
- If *value_expr* is null, then no element is created for that value expression.

> **See Also:**
>
> SYS_XMLGEN

Examples

The following example produces an `Emp` element for a series of employees, with nested elements that provide the employee's name and hire date:

```
SELECT XMLELEMENT("Emp", XMLELEMENT("Name",
   e.job_id||' '||e.last_name),
   XMLELEMENT("Hiredate", e.hire_date)) as "Result"
   FROM employees e WHERE employee_id > 200;

Result
-------------------------------------------------------------------
<Emp>
  <Name>MK_MAN Hartstein</Name>
  <Hiredate>2004-02-17</Hiredate>
</Emp>

<Emp>
  <Name>MK_REP Fay</Name>
  <Hiredate>2005-08-17</Hiredate>
</Emp>
```

```
<Emp>
  <Name>HR_REP Mavris</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

<Emp>
  <Name>PR_REP Baer</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

<Emp>
  <Name>AC_MGR Higgins</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

<Emp>
  <Name>AC_ACCOUNT Gietz</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

6 rows selected.
```

The following similar example uses the `XMLElement` function with the *`XML_attributes_clause`* to create nested XML elements with attribute values for the top-level element:

```
SELECT XMLELEMENT("Emp",
      XMLATTRIBUTES(e.employee_id AS "ID", e.last_name),
      XMLELEMENT("Dept", e.department_id),
      XMLELEMENT("Salary", e.salary)) AS "Emp Element"
   FROM employees e
   WHERE e.employee_id = 206;

Emp Element
---------------------------------------------------------------
<Emp ID="206" LAST_NAME="Gietz">
  <Dept>110</Dept>
  <Salary>8300</Salary>
</Emp>
```

Notice that the `AS` *`identifier`* clause was not specified for the `last_name` column. As a result, the XML returned uses the column name `last_name` as the default.

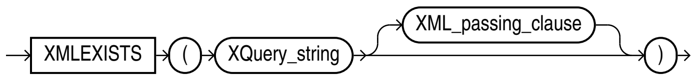
Finally, the next example uses a subquery within the *`XML_attributes_clause`* to retrieve information from another table into the attributes of an element:

```
SELECT XMLELEMENT("Emp", XMLATTRIBUTES(e.employee_id, e.last_name),
   XMLELEMENT("Dept", XMLATTRIBUTES(e.department_id,
   (SELECT d.department_name FROM departments d
   WHERE d.department_id = e.department_id) as "Dept_name")),
   XMLELEMENT("salary", e.salary),
   XMLELEMENT("Hiredate", e.hire_date)) AS "Emp Element"
   FROM employees e
   WHERE employee_id = 205;

Emp Element
-------------------------------------------------------------------
<Emp EMPLOYEE_ID="205" LAST_NAME="Higgins">
  <Dept DEPARTMENT_ID="110" Dept_name="Accounting"/>
  <salary>12008</salary>
```

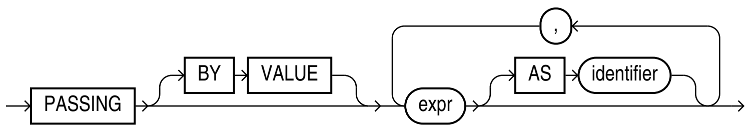
```
  <Hiredate>2002-06-07</Hiredate>
</Emp>
```

XMLEXISTS

Syntax



***XML_passing_clause*::=**



Purpose

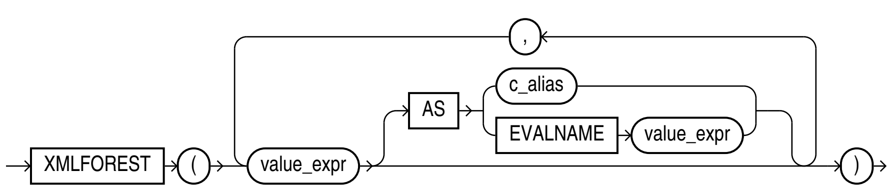
`XMLExists` checks whether a given XQuery expression returns a nonempty XQuery sequence. If so, the function returns `TRUE`; otherwise, it returns `FALSE`. The argument *`XQuery_string`* is a literal string, but it can contain XQuery variables that you bind using the *`XML_passing_clause`*.

The *`expr`* in the *`XML_passing_clause`* is an expression returning an `XMLType` or an instance of a SQL scalar data type that is used as the context for evaluating the XQuery expression. You can specify only one *`expr`* in the `PASSING` clause without an identifier. The result of evaluating each *`expr`* is bound to the corresponding identifier in the *`XQuery_string`*. If any *`expr`* that is not followed by an `AS` clause, then the result of evaluating that expression is used as the context item for evaluating the *`XQuery_string`*. If *`expr`* is a relational column, then its declared collation is ignored by Oracle XML DB.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on uses for this function and examples

XMLFOREST

Syntax



Purpose

XMLForest converts each of its argument parameters to XML, and then returns an XML fragment that is the concatenation of these converted arguments.

- If *value_expr* is a scalar expression, then you can omit the AS clause, and Oracle Database uses the column name as the element name.
- If *value_expr* is an object type or collection, then the AS clause is mandatory, and Oracle uses the specified expression as the enclosing tag.

  You can do this by specifying *c_alias*, which is a string literal, or by specifying EVALNAME *value_expr*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the identifier. The identifier does not have to be a column name or column reference. It cannot be an expression or null. It can be up to 4000 characters if the initialization parameter MAX_STRING_SIZE = STANDARD, and 32767 characters if MAX_STRING_SIZE = EXTENDED. See "Extended Data Types" for more information.
- If *value_expr* is null, then no element is created for that *value_expr*.

Examples

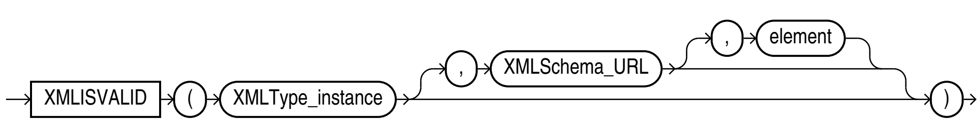
The following example creates an Emp element for a subset of employees, with nested employee_id, last_name, and salary elements as the contents of Emp:

```
SELECT XMLELEMENT("Emp",
   XMLFOREST(e.employee_id, e.last_name, e.salary))
   "Emp Element"
   FROM employees e WHERE employee_id = 204;

Emp Element
----------------------------------------------------------------
<Emp>
  <EMPLOYEE_ID>204</EMPLOYEE_ID>
  <LAST_NAME>Baer</LAST_NAME>
  <SALARY>10000</SALARY>
</Emp>
```

Refer to the example for XMLCOLATTVAL to compare the output of these two functions.

XMLISVALID

Syntax



Purpose

XMLISVALID checks whether the input *XMLType_instance* conforms to the relevant XML schema. It does not change the validation status recorded for *XMLType_instance*.

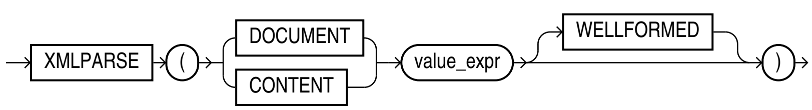
If the input XML document is determined to be valid, then `XMLISVALID` returns 1; otherwise, it returns 0. If you provide *`XMLSchema_URL`* as an argument, then that is used to check conformance. Otherwise, the XML schema specified by the XML document is used to check conformance.

- *`XMLType_instance`* is the XMLType instance to be validated.
- *`XMLSchema_URL`* is the URL of the XML schema against which to check conformance.
- *`element`* is the element of the specified schema against which to check conformance. Use this if you have an XML schema that defines more than one top level element, and you want to check conformance against a specific one of those elements.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for information on the use of this function, including examples

XMLPARSE

Syntax



Purpose

`XMLParse` parses and generates an XML instance from the evaluated result of *`value_expr`*. The *`value_expr`* must resolve to a string. If *`value_expr`* resolves to null, then the function returns null.

- If you specify `DOCUMENT`, then *`value_expr`* must resolve to a singly rooted XML document.
- If you specify `CONTENT`, then *`value_expr`* must resolve to a valid XML value.
- When you specify `WELLFORMED`, you are guaranteeing that *`value_expr`* resolves to a well-formed XML document, so the database does not perform validity checks to ensure that the input is well formed.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

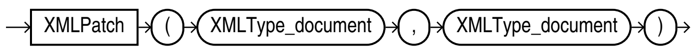
The following example uses the `DUAL` table to illustrate the syntax of `XMLParse`:

```
SELECT XMLPARSE(CONTENT '124 <purchaseOrder poNo="12435">
   <customerName> Acme Enterprises</customerName>
```

```
   <itemNo>32987457</itemNo>
   </purchaseOrder>'
WELLFORMED) AS PO FROM DUAL;

PO
-----------------------------------------------------------------
124 <purchaseOrder poNo="12435">
   <customerName> Acme Enterprises</customerName>
   <itemNo>32987457</itemNo>
   </purchaseOrder>
```

XMLPATCH

Syntax



Purpose

The `XMLPatch` function is the SQL interface for the XmlPatch C API. This function patches an XML document with the changes specified. A patched `XMLType` document is returned.

- For the first argument, specify the name of the input `XMLType` document.
- For the second argument, specify the XMLType document containing the changes to be applied to the first document. The changes should conform to the Xdiff XML schema. You can supply the XML output from the Oracle XML Developer's Kit Java method `diff()`.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide* for more information on using this function, including examples, and *Oracle Database XML C API Reference* for information on the XML APIs for C

Examples

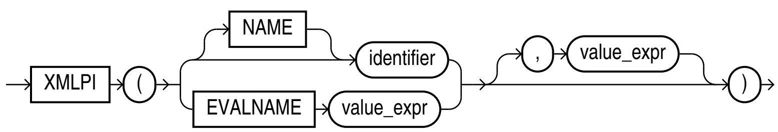
The following example patches an `XMLType` document with the changes specified in another `XMLType` and returns a patched `XMLType` document:

```
SELECT XMLPATCH(
XMLTYPE('<?xml version="1.0"?>
<bk:book xmlns:bk="http://example.com">
   <bk:tr>
        <bk:td>
                <bk:chapter>
                        Chapter 1.
                </bk:chapter>
        </bk:td>
        <bk:td>
                 <bk:chapter>
                        Chapter 2.
```

```
                </bk:chapter>
        </bk:td>
   </bk:tr>
</bk:book>'),
XMLTYPE('<?xml version="1.0"?>
<xd:xdiff xsi:schemaLocation="http://xmlns.oracle.com/xdb/xdiff.xsd
  http://xmlns.oracle.com/xdb/xdiff.xsd"
  xmlns:xd="http://xmlns.oracle.com/xdb/xdiff.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:bk="http://example.com">
  <?oracle-xmldiff operations-in-docorder="true" output-model="snapshot"
    diff-algorithm="global"?>
  <xd:delete-node xd:node-type="element"
   xd:xpath="/bk:book[1]/bk:tr[1]/bk:td[2]/bk:chapter[1]"/>
</xd:xdiff>')
)
FROM DUAL;
```

XMLPI

Syntax



Purpose

`XMLPI` generates an XML processing instruction using *`identifier`* and optionally the evaluated result of *`value_expr`*. A processing instruction is commonly used to provide to an application information that is associated with all or part of an XML document. The application uses the processing instruction to determine how best to process the XML document.

You must specify a value for Oracle Database to use an the enclosing tag. You can do this by specifying *`identifier`*, which is a string literal, or by specifying `EVALNAME` *`value_expr`*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the identifier. The identifier does not have to be a column name or column reference. It cannot be an expression or null. It can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`. See "Extended Data Types" for more information.

The optional *`value_expr`* must resolve to a string. If you omit the optional *`value_expr`*, then a zero-length string is the default. The value returned by the function takes this form:

```
<?identifier string?>
```

`XMLPI` is subject to the following restrictions:

- The *`identifier`* must be a valid target for a processing instruction.
- You cannot specify `xml` in any case combination for *`identifier`*.
- The *`identifier`* cannot contain the consecutive characters `?>`.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

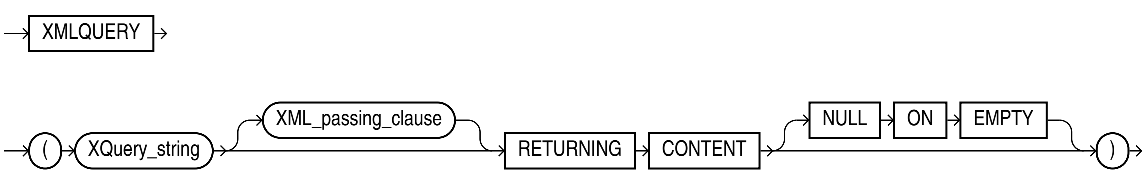
Examples

The following statement uses the `DUAL` table to illustrate the use of the `XMLPI` syntax:

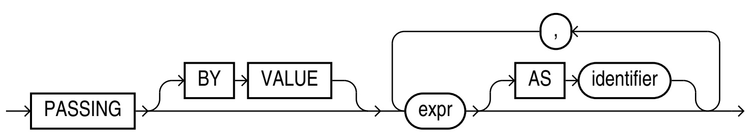
```
SELECT XMLPI(NAME "Order analysisComp", 'imported, reconfigured, disassembled')
   AS "XMLPI" FROM DUAL;

XMLPI
--------------------------------------------------------------------------------
<?Order analysisComp imported, reconfigured, disassembled?>
```

XMLQUERY

Syntax



***XML_passing_clause*::=**



Purpose

`XMLQUERY` lets you query XML data in SQL statements. It takes an XQuery expression as a string literal, an optional context item, and other bind variables and returns the result of evaluating the XQuery expression using these input values.

- *`XQuery_string`* is a complete XQuery expression, including prolog.
- The *`expr`* in the *`XML_passing_clause`* is an expression returning an `XMLType` or an instance of a SQL scalar data type that is used as the context for evaluating the XQuery expression. You can specify only one *`expr`* in the `PASSING` clause without an identifier. The result of evaluating each *`expr`* is bound to the corresponding identifier in the *`XQuery_string`*. If any *`expr`* that is not followed by an `AS` clause, then the result of evaluating that expression is used as the context item for evaluating the *`XQuery_string`*. If *`expr`* is a relational column, then its declared collation is ignored by Oracle XML DB.

- `RETURNING CONTENT` indicates that the result from the XQuery evaluation is either an XML 1.0 document or a document fragment conforming to the XML 1.0 semantics.
- If the result set is empty, then the function returns the SQL `NULL` value. The `NULL ON EMPTY` keywords are implemented by default and are shown for semantic clarity.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

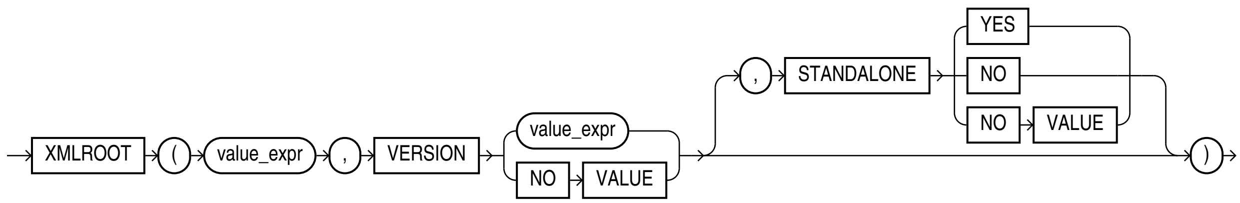
The following statement specifies the *warehouse_spec* column of the `oe.warehouses` table in the *XML_passing_clause* as a context item. The statement returns specific information about the warehouses with area greater than 50K.

```
SELECT warehouse_name,
EXTRACTVALUE(warehouse_spec, '/Warehouse/Area'),
XMLQuery(
   'for $i in /Warehouse
   where $i/Area > 50000
   return <Details>
             <Docks num="{$i/Docks}"/>
             <Rail>
               {
               if ($i/RailAccess = "Y") then "true" else "false"
               }
             </Rail>
          </Details>' PASSING warehouse_spec RETURNING CONTENT) "Big_warehouses"
   FROM warehouses;

WAREHOUSE_ID Area      Big_warehouses
------------ --------- --------------------------------------------------------
           1     25000
           2     50000
           3     85700 <Details><Docks></Docks><Rail>false</Rail></Details>
           4    103000 <Details><Docks num="3"></Docks><Rail>true</Rail></Details>
 . . .
```

XMLROOT

> **Note:**
>
> The `XMLROOT` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you instead use the SQL/XML function `XMLSERIALIZE` with a version number. See *Oracle XML DB Developer's Guide* for more information on the `XMLSERIALIZE` function.

Syntax



Purpose

`XMLROOT` lets you create a new XML value by providing version and standalone properties in the XML root information (prolog) of an existing XML value. If the *`value_expr`* already has a prolog, then the database returns an error. If the input is null, then the function returns null.

The value returned takes the following form:

```
<?xml version = "version" [ STANDALONE = "{yes | no}" ]?>
```

- The first *`value_expr`* specifies the XML value for which you are providing prolog information.
- In the `VERSION` clause, *`value_expr`* must resolve to a string representing a valid XML version. If you specify `NO VALUE` for `VERSION`, then the version defaults to 1.0.
- If you omit the optional `STANDALONE` clause, or if you specify it with `NO VALUE`, then the standalone property is absent from the value returned by the function.

Examples

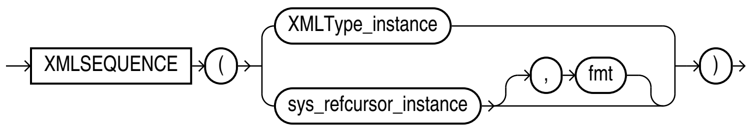
The following statement uses the `DUAL` table to illustrate the syntax of `XMLROOT`:

```
SELECT XMLROOT ( XMLType('<poid>143598</poid>'), VERSION '1.0', STANDALONE YES)
   AS "XMLROOT" FROM DUAL;

XMLROOT
--------------------------------------------------------------------------------
<?xml version="1.0" standalone="yes"?>
<poid>143598</poid>
```

XMLSEQUENCE

> **Note:**
>
> The `XMLSEQUENCE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLTABLE` function instead. See XMLTABLE for more information.

Syntax



Purpose

`XMLSequence` has two forms:

- The first form takes as input an `XMLType` instance and returns a varray of the top-level nodes in the `XMLType`. This form is effectively superseded by the SQL/XML standard function `XMLTable`, which provides for more readable SQL code. Prior to Oracle Database 10g Release 2, `XMLSequence` was used with SQL function `TABLE` to do some of what can now be done better with the `XMLTable` function.
- The second form takes as input a `REFCURSOR` instance, with an optional instance of the `XMLFormat` object, and returns as an `XMLSequence` type an XML document for each row of the cursor.

Because `XMLSequence` returns a collection of `XMLType`, you can use this function in a `TABLE` clause to unnest the collection values into multiple rows, which can in turn be further processed in the SQL query.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function, and XMLTABLE

Examples

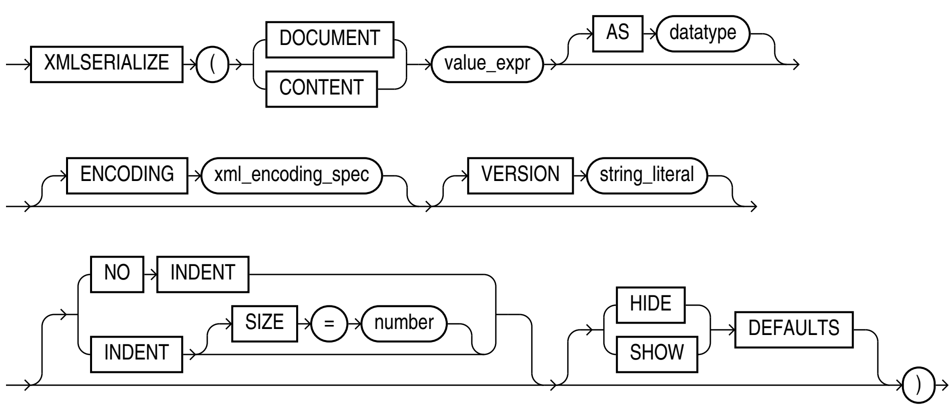
The following example shows how `XMLSequence` divides up an XML document with multiple elements into `VARRAY` single-element documents. In this example, the `TABLE` keyword instructs Oracle Database to consider the collection a table value that can be used in the `FROM` clause of the subquery:

```
SELECT EXTRACT(warehouse_spec, '/Warehouse') as "Warehouse"
   FROM warehouses WHERE warehouse_name = 'San Francisco';

Warehouse
------------------------------------------------------------
<Warehouse>
  <Building>Rented</Building>
  <Area>50000</Area>
  <Docks>1</Docks>
  <DockType>Side load</DockType>
  <WaterAccess>Y</WaterAccess>
  <RailAccess>N</RailAccess>
  <Parking>Lot</Parking>
  <VClearance>12 ft</VClearance>
</Warehouse>
```

```
1 row selected.

SELECT VALUE(p)
   FROM warehouses w,
   TABLE(XMLSEQUENCE(EXTRACT(warehouse_spec, '/Warehouse/*'))) p
   WHERE w.warehouse_name = 'San Francisco';

VALUE(P)
----------------------------------------------------------------
<Building>Rented</Building>
<Area>50000</Area>
<Docks>1</Docks>
<DockType>Side load</DockType>
<WaterAccess>Y</WaterAccess>
<RailAccess>N</RailAccess>
<Parking>Lot</Parking>
<VClearance>12 ft</VClearance>

8 rows selected.
```

XMLSERIALIZE

Syntax



Purpose

`XMLSerialize` creates a string or LOB containing the contents of *`value_expr`*.

- If you specify `DOCUMENT`, then the *`value_expr`* must be a valid XML document.
- If you specify `CONTENT`, then the *`value_expr`* need not be a singly rooted XML document. However it must be valid XML content.
- The *`datatype`* specified can be a string type (`VARCHAR2` or `VARCHAR`, but not `NVARCHAR2`), `BLOB`, or `CLOB`. The default is `CLOB`.
- If *`datatype`* is `BLOB`, then you can specify the `ENCODING` clause to use the specified encoding in the prolog. The *`xml_encoding_spec`* is an XML encoding declaration (`encoding="..."`).
- Specify the `VERSION` clause to use the version you provide as *`string_literal`* in the XML declaration (`<?xml version="..." ...?>`).

- Specify `NO INDENT` to strip all insignificant whitespace from the output. Specify `INDENT SIZE = N`, where *N* is a whole number, for output that is pretty-printed using a relative indentation of *N* spaces. If *N* is `0`, then pretty-printing inserts a newline character after each element, placing each element on a line by itself, but omitting all other insignificant whitespace in the output. If `INDENT` is present without a `SIZE` specification, then 2-space indenting is used. If you omit this clause, then the behavior (pretty-printing or not) is indeterminate.
- `HIDE DEFAULTS` and `SHOW DEFAULTS` apply only to XML schema-based data. If you specify `SHOW DEFAULTS` and the input data is missing any optional elements or attributes for which the XML schema defines default values, then those elements or attributes are included in the output with their default values. If you specify `HIDE DEFAULTS`, then no such elements or attributes are included in the output. `HIDE DEFAULTS` is the default behavior.

See Also:

- *Oracle XML DB Developer's Guide* for more information on this function
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `XMLSERIALIZE`

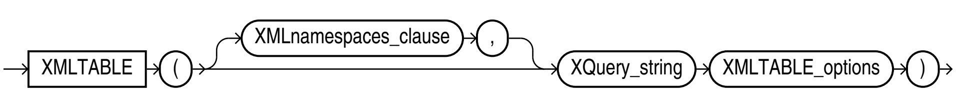
Examples

The following statement uses the `DUAL` table to illustrate the syntax of `XMLSerialize`:

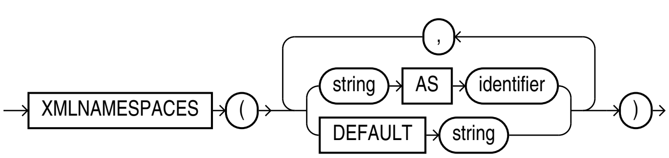
```
SELECT XMLSERIALIZE(CONTENT XMLTYPE('<Owner>Grandco</Owner>')) AS xmlserialize_doc
   FROM DUAL;

XMLSERIALIZE_DOC
----------------
<Owner>Grandco</Owner>
```

XMLTABLE

Syntax

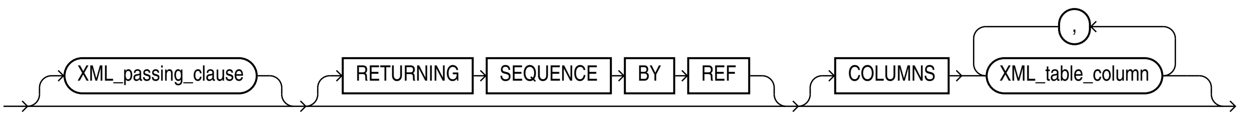


***XMLnamespaces_clause*::=**

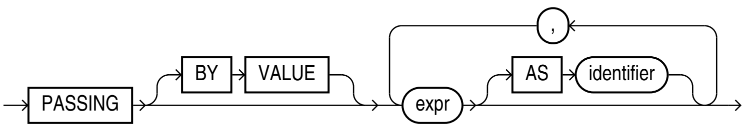


> **Note:**
>
> You can specify at most one `DEFAULT` *`string`* clause.

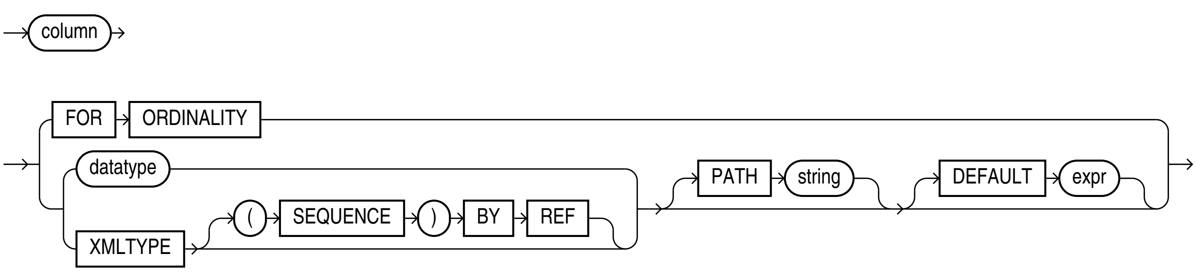
XMLTABLE_options::=



XML_passing_clause::=



XML_table_column::=



Purpose

`XMLTable` maps the result of an XQuery evaluation into relational rows and columns. You can query the result returned by the function as a virtual relational table using SQL.

- The `XMLNAMESPACES` clause contains a set of XML namespace declarations. These declarations are referenced by the XQuery expression (the evaluated *`XQuery_string`*), which computes the row, and by the XPath expression in the `PATH` clause of *`XML_table_column`*, which computes the columns for the entire `XMLTable` function. If you want to use qualified names in the `PATH` expressions of the `COLUMNS` clause, then you need to specify the `XMLNAMESPACES` clause.
- *`XQuery_string`* is a literal string. It is a complete XQuery expression and can include prolog declarations. The value of XQuery_string serves as input to the XMLTable function; it is this XQuery result that is decomposed and stored as relational data.

- The *expr* in the *XML_passing_clause* is an expression returning an XMLType or an instance of a SQL scalar data type that is used as the context for evaluating the XQuery expression. You can specify only one *expr* in the PASSING clause without an identifier. The result of evaluating each *expr* is bound to the corresponding identifier in the *XQuery_string*. If any *expr* that is not followed by an AS clause, then the result of evaluating that expression is used as the context item for evaluating the *XQuery_string*. This clause supports only passing by value, not passing by reference. Therefore, the BY VALUE keywords are optional and are provided for semantic clarity.
- The optional RETURNING SEQUENCE BY REF clause causes the result of the XQuery evaluation to be returned by reference. This allows you to refer to any part of the source data in the *XML_table_column* clause.

  If you omit this clause, then the result of the XQuery evaluation is returned by value. That is, a copy of the targeted nodes is returned instead of a reference to the actual nodes. In this case, you cannot refer to any data that is not in the returned copy in the *XML_table_column* clause. In particular, you cannot refer to data that precedes the targeted nodes in the source data.
- The optional COLUMNS clause defines the columns of the virtual table to be created by XMLTable.
  - If you omit the COLUMNS clause, then XMLTable returns a row with a single XMLType pseudocolumn named COLUMN_VALUE.
  - FOR ORDINALITY specifies that *column* is to be a column of generated row numbers. There must be at most one FOR ORDINALITY clause. It is created as a NUMBER column.
  - For each resulting column except the FOR ORDINALITY column, you must specify the column data type, which can be XMLType or any other data type.

    If the column data type is XMLType, then specify the XMLTYPE clause. If you specify the optional (SEQUENCE) BY REF clause, then a reference to the source data targeted by the PATH expression is returned as the column content. Otherwise, *column* contains a copy of that targeted data.

    Returning the XMLType data by reference lets you specify other columns whose paths target nodes in the source data that are outside those targeted by the PATH expression for column.

    If the column data type is any other data type, then specify *datatype*.
  - The optional PATH clause specifies that the portion of the XQuery result that is addressed by XQuery expression string is to be used as the column content.

    If you omit PATH, then the XQuery expression *column* is assumed. For example:

    ```
    XMLTable(... COLUMNS xyz)
    ```

    is equivalent to

    ```
    XMLTable(... COLUMNS xyz PATH 'XYZ')
    ```

    You can use different PATH clauses to split the XQuery result into different virtual-table columns.
  - The optional DEFAULT clause specifies the value to use when the PATH expression results in an empty sequence. Its *expr* is an XQuery expression that is evaluated to produce the default value.

See Also:

- *Oracle XML DB Developer's Guide* for more information on the `XMLTable` function, including additional examples, and on XQuery in general
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to each character data type column in the table generated by `XMLTABLE`

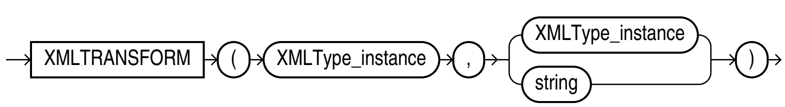
Examples

The following example converts the result of applying the XQuery `'/Warehouse'` to each value in the `warehouse_spec` column of the `warehouses` table into a virtual relational table with columns `Water` and `Rail`:

```
SELECT warehouse_name warehouse,
   warehouse2."Water", warehouse2."Rail"
   FROM warehouses,
   XMLTABLE('/Warehouse'
      PASSING warehouses.warehouse_spec
      COLUMNS
         "Water" varchar2(6) PATH 'WaterAccess',
         "Rail" varchar2(6) PATH 'RailAccess')
      warehouse2;

WAREHOUSE                           Water  Rail
----------------------------------- ------ ------
Southlake, Texas                    Y      N
San Francisco                       Y      N
New Jersey                          N      N
Seattle, Washington                 N      Y
```

XMLTRANSFORM

Syntax



Purpose

`XMLTransform` takes as arguments an `XMLType` instance and an XSL style sheet, which is itself a form of `XMLType` instance. It applies the style sheet to the instance and returns an `XMLType`.

This function is useful for organizing data according to a style sheet as you are retrieving it from the database.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

The `XMLTransform` function requires the existence of an XSL style sheet. Here is an example of a very simple style sheet that alphabetizes elements within a node:

```
CREATE TABLE xsl_tab (col1 XMLTYPE);

INSERT INTO xsl_tab VALUES (
   XMLTYPE.createxml(
   '<?xml version="1.0"?>
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
      <xsl:output encoding="utf-8"/>
      <!-- alphabetizes an xml tree -->
      <xsl:template match="*">
        <xsl:copy>
          <xsl:apply-templates select="*|text()">
            <xsl:sort select="name(.)" data-type="text" order="ascending"/>
          </xsl:apply-templates>
        </xsl:copy>
      </xsl:template>
      <xsl:template match="text()">
        <xsl:value-of select="normalize-space(.)"/>
      </xsl:template>
    </xsl:stylesheet>  '));

1 row created.
```

The next example uses the `xsl_tab` XSL style sheet to alphabetize the elements in one `warehouse_spec` of the sample table `oe.warehouses`:

```
SELECT XMLTRANSFORM(w.warehouse_spec, x.col1).GetClobVal()
   FROM warehouses w, xsl_tab x
   WHERE w.warehouse_name = 'San Francisco';

XMLTRANSFORM(W.WAREHOUSE_SPEC,X.COL1).GETCLOBVAL()
--------------------------------------------------------------------------------
<Warehouse>
  <Area>50000</Area>
  <Building>Rented</Building>
  <DockType>Side load</DockType>
  <Docks>1</Docks>
  <Parking>Lot</Parking>
  <RailAccess>N</RailAccess>
  <VClearance>12 ft</VClearance>
  <WaterAccess>Y</WaterAccess>
</Warehouse>
```

ROUND and TRUNC Date Functions

Table 7-13 lists the format models you can use with the `ROUND` and `TRUNC` date functions and the units to which they round and truncate dates. The default model, 'DD', returns the date rounded or truncated to the day with a time of midnight.

Table 7-13 Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
CC SCC	One greater than the first two digits of a four-digit year
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	Year containing the calendar week, as defined by the ISO 8601 standard
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the calendar week as defined by the ISO 8601 standard, which is Monday
W	Same day of the week as the first day of the month
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter `NLS_TERRITORY`.

See Also:

Oracle Database Reference and *Oracle Database Globalization Support Guide* for information on this parameter

About User-Defined Functions

You can write user-defined functions in PL/SQL, Java, or C to provide functionality that is not available in SQL or SQL built-in functions. User-defined functions can appear in a SQL statement wherever an expression can occur.

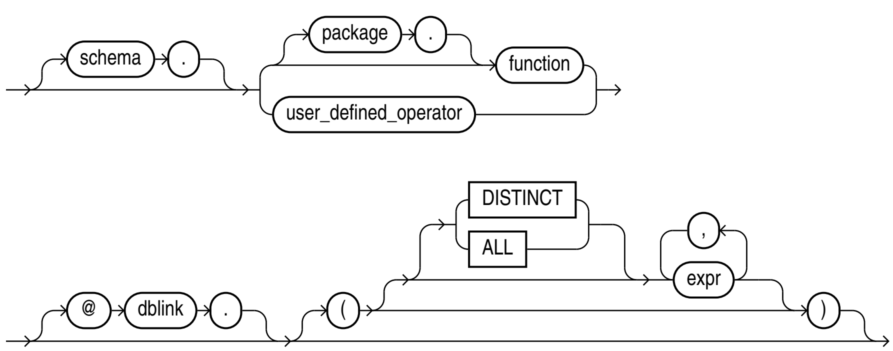
For example, user-defined functions can be used in the following:

- The select list of a `SELECT` statement
- The condition of a `WHERE` clause
- `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses
- The `VALUES` clause of an `INSERT` statement
- The `SET` clause of an `UPDATE` statement

Note:

Oracle SQL does not support calling of functions with Boolean parameters or returns. Therefore, if your user-defined functions will be called from SQL statements, you must design them to return numbers (0 or 1) or character strings ('`TRUE`' or '`FALSE`').

***user_defined_function*::=**



The optional expression list must match attributes of the function, package, or operator.

Restriction on User-defined Functions

The `DISTINCT` and `ALL` keywords are valid only with a user-defined aggregate function.

> **See Also:**
>
> - CREATE FUNCTION for information on creating functions, including restrictions on user-defined functions
> - *Oracle Database Development Guide* for a complete discussion of the creation and use of user functions

Prerequisites

User-defined functions must be created as top-level functions or declared with a package specification before they can be named within a SQL statement.

To use a user function in a SQL expression, you must own or have `EXECUTE` privilege on the user function. To query a view defined with a user function, you must have the `READ` or `SELECT` privilege on the view. No separate `EXECUTE` privileges are needed to select from the view.

> **See Also:**
>
> CREATE FUNCTION for information on creating top-level functions and CREATE PACKAGE for information on specifying packaged functions

Name Precedence

Within a SQL statement, the names of database columns take precedence over the names of functions with no parameters. For example, if the Human Resources manager creates the following two objects in the `hr` schema:

```
CREATE TABLE new_emps (new_sal NUMBER, ...);
CREATE FUNCTION new_sal RETURN NUMBER IS BEGIN ... END;
```

then in the following two statements, the reference to `new_sal` refers to the column `new_emps.new_sal`:

```
SELECT new_sal FROM new_emps;
SELECT new_emps.new_sal FROM new_emps;
```

To access the function `new_sal`, you would enter:

```
SELECT hr.new_sal FROM new_emps;
```

Here are some sample calls to user functions that are allowed in SQL expressions:

```
circle_area (radius)
payroll.tax_rate (empno)
hr.employees.tax_rate (dependent, empno)@remote
```

Example

To call the `tax_rate` user function from schema `hr`, execute it against the `ss_no` and `sal` columns in `tax_table`, specify the following:

```
SELECT hr.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table WHERE ss_no = tax_id;
```

The `INTO` clause is PL/SQL that lets you place the results into the variable `income_tax`.

Naming Conventions

If only one of the optional schema or package names is given, then the first identifier can be either a schema name or a package name. For example, to determine whether `PAYROLL` in the reference `PAYROLL.TAX_RATE` is a schema or package name, Oracle Database proceeds as follows:

1. Check for the `PAYROLL` package in the current schema.
2. If a `PAYROLL` package is not found, then look for a schema name `PAYROLL` that contains a top-level `TAX_RATE` function. If no such function is found, then return an error.
3. If the `PAYROLL` package is found in the current schema, then look for a `TAX_RATE` function in the `PAYROLL` package. If no such function is found, then return an error.

You can also refer to a stored top-level function using any synonym that you have defined for it.

8

Common SQL DDL Clauses

This chapter describes some SQL data definition clauses that appear in multiple SQL statements.

This chapter contains these sections:

- *allocate_extent_clause*
- *constraint*
- *deallocate_unused_clause*
- *file_specification*
- *logging_clause*
- *parallel_clause*
- *physical_attributes_clause*
- *size_clause*
- *storage_clause*

allocate_extent_clause

Purpose

Use the `allocate_extent_clause` clause to explicitly allocate a new extent for a database object.

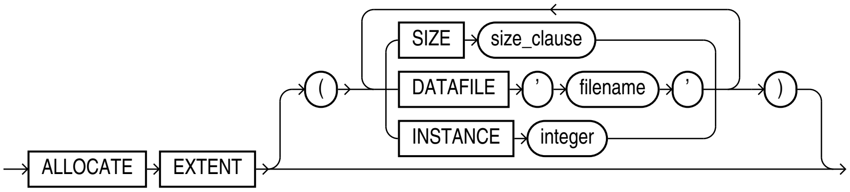
Explicitly allocating an extent with this clause does not change the values of the NEXT and PCTINCREASE storage parameters, so does not affect the size of the next extent to be allocated implicitly by Oracle Database. Refer to *storage_clause* for information about the NEXT and PCTINCREASE storage parameters.

You can allocate an extent in the following SQL statements:

- ALTER CLUSTER (see ALTER CLUSTER)
- ALTER INDEX: to allocate an extent to the index, an index partition, or an index subpartition (see ALTER INDEX)
- ALTER MATERIALIZED VIEW: to allocate an extent to the materialized view, one of its partitions or subpartitions, or the overflow segment of an index-organized materialized view (see ALTER MATERIALIZED VIEW)
- ALTER MATERIALIZED VIEW LOG (see ALTER MATERIALIZED VIEW LOG)
- ALTER TABLE: to allocate an extent to the table, a table partition, a table subpartition, the mapping table of an index-organized table, the overflow segment of an index-organized table, or a LOB storage segment (see ALTER TABLE)

Syntax

***allocate_extent_clause*::=**



(*size_clause*::=)

Semantics

This section describes the parameters of the `allocate_extent_clause`. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

You cannot specify the `allocate_extent_clause` and the `deallocate_unused_clause` in the same statement.

SIZE

Specify the size of the extent in bytes. The value of `integer` can be 0 through 2147483647. To specify a larger extent size, use an integer within this range with `K`, `M`, `G`, or `T` to specify the extent size in kilobytes, megabytes, gigabytes, or terabytes.

For a table, index, materialized view, or materialized view log, if you omit `SIZE`, then Oracle Database determines the size based on the values of the storage parameters of the object. However, for a cluster, Oracle does not evaluate the cluster's storage parameters, so you must specify `SIZE` if you do not want Oracle to use a default value.

DATAFILE '*filename*'

Specify one of the data files in the tablespace of the table, cluster, index, materialized view, or materialized view log to contain the new extent. If you omit `DATAFILE`, then Oracle chooses the data file.

INSTANCE *integer*

Use this parameter only if you are using Oracle Real Application Clusters.

Specifying `INSTANCE` `integer` makes the new extent available to the freelist group associated with the specified instance. If the instance number exceeds the maximum number of freelist groups, then Oracle divides the specified number by the maximum number and uses the remainder to identify the freelist group to be used. An instance is identified by the value of its initialization parameter `INSTANCE_NUMBER`.

If you omit this parameter, then the space is allocated to the table, cluster, index, materialized view, or materialized view log but is not drawn from any particular freelist group. Instead, Oracle uses the master freelist and allocates space as needed.

> **Note:**
>
> If you are using automatic segment-space management, then the `INSTANCE` parameter of the *`allocate_extent_clause`* may not reserve the newly allocated space for the specified instance, because automatic segment-space management does not maintain rigid affinity between extents and instances.

constraint

Purpose

Use a *`constraint`* to define an **integrity constraint—**a rule that restricts the values in a database. Oracle Database lets you create six types of constraints and lets you declare them in two ways.

The six types of integrity constraint are described briefly here and more fully in "Semantics":

- A **`NOT NULL` constraint** prohibits a database value from being null.
- A **unique constraint** prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.
- A **primary key constraint** combines a `NOT NULL` constraint and a unique constraint in a single declaration. It prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null.
- A **foreign key constraint** requires values in one table to match values in another table.
- A **check constraint** requires a value in the database to comply with a specified condition.
- A `REF` column by definition references an object in another object type or in a relational table. A **REF constraint** lets you further describe the relationship between the `REF` column and the object it references.

You can define constraints syntactically in two ways:

- As part of the definition of an individual column or attribute. This is called **inline** specification.
- As part of the table definition. This is called **out-of-line** specification.

`NOT NULL` constraints must be declared inline. All other constraints can be declared either inline or out of line.

Constraint clauses can appear in the following statements:

- `CREATE TABLE` (see CREATE TABLE)
- `ALTER TABLE` (see ALTER TABLE)
- `CREATE VIEW` (see CREATE VIEW)
- `ALTER VIEW` (see ALTER VIEW)

View Constraints

Oracle Database does not enforce view constraints. However, you can enforce constraints on views through constraints on base tables.

You can specify only unique, primary key, and foreign key constraints on views, and they are supported only in `DISABLE NOVALIDATE` mode. You cannot define view constraints on attributes of an object column.

External Table Constraints

You can specify only `NOT NULL`, unique, primary key, and foreign key constraints on external tables. Unique, primary key, and foreign key constraints are supported only in `RELY DISABLE` mode.

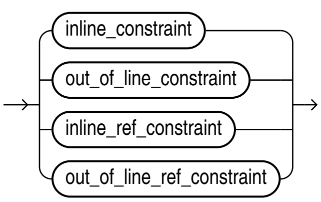
Prerequisites

You must have the privileges necessary to issue the statement in which you are defining the constraint.

To create a foreign key constraint, in addition, the parent table or view must be in your own schema or you must have the `REFERENCES` privilege on the columns of the referenced key in the parent table or view.

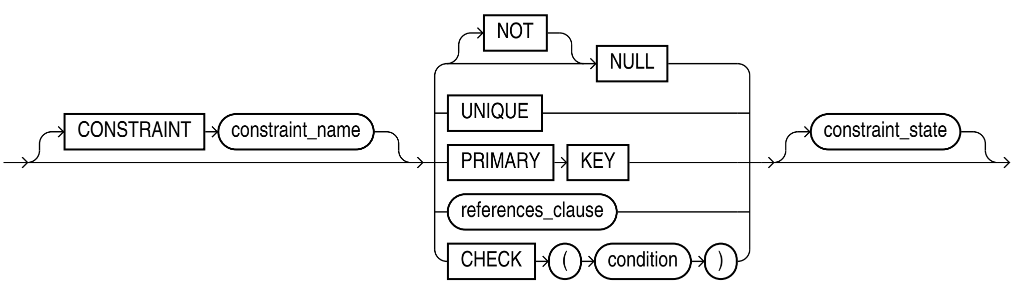
Syntax

***constraint*::=**



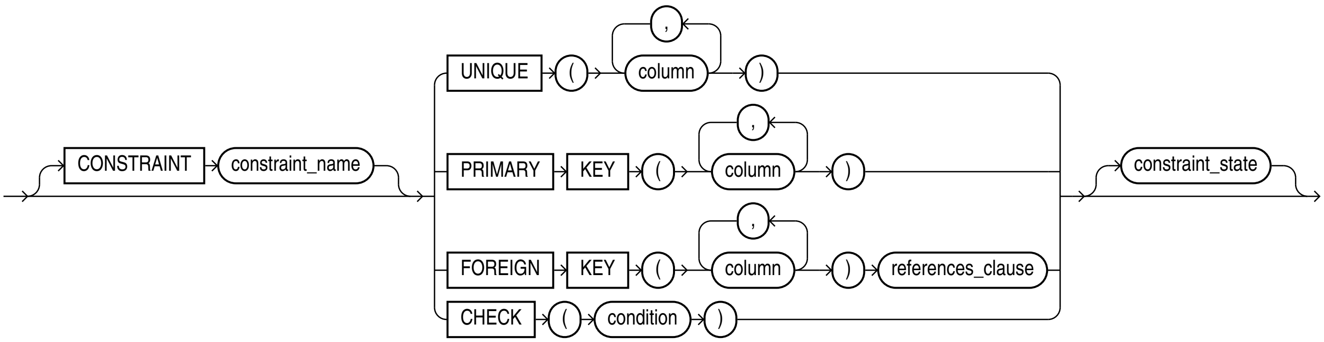
(*inline_constraint*::=, *out_of_line_constraint*::=, *inline_ref_constraint*::=, *out_of_line_ref_constraint*::=)

***inline_constraint*::=**



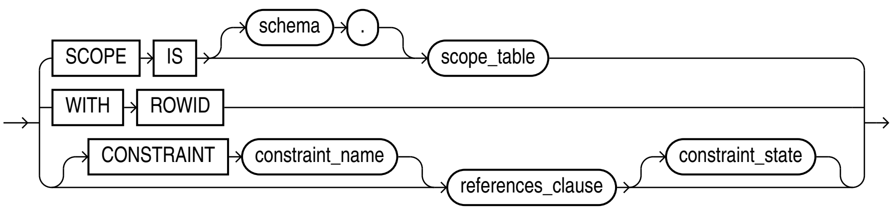
(*references_clause*::=)

out_of_line_constraint::=



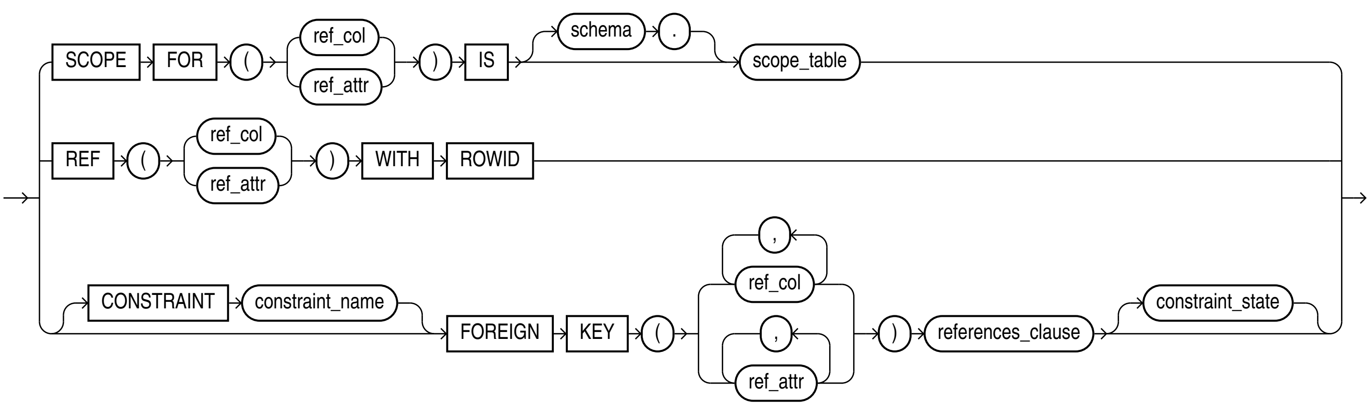
(*references_clause*::=, *constraint_state*::=)

inline_ref_constraint::=



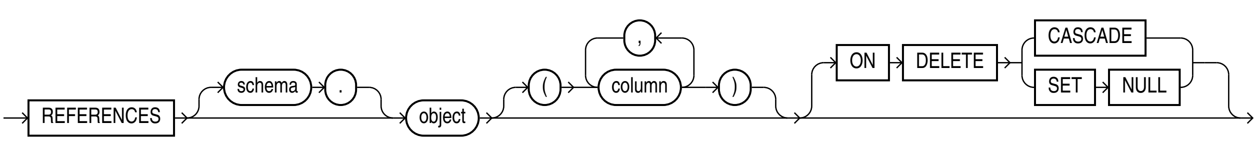
(*references_clause*::=, *constraint_state*::=)

out_of_line_ref_constraint::=

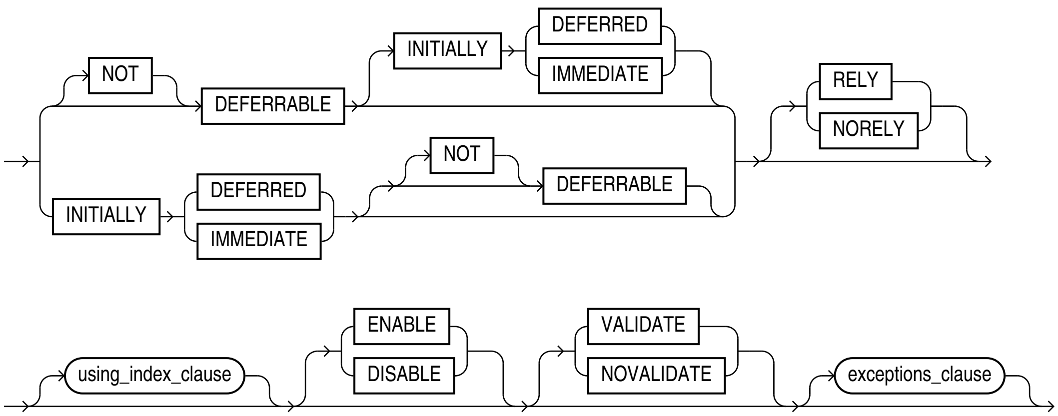


(*references_clause*::=, *constraint_state*::=)

references_clause::=

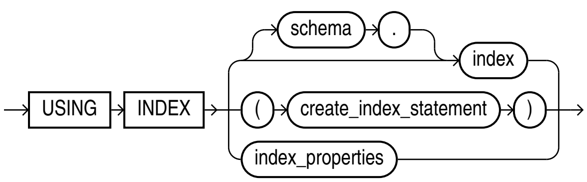


***constraint_state*::=**



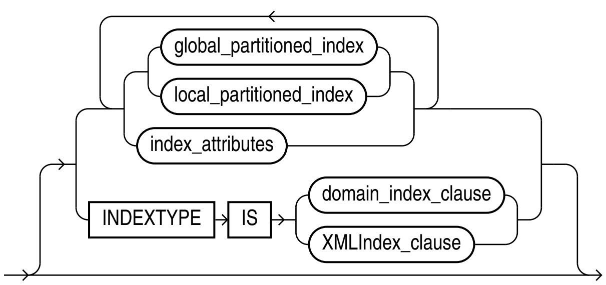
(*using_index_clause*::=, *exceptions_clause*::=)

***using_index_clause*::=**



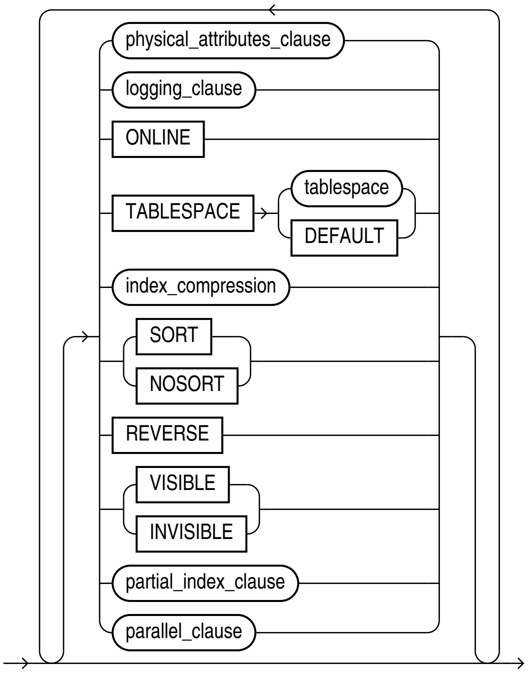
(*create_index*::=, *index_properties*::=)

***index_properties*::=**



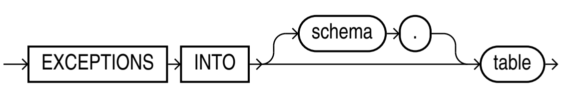
(*global_partitioned_index*::=, *local_partitioned_index*::=--part of `CREATE INDEX`, *index_attributes*::=. The `INDEXTYPE IS` ... clause is not valid when defining a constraint.)

***index_attributes*::=**



(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, *partial_index_clause*::=--all part of `CREATE INDEX`, *`parallel_clause`*: not supported in *`using_index_clause`*)

***exceptions_clause*::=**



Semantics

This section describes the semantics of *`constraint`*. For additional information, refer to the SQL statement in which you define or redefine a constraint for a table or view.

Oracle Database does not support constraints on columns or attributes whose type is a user-defined object, nested table, `VARRAY`, `REF`, or LOB, with two exceptions:

- `NOT NULL` constraints are supported for a column or attribute whose type is user-defined object, `VARRAY`, `REF`, or LOB.
- `NOT NULL`, foreign key, and `REF` constraints are supported on a column of type `REF`.

CONSTRAINT *constraint_name*

Specify a name for the constraint. The name must satisfy the requirements listed in "Database Object Naming Rules ". If you omit this identifier, then Oracle Database generates a name with the form `SYS_C`*`n`*. Oracle stores the name and the definition of the integrity constraint in the USER_, ALL_, and DBA_CONSTRAINTS data dictionary views (in the CONSTRAINT_NAME and SEARCH_CONDITION columns, respectively).

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views

NOT NULL Constraints

A NOT NULL constraint prohibits a column from containing nulls. The NULL keyword by itself does not actually define an integrity constraint, but you can specify it to explicitly permit a column to contain nulls. You must define NOT NULL and NULL using inline specification. If you specify neither NOT NULL nor NULL, then the default is NULL.

NOT NULL constraints are the only constraints you can specify inline on XMLType and VARRAY columns.

To satisfy a NOT NULL constraint, every row in the table must contain a value for the column.

> **Note:**
>
> Oracle Database does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, then you must either specify NOT NULL constraints for at least one of the index key columns or create a bitmap index.

Restrictions on NOT NULL Constraints

NOT NULL constraints are subject to the following restrictions:

- You cannot specify NULL or NOT NULL in a view constraint.
- You cannot specify NULL or NOT NULL for an attribute of an object. Instead, use a CHECK constraint with the IS [NOT] NULL condition.

Unique Constraints

A **unique** constraint designates a column as a unique key. A **composite unique key** designates a combination of columns as the unique key. When you define a unique constraint inline, you need only the UNIQUE keyword. When you define a unique constraint out of line, you must also specify one or more columns. You must define a composite unique key out of line.

To satisfy a unique constraint, no two rows in the table can have the same value for the unique key. However, the unique key made up of a single column can contain nulls. To satisfy a composite unique key, no two rows in the table or view can have the same combination of values in the key columns. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.

Unique constraints are sensitive to declared collations of their key columns. See Collation Sensitivity of Constraints for more details.

When you specify a unique constraint on one or more columns, Oracle implicitly creates an index on the unique key. If you are defining uniqueness for purposes of query performance, then Oracle recommends that you instead create the unique index explicitly using a `CREATE UNIQUE INDEX` statement. You can also use the `CREATE UNIQUE INDEX` statement to create a unique function-based index that defines a conditional unique constraint. See "Using a Function-based Index to Define Conditional Uniqueness: Example" for more information.

When you specify an enabled unique constraint on an extended data type column, you may receive a "maximum key length exceeded" error when Oracle tries to create the index to enforce uniqueness for the enabled constraint. See "Creating an Index on an Extended Data Type Column" for information on how to work around this issue.

Restrictions on Unique Constraints

Unique constraints are subject to the following restrictions:

- None of the columns in the unique key can be of LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `OBJECT`, `REF`, `TIMESTAMP WITH TIME ZONE`, or user-defined type. However, the unique key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- A composite unique key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.
- You cannot specify a unique key when creating a subview in an inheritance hierarchy. The unique key can be specified only for the top-level (root) view.
- When you specify a unique constraint for an external table, you must specify the `RELY` and `DISABLE` constraint states. See External Table Constraints for more information.

> **See Also:**
>
> "Unique Key Example" and Composite Unique Key Example

Primary Key Constraints

A **primary key** constraint designates a column as the primary key of a table or view. A **composite primary key** designates a combination of columns as the primary key. When you define a primary key constraint inline, you need only the `PRIMARY KEY` keywords. When you define a primary key constraint out of line, you must also specify one or more columns. You must define a composite primary key out of line.

To satisfy a primary key constraint:

- No primary key value can appear in more than one row in the table.
- No column that is part of the primary key can contain a null.

When you create a primary key constraint:

- Oracle Database uses an existing index if it contains a unique set of values before enforcing the primary key constraint. The existing index can be defined as unique or nonunique. When a DML operation is performed, the primary key constraint is enforced using this existing index.

- If no existing index can be used, then Oracle Database generates a unique index.

When you drop a primary key constraint:

- If the primary key was created using an existing index, then the index is not dropped.
- If the primary key was created using a system-generated index, then the index is dropped.

When you designate an extended data type column as an enabled primary key, you may receive a "maximum key length exceeded" error when Oracle tries to create the index to enforce uniqueness for the enabled constraint. See "Creating an Index on an Extended Data Type Column" for information on how to work around this issue.

Primary key constraints are sensitive to declared collations of their key columns. See Collation Sensitivity of Constraints for more details.

Restrictions on Primary Key Constraints

Primary constraints are subject to the following restrictions:

- A table or view can have only one primary key.
- None of the columns in the primary key can be LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `BFILE`, `REF`, `TIMESTAMP WITH TIME ZONE`, or user-defined type. However, the primary key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- The size of the primary key cannot exceed approximately one database block.
- A composite primary key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.
- You cannot specify a primary key when creating a subview in an inheritance hierarchy. The primary key can be specified only for the top-level (root) view.
- When you specify a primary key constraint for an external table, you must specify the `RELY` and `DISABLE` constraint states. See External Table Constraints for more information.

> **See Also:**
>
> "Primary Key Example" and "Composite Primary Key Example"

Foreign Key Constraints

A **foreign key constraint** (also called a **referential integrity constraint**) designates a column as the foreign key and establishes a relationship between that foreign key and a specified primary or unique key, called the **referenced key**. A **composite foreign key** designates a combination of columns as the foreign key.

The table or view containing the foreign key is called the **child** object, and the table or view containing the referenced key is called the **parent** object. The foreign key and the referenced key can be in the same table or view. In this case, the parent and child tables are the same. If you identify only the parent table or view and omit the column name, then the foreign key automatically references the primary key of the parent table or view. The corresponding column or columns of the foreign key and the referenced key must match in order, data types, and declared collations.

Foreign key constraints are sensitive to declared collations of the referenced primary or unique key columns. See Collation Sensitivity of Constraints for more details.

You can define a foreign key constraint on a single key column either inline or out of line. You must specify a composite foreign key and a foreign key on an attribute out of line.

To satisfy a composite foreign key constraint, the composite foreign key must refer to a composite unique key or a composite primary key in the parent table or view, or the value of at least one of the columns of the foreign key must be null.

You can designate the same column or combination of columns as both a foreign key and a primary or unique key. You can also designate the same column or combination of columns as both a foreign key and a cluster key.

You can define multiple foreign keys in a table or view. Also, a single column can be part of more than one foreign key.

Restrictions on Foreign Key Constraints

Foreign key constraints are subject to the following restrictions:

- None of the columns in the foreign key can be of LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `BFILE`, `REF`, `TIMESTAMP WITH TIME ZONE`, or user-defined type. However, the primary key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- The referenced unique or primary key constraint on the parent table or view must already be defined.
- A composite foreign key cannot have more than 32 columns.
- The child and parent tables must be on the same database. To enable referential integrity constraints across nodes of a distributed database, you must use database triggers. See CREATE TRIGGER .
- If either the child or parent object is a view, then the constraint is subject to all restrictions on view constraints. See "View Constraints ".
- You cannot define a foreign key constraint in a `CREATE TABLE` statement that contains an `AS` *`subquery`* clause. Instead, you must create the table without the constraint and then add it later with an `ALTER TABLE` statement.
- When a table has a foreign key, and the parent of the foreign key is an index-organized table, a session that updates a row that contains the foreign key can hang when another session is updating a non-key column in the parent table.
- When you specify a foreign key constraint for an external table, you must specify the `RELY` and `DISABLE` constraint states. See External Table Constraints for more information.

See Also:

- *Oracle Database Development Guide* for more information on using constraints
- "Foreign Key Constraint Example" and "Composite Foreign Key Constraint Example"

references_clause

Foreign key constraints use the *references_clause* syntax. When you specify a foreign key constraint inline, you need only the *references_clause*. When you specify a foreign key constraint out of line, you must also specify the FOREIGN KEY keywords and one or more columns.

ON DELETE Clause

The ON DELETE clause lets you determine how Oracle Database automatically maintains referential integrity if you remove a referenced primary or unique key value. If you omit this clause, then Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

- Specify CASCADE if you want Oracle to remove dependent foreign key values.
- Specify SET NULL if you want Oracle to convert dependent foreign key values to NULL. You cannot specify this clause for a virtual column, because the values in a virtual column cannot be updated directly. Rather, the values from which the virtual column are derived must be updated.

Restriction on ON DELETE

You cannot specify this clause for a view constraint.

> **See Also:**
>
> "ON DELETE Example"

Check Constraints

A **check constraint** lets you specify a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null). When Oracle evaluates a check constraint condition for a particular row, any column names in the condition refer to the column values in that row.

The syntax for inline and out-of-line specification of check constraints is the same. However, inline specification can refer only to the column (or the attributes of the column if it is an object column) currently being defined, whereas out-of-line specification can refer to multiple columns or attributes.

Oracle does not verify that conditions of check constraints are not mutually exclusive. Therefore, if you create multiple check constraints for a column, design them carefully so their purposes do not conflict. Do not assume any particular order of evaluation of the conditions.

If the condition of a check constraint depends on NLS parameters, such as NLS_DATE_FORMAT, Oracle evaluates the condition using the database values of the parameters, not the session values. You can find the database values of the NLS parameters in the data dictionary view NLS_DATABASE_PARAMETERS. These values are associated with a database by the DDL statement CREATE DATABASE and never change afterwards.

See Also:

- Conditions for additional information and syntax
- "Check Constraint Examples" and "Attribute-Level Constraints Example"

Restrictions on Check Constraints

Check constraints are subject to the following restrictions:

- You cannot specify a check constraint for a view. However, you can define the view using the `WITH CHECK OPTION` clause, which is equivalent to specifying a check constraint for the view.
- The condition of a check constraint can refer to any column in the table, but it cannot refer to columns of other tables.
- Conditions of check constraints cannot contain the following constructs:
  - Subqueries and scalar subquery expressions
  - Calls to the functions that are not deterministic (`CURRENT_DATE`, `CURRENT_TIMESTAMP`, `DBTIMEZONE`, `LOCALTIMESTAMP`, `SESSIONTIMEZONE`, `SYSDATE`, `SYSTIMESTAMP`, `UID`, `USER`, and `USERENV`)
  - Calls to user-defined functions
  - Dereferencing of `REF` columns (for example, using the `DEREF` function)
  - Nested table columns or attributes
  - The pseudocolumns `CURRVAL`, `NEXTVAL`, `LEVEL`, or `ROWNUM`
  - Date constants that are not fully specified
  - You cannot specify a check constraint for an external table.

REF Constraints

`REF` constraints let you describe the relationship between a column of type `REF` and the object it references.

ref_constraint

`REF` constraints use the *`ref_constraint`* syntax. You define a `REF` constraint either inline or out of line. Out-of-line specification requires you to specify the `REF` column or attribute you are further describing.

- For *`ref_column`*, specify the name of a `REF` column of an object or relational table.
- For *`ref_attribute`*, specify an embedded `REF` attribute within an object column of a relational table.

Both inline and out-of-line specification let you define a scope constraint, a rowid constraint, or a referential integrity constraint on a `REF` column.

If the scope table or referenced table of the `REF` column has a primary-key-based object identifier, then the `REF` column is a **user-defined `REF` column**.

See Also:

- *Oracle Database Object-Relational Developer's Guide* for more information on REF data types
- "Foreign Key Constraints", and "REF Constraint Examples"

SCOPE REF Constraints

In a table with a REF column, each REF value in the column can conceivably reference a row in a different object table. The SCOPE clause restricts the scope of references to a single table, *scope_table*. The values in the REF column or attribute point to objects in *scope_table*, in which object instances of the same type as the REF column are stored.

Specify the SCOPE clause to restrict the scope of references in the REF column to a single table. For you to specify this clause, *scope_table* must be in your own schema, or you must have the READ or SELECT privilege on *scope_table*, or you must have the READ ANY TABLE or SELECT ANY TABLE system privilege. You can specify only one scope table for each REF column.

Restrictions on Scope Constraints

Scope constraints are subject to the following restrictions:

- You cannot add a scope constraint to an existing column unless the table is empty.
- You cannot specify a scope constraint for the REF elements of a VARRAY column.
- You must specify this clause if you specify AS *subquery* and the subquery returns user-defined REF data types.
- You cannot subsequently drop a scope constraint from a REF column.
- You cannot specify a scope constraint for an external table.

Rowid REF Constraints

Specify WITH ROWID to store the rowid along with the REF value in *ref_column* or *ref_attribute*. Storing the rowid with the REF value can improve the performance of dereferencing operations, but will also use more space. Default storage of REF values is without rowids.

See Also:

The function DEREF for an example of dereferencing

Restrictions on Rowid Constraints

Rowid constraints are subject to the following restrictions:

- You cannot define a rowid constraint for the REF elements of a VARRAY column.
- You cannot subsequently drop a rowid constraint from a REF column.

- If the REF column or attribute is scoped, then this clause is ignored and the rowid is not stored with the REF value.
- You cannot specify a rowid constraint for an external table.

Referential Integrity Constraints on REF Columns

The *references_clause* of the *ref_constraint* syntax lets you define a foreign key constraint on the REF column. This clause also implicitly restricts the scope of the REF column or attribute to the referenced table. However, whereas a foreign key constraint on a non-REF column references an actual column in the parent table, a foreign key constraint on a REF column references the implicit object identifier column of the parent table.

If you do not specify a constraint name, then Oracle generates a system name for the constraint of the form SYS_C*n*.

If you add a referential integrity constraint to an existing REF column that is already scoped, then the referenced table must be the same as the scope table of the REF column. If you later drop the referential integrity constraint, then the REF column will remain scoped to the referenced table.

As is the case for foreign key constraints on other types of columns, you can use the *references_clause* alone for inline declaration. For out-of-line declaration you must also specify the FOREIGN KEY keywords plus one or more REF columns or attributes.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information on object identifiers

Restrictions on Foreign Key Constraints on REF Columns

Foreign key constraints on REF columns have the following additional restrictions:

- Oracle implicitly adds a scope constraint when you add a referential integrity constraint to an existing unscoped REF column. Therefore, all the restrictions that apply for scope constraints also apply in this case.
- You cannot specify a column after the object name in the *references_clause*.

Collation Sensitivity of Constraints

Starting with Oracle Database 12*c* Release 2 (12.2), primary key, unique, and foreign key constraints are sensitive to declared collations of their key columns. A primary or unique key character column value from a new or updated row is compared with values in existing rows using the declared collation of the key column. For example, if the declared collation of the key column is the case-insensitive collation BINARY_CI, a new or updated row may be rejected if the new key column value differs from some existing key value only by case. The collation BINARY_CI treats character values differing only by case as equal.

A foreign key character column value is compared to parent primary or unique key column values using the declared collation of the parent key column. For example, if the declared collation of the key column is the case-insensitive collation BINARY_CI, a new or updated child row may be accepted even if there is no identical parent key value for the corresponding foreign key value, provided there exists a value differing only by case.

The declared collation of a foreign key column must be the same as the collation of the corresponding parent key column.

Columns in a composite key of a constraint may have different declared collations.

When the declared collation of a key column of a constraint is a pseudo-collation, the constraint uses a corresponding variant of the collation `BINARY`. Pseudo-collations cannot be used directly to compare values for a constraint, because constraints are static and cannot depend on session NLS parameters on which the pseudo-collations depend. Therefore:

- The pseudo-collations `USING_NLS_COMP`, `USING_NLS_SORT`, and `USING_NLS_SORT_CS` use the collation `BINARY`.
- The pseudo-collation `USING_NLS_COMP_CI` uses the collation `BINARY_CI`.
- The pseudo-collation `USING_NLS_COMP_AI` uses the collation `BINARY_AI`.

When the effective collation used by a primary or unique key column is not `BINARY`, Oracle creates a hidden virtual column for this column. The expression of the virtual column calculates collation keys for character values of the original key column. The primary key or unique constraint is internally created on the virtual column instead of the original column. The virtual column is visible in the data dictionary views of the `*_TAB_COLS` family. For each of these hidden virtual columns, the `COLLATED_COLUMN_ID` of the `*_TAB_COLS` views contains the internal sequence number pointing to the corresponding original key column. The hidden virtual columns count to the 1000-column limit of a table.

> **See Also:**
>
> - Case-Insensitive Constraints Example
> - *Oracle Database Globalization Support Guide* for more details about collations

Specifying Constraint State

As part of constraint definition, you can specify how and when Oracle should enforce the constraint.

constraint_state

You can use *`constraint_state`* with both inline and out-of-line specification. Except for the clauses `DEFERRABLE` and `INITIALLY`, that may be specified in any order, you must specify the rest of the component clauses in the order shown, and each clause only once.

DEFERRABLE Clause

The `DEFERRABLE` and `NOT DEFERRABLE` parameters indicate whether or not, in subsequent transactions, constraint checking can be deferred until the end of the transaction using the `SET CONSTRAINT[S]` statement. If you omit this clause, then the default is `NOT DEFERRABLE`.

- Specify `NOT DEFERRABLE` to indicate that in subsequent transactions you cannot use the `SET CONSTRAINT[S]` clause to defer checking of this constraint until the transaction is committed. The checking of a `NOT DEFERRABLE` constraint can never be deferred to the end of the transaction.

  If you declare a new constraint `NOT DEFERRABLE`, then it must be valid at the time the `CREATE TABLE` or `ALTER TABLE` statement is committed or the statement will fail.

- Specify `DEFERRABLE` to indicate that in subsequent transactions you can use the `SET CONSTRAINT[S]` clause to defer checking of this constraint until a `COMMIT` statement is submitted. If the constraint check fails, then the database returns an error and the transaction is not committed. This setting in effect lets you disable the constraint temporarily while making changes to the database that might violate the constraint until all the changes are complete.

> **Note:**
>
> The optimizer does not consider indexes on deferrable constraints as usable.

You cannot alter the deferrability of a constraint. Whether you specify either of these parameters, or make the constraint `NOT DEFERRABLE` implicitly by specifying neither of them, you cannot specify this clause in an `ALTER TABLE` statement. You must drop the constraint and re-create it.

> **See Also:**
>
> - SET CONSTRAINT[S] for information on setting constraint checking for a transaction
> - *Oracle Database Administrator's Guide* and *Oracle Database Concepts* for more information about deferred constraints
> - "DEFERRABLE Constraint Examples"

Restriction on [NOT] DEFERRABLE

You cannot specify either of these parameters for a view constraint.

INITIALLY Clause

The `INITIALLY` clause establishes the default checking behavior for constraints that are `DEFERRABLE`. The `INITIALLY` setting can be overridden by a `SET CONSTRAINT(S)` statement in a subsequent transaction.

- Specify `INITIALLY IMMEDIATE` to indicate that Oracle should check this constraint at the end of each subsequent SQL statement. If you do not specify `INITIALLY` at all, then the default is `INITIALLY IMMEDIATE`.

  If you declare a new constraint `INITIALLY IMMEDIATE`, then it must be valid at the time the `CREATE TABLE` or `ALTER TABLE` statement is committed or the statement will fail.

- Specify `INITIALLY DEFERRED` to indicate that Oracle should check this constraint at the end of subsequent transactions.

This clause is not valid if you have declared the constraint to be `NOT DEFERRABLE`, because a `NOT DEFERRABLE` constraint is automatically `INITIALLY IMMEDIATE` and cannot ever be `INITIALLY DEFERRED`.

RELY Clause

The `RELY` and `NORELY` parameters specify whether a constraint in `NOVALIDATE` mode is to be taken into account for query rewrite. Specify `RELY` to activate a constraint in `NOVALIDATE` mode for query rewrite in an unenforced query rewrite integrity mode. The constraint is in `NOVALIDATE` mode, so Oracle does not enforce it. The default is `NORELY`.

Unenforced constraints are generally useful only with materialized views and query rewrite. Depending on the `QUERY_REWRITE_INTEGRITY` mode, query rewrite can use only constraints that are in `VALIDATE` mode, or that are in `NOVALIDATE` mode with the `RELY` parameter set, to determine join information.

Restriction on the RELY Clause

You cannot set a nondeferrable `NOT NULL` constraint to `RELY`.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on materialized views and query rewrite

Using Indexes to Enforce Constraints

When defining the state of a unique or primary key constraint, you can specify an index for Oracle to use to enforce the constraint, or you can instruct Oracle to create the index used to enforce the constraint.

using_index_clause

You can specify the *`using_index_clause`* only when enabling unique or primary key constraints. You can specify the clauses of the *`using_index_clause`* in any order, but you can specify each clause only once.

- If you specify *`schema.index`*, then Oracle attempts to enforce the constraint using the specified index. If Oracle cannot find the index or cannot use the index to enforce the constraint, then Oracle returns an error.
- If you specify the *`create_index_statement`*, then Oracle attempts to create the index and use it to enforce the constraint. If Oracle cannot create the index or cannot use the index to enforce the constraint, then Oracle returns an error.
- If you neither specify an existing index nor create a new index, then Oracle creates the index. In this case:
  - The index receives the same name as the constraint.
  - If *`table`* is partitioned, then you can specify a locally or globally partitioned index for the unique or primary key constraint.

Restrictions on the *using_index_clause*

The following restrictions apply to the *`using_index_clause`*:

- You cannot specify this clause for a view constraint.
- You cannot specify this clause for a NOT NULL, foreign key, or check constraint.
- You cannot specify an index (*schema.index*) or create an index (*create_index_statement*) when enabling the primary key of an index-organized table.
- You cannot specify the *parallel_clause* of *index_attributes*.
- The INDEXTYPE IS ... clause of *index_properties* is not valid in the definition of a constraint.

> **See Also:**
>
> - CREATE INDEX for a description of *index_attributes*, the *global_partitioned_index* and *local_partitioned_index* clauses, and for a description of NOSORT and the *logging_clause* in relation to indexes
> - *physical_attributes_clause* and PCTFREE parameters and *storage_clause*
> - "Explicit Index Control Example"

ENABLE Clause

Specify ENABLE if you want the constraint to be applied to the data in the table.

If you enable a unique or primary key constraint, and if no index exists on the key, then Oracle Database creates a unique index. Unless you specify KEEP INDEX when subsequently disabling the constraint, this index is dropped and the database rebuilds the index every time the constraint is reenabled.

You can also avoid rebuilding the index and eliminate redundant indexes by creating new primary key and unique constraints initially disabled. Then create (or use existing) nonunique indexes to enforce the constraint. Oracle does not drop a nonunique index when the constraint is disabled, so subsequent ENABLE operations are facilitated.

- ENABLE VALIDATE specifies that all old and new data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If any row in the table violates the integrity constraint, then the constraint remains disabled and Oracle returns an error. If all rows comply with the constraint, then Oracle enables the constraint. Subsequently, if new data violates the constraint, then Oracle does not execute the statement and returns an error indicating the integrity constraint violation.

  If you place a primary key constraint in ENABLE VALIDATE mode, then the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before entering data into the column and before enabling the primary key constraint of the table.
- ENABLE NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint. This clause does not ensure that existing data in the table complies with the constraint.

If you specify neither VALIDATE nor NOVALIDATE, then the default is VALIDATE.

If you change the state of any single constraint from ENABLE NOVALIDATE to ENABLE VALIDATE, then the operation can be performed in parallel, and does not block reads, writes, or other DDL operations.

Restriction on the ENABLE Clause

You cannot enable a foreign key that references a disabled unique or primary key.

DISABLE Clause

Specify DISABLE to disable the integrity constraint. Disabled integrity constraints appear in the data dictionary along with enabled constraints. If you do not specify this clause when creating a constraint, then Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, because it lets you load large amounts of data while also saving space by not having an index. This setting lets you load data from a nonpartitioned table into a partitioned table using the *exchange_partition_subpart* clause of the ALTER TABLE statement or using SQL*Loader. All other modifications to the table (inserts, updates, and deletes) by other SQL statements are disallowed.

  See Also:

  Oracle Database Data Warehousing Guide for more information on using this setting

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated).

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DISABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

  See Also:

  Oracle Database SQL Tuning Guide for information on when to use this setting

If you specify neither VALIDATE nor NOVALIDATE, then the default is NOVALIDATE.

If you disable a unique or primary key constraint that is using a unique index, then Oracle drops the unique index. Refer to the CREATE TABLE *enable_disable_clause* for additional notes and restrictions.

VALIDATE | NOVALIDATE

The behavior of VALIDATE and NOVALIDATE depends on whether the constraint is enabled or disabled, either explicitly or by default.

Note on Foreign Key Constraints in NOVALIDATE Mode

When a foreign key constraint is in NOVALIDATE mode, if existing data in the table does not comply with the constraint and the QUERY_REWRITE_INTEGRITY parameter is not set to ENFORCED, then the optimizer may use join elimination during queries on the table. In this case, a query may return table rows with noncompliant foreign key values even if the query contains a join condition that should filter out those rows.

Handling Constraint Exceptions

When defining the state of a constraint, you can specify a table into which Oracle places the rowids of all rows violating the constraint.

exceptions_clause

Use the *exceptions_clause* syntax to define exception handling. If you omit *schema*, then Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, then Oracle assumes that the table is named EXCEPTIONS. The EXCEPTIONS table or the table you specify must exist on your local database.

You can create the EXCEPTIONS table using one of these scripts:

- UTLEXCPT.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLEXPT1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own exceptions table, then it must follow the format prescribed by one of these two scripts.

If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), then you must create a separate exceptions table for each index-organized table to accommodate its primary-key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

Restrictions on the *exceptions_clause*

The following restrictions apply to the *exceptions_clause*:

- You cannot specify this clause for a view constraint.
- You cannot specify this clause in a CREATE TABLE statement, because no rowids exist until *after* the successful completion of the statement.

> **See Also:**
>
> – The DBMS_IOT package in *Oracle Database PL/SQL Packages and Types Reference* for information on the SQL scripts
>
> – *Oracle Database Performance Tuning Guide* for information on eliminating migrated and chained rows

View Constraints

Oracle does not enforce view constraints. However, operations on views are subject to the integrity constraints defined on the underlying base tables. This means that you can enforce constraints on views through constraints on base tables.

Notes on View Constraints

View constraints are a subset of table constraints and are subject to the following restrictions:

- You can specify only unique, primary key, and foreign key constraints on views. However, you can define the view using the `WITH CHECK OPTION` clause, which is equivalent to specifying a check constraint for the view.
- View constraints are supported only in `DISABLE NOVALIDATE` mode. You cannot specify any other mode. You must specify the keyword `DISABLE` when you declare the view constraint. You need not specify `NOVALIDATE` explicitly, as it is the default.
- The `RELY` and `NORELY` parameters are optional. View constraints, because they are unenforced, are usually specified with the `RELY` parameter to make them more useful. The `RELY` or `NORELY` keyword must precede the `DISABLE` keyword.
- Because view constraints are not enforced directly, you cannot specify `INITIALLY DEFERRED` or `DEFERRABLE`.
- You cannot specify the *`using_index_clause`*, the *`exceptions_clause`* clause, or the `ON DELETE` clause of the *`references_clause`*.
- You cannot define view constraints on attributes of an object column.

External Table Constraints

Starting with Oracle Database 12*c* Release 2 (12.2), you can specify `NOT NULL`, unique, primary key, and foreign key constraints on external tables.

`NOT NULL` constraints on external tables are enforced and prohibit columns from containing nulls.

Unique, primary key, and foreign key constraints are supported on external tables only in `RELY DISABLE` mode. You must specify the keywords `RELY` and `DISABLE` when you create these constraints. These constraints are declarative and are not enforced. They can increase query performance and reduce resource consumption because more optimizer transformations can be taken into account. In order for the optimizer to utilize these `RELY DISABLE` constraints, the `QUERY_REWRITE_INTEGRITY` initialization parameter must be set to either `trusted` or `stale_tolerated`.

Examples

Unique Key Example

The following statement is a variation of the statement that created the sample table `sh.promotions`. It defines inline and implicitly enables a unique key on the `promo_id` column (other constraints are not shown):

```
CREATE TABLE promotions_var1
    ( promo_id         NUMBER(6)
                       CONSTRAINT promo_id_u  UNIQUE
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    ) ;
```

The constraint promo_id_u identifies the promo_id column as a unique key. This constraint ensures that no two promotions in the table have the same ID. However, the constraint does allow promotions without identifiers.

Alternatively, you can define and enable this constraint out of line:

```
CREATE TABLE promotions_var2
    ( promo_id         NUMBER(6)
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    , CONSTRAINT promo_id_u UNIQUE (promo_id)
   USING INDEX PCTFREE 20
      TABLESPACE stocks
      STORAGE (INITIAL 8M) );
```

The preceding statement also contains the *using_index_clause*, which specifies storage characteristics for the index that Oracle creates to enable the constraint.

Composite Unique Key Example

The following statement defines and enables a composite unique key on the combination of the warehouse_id and warehouse_name columns of the oe.warehouses table:

```
ALTER TABLE warehouses
   ADD CONSTRAINT wh_unq UNIQUE (warehouse_id, warehouse_name)
   USING INDEX PCTFREE 5
   EXCEPTIONS INTO wrong_id;
```

The wh_unq constraint ensures that the same combination of warehouse_id and warehouse_name values does not appear in the table more than once.

The ADD CONSTRAINT clause also specifies other properties of the constraint:

- The USING INDEX clause specifies storage characteristics for the index Oracle creates to enable the constraint.
- The EXCEPTIONS INTO clause causes Oracle to write to the wrong_id table information about any rows currently in the warehouses table that violate the constraint. If the wrong_id exceptions table does not already exist, then this statement will fail.

Primary Key Example

The following statement is a variation of the statement that created the sample table hr.locations. It creates the locations_demo table and defines and enables a primary key on the location_id column (other constraints from the hr.locations table are omitted):

```
CREATE TABLE locations_demo
    ( location_id    NUMBER(4) CONSTRAINT loc_id_pk PRIMARY KEY
    , street_address VARCHAR2(40)
    , postal_code    VARCHAR2(12)
    , city           VARCHAR2(30)
    , state_province VARCHAR2(25)
    , country_id     CHAR(2)
    ) ;
```

The loc_id_pk constraint, specified inline, identifies the location_id column as the primary key of the locations_demo table. This constraint ensures that no two locations in the table have the same location number and that no location identifier is NULL.

Alternatively, you can define and enable this constraint out of line:

```
CREATE TABLE locations_demo
    ( location_id    NUMBER(4)
    , street_address VARCHAR2(40)
    , postal_code    VARCHAR2(12)
    , city           VARCHAR2(30)
    , state_province VARCHAR2(25)
    , country_id     CHAR(2)
    , CONSTRAINT loc_id_pk PRIMARY KEY (location_id));
```

NOT NULL Example

The following statement alters the `locations_demo` table (created in "Primary Key Example") to define and enable a `NOT NULL` constraint on the `country_id` column:

```
ALTER TABLE locations_demo
   MODIFY (country_id CONSTRAINT country_nn NOT NULL);
```

The constraint `country_nn` ensures that no location in the table has a null `country_id`.

Composite Primary Key Example

The following statement defines a composite primary key on the combination of the `prod_id` and `cust_id` columns of the sample table `sh.sales`:

```
ALTER TABLE sales
    ADD CONSTRAINT sales_pk PRIMARY KEY (prod_id, cust_id) DISABLE;
```

This constraint identifies the combination of the `prod_id` and `cust_id` columns as the primary key of the `sales` table. The constraint ensures that no two rows in the table have the same combination of values for the `prod_id` column and `cust_id` columns.

The constraint clause (`PRIMARY KEY`) also specifies the following properties of the constraint:

- The constraint definition does not include a constraint name, so Oracle generates a name for the constraint.
- The `DISABLE` clause causes Oracle to define the constraint but not enable it.

Foreign Key Constraint Example

The following statement creates the `dept_20` table and defines and enables a foreign key on the `department_id` column that references the primary key on the `department_id` column of the `departments` table:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4),
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   CONSTRAINT fk_deptno
                    REFERENCES departments(department_id) );
```

The constraint `fk_deptno` ensures that all departments given for employees in the `dept_20` table are present in the `departments` table. However, employees can have null department numbers, meaning they are not assigned to any department. To ensure that all employees are assigned to a department, you could create a NOT NULL constraint on the department_id column in the dept_20 table in addition to the REFERENCES constraint.

Before you define and enable this constraint, you must define and enable a constraint that designates the department_id column of the departments table as a primary or unique key.

The foreign key constraint definition does not use the FOREIGN KEY clause, because the constraint is defined inline. The data type of the *department_id* column is not needed, because Oracle automatically assigns to this column the data type of the referenced key.

The constraint definition identifies both the parent table and the columns of the referenced key. Because the referenced key is the primary key of the parent table, the referenced key column names are optional.

Alternatively, you can define this foreign key constraint out of line:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4),
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id,
   CONSTRAINT fk_deptno
      FOREIGN  KEY (department_id)
      REFERENCES  departments(department_id) );
```

The foreign key definitions in both variations of this statement omit the ON DELETE clause, causing Oracle to prevent the deletion of a department if any employee works in that department.

ON DELETE Example

This statement creates the dept_20 table, defines and enables two referential integrity constraints, and uses the ON DELETE clause:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4) PRIMARY KEY,
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4) CONSTRAINT fk_mgr
                    REFERENCES employees ON DELETE SET NULL,
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   NUMBER(2)   CONSTRAINT fk_deptno
                    REFERENCES departments(department_id)
                    ON DELETE CASCADE );
```

Because of the first ON DELETE clause, if manager number 2332 is deleted from the employees table, then Oracle sets to null the value of manager_id for all employees in the dept_20 table who previously had manager 2332.

Because of the second ON DELETE clause, Oracle cascades any deletion of a department_id value in the departments table to the department_id values of its dependent rows of the dept_20 table. For example, if Department 20 is deleted from the departments table, then Oracle deletes all of the employees in Department 20 from the dept_20 table.

Composite Foreign Key Constraint Example

The following statement defines and enables a foreign key on the combination of the `employee_id` and `hire_date` columns of the `dept_20` table:

```
ALTER TABLE dept_20
   ADD CONSTRAINT fk_empid_hiredate
   FOREIGN KEY (employee_id, hire_date)
   REFERENCES hr.job_history(employee_id, start_date)
   EXCEPTIONS INTO wrong_emp;
```

The constraint `fk_empid_hiredate` ensures that all the employees in the `dept_20` table have `employee_id` and `hire_date` combinations that exist in the `employees` table. Before you define and enable this constraint, you must define and enable a constraint that designates the combination of the `employee_id` and `hire_date` columns of the `employees` table as a primary or unique key.

The `EXCEPTIONS INTO` clause causes Oracle to write information to the `wrong_emp` table about any rows in the `dept_20` table that violate the constraint. If the `wrong_emp` exceptions table does not already exist, then this statement will fail.

Check Constraint Examples

The following statement creates a `divisions` table and defines a `check` constraint in each column of the table:

```
CREATE TABLE divisions
   (div_no    NUMBER  CONSTRAINT check_divno
              CHECK (div_no BETWEEN 10 AND 99)
              DISABLE,
    div_name  VARCHAR2(9)  CONSTRAINT check_divname
              CHECK (div_name = UPPER(div_name))
              DISABLE,
    office    VARCHAR2(10)  CONSTRAINT check_office
              CHECK (office IN ('DALLAS','BOSTON',
              'PARIS','TOKYO'))
              DISABLE);
```

Each constraint restricts the values of the column in which it is defined:

- `check_divno` ensures that no division numbers are less than 10 or greater than 99.
- `check_divname` ensures that all division names are in uppercase.
- `check_office` restricts office locations to Dallas, Boston, Paris, or Tokyo.

Because each `CONSTRAINT` clause contains the `DISABLE` clause, Oracle only defines the constraints and does not enable them.

The following statement creates the `dept_20` table, defining out of line and implicitly enabling a check constraint:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4) PRIMARY KEY,
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   NUMBER(2),
    CONSTRAINT check_sal CHECK (salary * commission_pct <= 5000));
```

This constraint uses an inequality condition to limit an employee's total commission, the product of `salary` and `commission_pct`, to $5000:

- If an employee has non-null values for both salary and commission, then the product of these values must not exceed $5000 to satisfy the constraint.
- If an employee has a null salary or commission, then the result of the condition is unknown and the employee automatically satisfies the constraint.

Because the constraint clause in this example does not supply a constraint name, Oracle generates a name for the constraint.

The following statement defines and enables a primary key constraint, two foreign key constraints, a `NOT NULL` constraint, and two check constraints:

```
CREATE TABLE order_detail
  (CONSTRAINT pk_od PRIMARY KEY (order_id, part_no),
   order_id    NUMBER
      CONSTRAINT fk_oid
         REFERENCES oe.orders(order_id),
   part_no     NUMBER
      CONSTRAINT fk_pno
         REFERENCES oe.product_information(product_id),
   quantity    NUMBER
      CONSTRAINT nn_qty NOT NULL
      CONSTRAINT check_qty CHECK (quantity > 0),
   cost        NUMBER
      CONSTRAINT check_cost CHECK (cost > 0) );
```

The constraints enable the following rules on table data:

- `pk_od` identifies the combination of the `order_id` and `part_no` columns as the primary key of the table. To satisfy this constraint, no two rows in the table can contain the same combination of values in the `order_id` and the `part_no` columns, and no row in the table can have a null in either the `order_id` or the `part_no` column.
- `fk_oid` identifies the `order_id` column as a foreign key that references the `order_id` column in the `orders` table in the sample schema `oe`. All new values added to the column `order_detail.order_id` must already appear in the column `oe.orders.order_id`.
- `fk_pno` identifies the `product_id` column as a foreign key that references the `product_id` column in the `product_information` table owned by `oe`. All new values added to the column `order_detail.product_id` must already appear in the column `oe.product_information.product_id`.
- `nn_qty` forbids nulls in the `quantity` column.
- `check_qty` ensures that values in the `quantity` column are always greater than zero.
- `check_cost` ensures the values in the cost column are always greater than zero.

This example also illustrates the following points about constraint clauses and column definitions:

- Out-of-line constraint definition can appear before or after the column definitions. In this example, the out-of-line definition of the `pk_od` constraint precedes the column definitions.
- A column definition can contain multiple inline constraint definitions. In this example, the definition of the `quantity` column contains the definitions of both the `nn_qty` and `check_qty` constraints.

- A table can have multiple `CHECK` constraints. Multiple `CHECK` constraints, each with a simple condition enforcing a single business rule, are preferable to a single `CHECK` constraint with a complicated condition enforcing multiple business rules. When a constraint is violated, Oracle returns an error identifying the constraint. Such an error more precisely identifies the violated business rule if the identified constraint enables a single business rule.

Case-Insensitive Constraints Example

The following statements create two tables in a parent-child relationship. The parent table is a product description table and the child table is a product component description table. Unique constraints are defined to assure that product and description values are unambiguous. For illustrative purposes, the product and component ID are case-insensitive character values. (In real-world applications, primary key IDs are usually numeric or case-normalized.)

```
CREATE TABLE products
    ( product_id VARCHAR2(20) COLLATE BINARY_CI
          CONSTRAINT product_pk PRIMARY KEY
    , description VARCHAR2(1000) COLLATE BINARY_CI
          CONSTRAINT product_description_unq UNIQUE
    );

CREATE TABLE product_components
    ( component_id VARCHAR2(40) COLLATE BINARY_CI
          CONSTRAINT product_component_pk PRIMARY KEY
    , product_id CONSTRAINT product_component_fk REFERENCES products(product_id)
    , description VARCHAR2(1000) COLLATE BINARY_CI
          CONSTRAINT product_component_descr_unq UNIQUE
    );
```

Note that if you do not specify the data type or the collation for a foreign key column, then they are inherited from the parent key column.

The following statements add a product and its components into the tables:

```
INSERT INTO products(product_id, description)
            VALUES('BICY0001', 'Men''s bicycle, fr 21", wh 24", gear 3x7');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_FRAME01', 'BICY0001', 'Aluminium frame 21"');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_WHEEL01', 'bicy0001', 'Wheels 24"');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_GEAR01', 'Bicy0001', 'Front derailleur 3
chainrings');
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_gear02', 'BiCy0001', 'Rear derailleur 7
chainrings');
```

Note the different case of the product ID in different component rows. Because the primary key on the product ID is declared as case-insensitive, all possible letter case combinations of the same ID are considered equal.

The following statement demonstrates that it is not possible to enter another product with the same description differing only by case. It fails with the error `ORA-00001: unique constraint (`*`schema`*`.PRODUCT_DESCRIPTION_UNQ) violated`.

```
INSERT INTO products(product_id, description)
            VALUES('BICY0002', 'MEN''S BICYCLE, fr 21", wh 24", gear 3x7');
```

Similarly, the following statement demonstrates that the primary key contraint of the product table is case-insensitive and does not allow values differing only by case. It fails with the error `ORA-00001: unique constraint (schema.PRODUCT_PK) violated`.

```
INSERT INTO products(component_id, product_id, description)
            VALUES('bicy0001', 'Women''s bicycle, fr 21", wh 24", gear 2x6');
```

The following statement demonstrates that it is not possible to enter another component with the same description differing only by case. It fails with the error `ORA-00001: unique constraint (schema.PRODUCT_COMPONENT_DESCR_UNQ) violated`.

```
INSERT INTO product_components(component_id, product_id, description)
            VALUES('BICY0001_gear03', 'BiCy0001', 'REAR DERAILLEUR 7 CHAINRINGS');
```

Attribute-Level Constraints Example

The following example guarantees that a value exists for both the `first_name` and `last_name` attributes of the `name` column in the `students` table:

```
CREATE TYPE person_name AS OBJECT
   (first_name VARCHAR2(30), last_name VARCHAR2(30));
/

CREATE TABLE students (name person_name, age INTEGER,
   CHECK (name.first_name IS NOT NULL AND
          name.last_name IS NOT NULL));
```

REF Constraint Examples

The following example creates a duplicate of the sample schema object type `cust_address_typ`, and then creates a table containing a `REF` column with a `SCOPE` constraint:

```
CREATE TYPE cust_address_typ_new AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
/
CREATE TABLE address_table OF cust_address_typ_new;

CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ_new
   SCOPE IS address_table);
```

The following example creates the same table but with a referential integrity constraint on the `REF` column that references the object identifier column of the parent table:

```
CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ REFERENCES address_table);
```

The following example uses the type `department_typ` and the table `departments_obj_t`, created in "Creating Object Tables: Examples". A table with a scoped `REF` is then created.

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ SCOPE IS departments_obj_t );
```

The following statement creates a table with a REF column which has a referential integrity constraint defined on it:

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ REFERENCES departments_obj_t);
```

Explicit Index Control Example

The following statement shows another way to create a unique (or primary key) constraint that gives you explicit control over the index (or indexes) Oracle uses to enforce the constraint:

```
CREATE TABLE promotions_var3
    ( promo_id         NUMBER(6)
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    , CONSTRAINT promo_id_u UNIQUE (promo_id, promo_cost)
         USING INDEX (CREATE UNIQUE INDEX promo_ix1
            ON promotions_var3 (promo_id, promo_cost))
    , CONSTRAINT promo_id_u2 UNIQUE (promo_cost, promo_id)
         USING INDEX promo_ix1);
```

This example also shows that you can create an index for one constraint and use that index to create and enable another constraint in the same statement.

DEFERRABLE Constraint Examples

The following statement creates table games with a NOT DEFERRABLE INITIALLY IMMEDIATE constraint check (by default) on the scores column:

```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

To define a unique constraint on a column as INITIALLY DEFERRED DEFERRABLE, issue the following statement:

```
CREATE TABLE games
  (scores NUMBER, CONSTRAINT unq_num UNIQUE (scores)
   INITIALLY DEFERRED DEFERRABLE);
```

deallocate_unused_clause

Purpose

Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of a database object segment and make the space available for other segments in the tablespace.

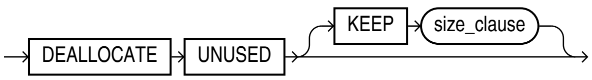
You can deallocate unused space using the following statements:

- ALTER CLUSTER (see ALTER CLUSTER)
- ALTER INDEX: to deallocate unused space from the index, an index partition, or an index subpartition (see ALTER INDEX)

- `ALTER MATERIALIZED VIEW`: to deallocate unused space from the overflow segment of an index-organized materialized view (see ALTER MATERIALIZED VIEW)
- `ALTER TABLE`: to deallocate unused space from the table, a table partition, a table subpartition, the mapping table of an index-organized table, the overflow segment of an index-organized table, or a LOB storage segment (see ALTER TABLE)

Syntax

***deallocate_unused_clause*::=**



(*size_clause*::=)

Semantics

This section describes the semantics of the *`deallocate_unused_clause`*. For additional information, refer to the SQL statement in which you set or reset this clause for a particular database object.

You cannot specify both the *`deallocate_unused_clause`* and the *`allocate_extent_clause`* in the same statement.

Oracle Database frees only unused space above the high water mark (the point beyond which database blocks have not yet been formatted to receive data). Oracle deallocates unused space beginning from the end of the object and moving toward the beginning of the object to the high water mark.

If an extent is completely contained in the deallocation, then the whole extent is freed for reuse. If an extent is partially contained in the deallocation, then the used part up to the high water mark becomes the extent, and the remaining unused space is freed for reuse.

Oracle credits the amount of the released space to the user quota for the tablespace in which the deallocation occurs.

The exact amount of space freed depends on the values of the `INITIAL`, `MINEXTENTS`, and `NEXT` storage parameters. Refer to the *storage_clause* for a description of these parameters.

KEEP *integer*

Specify the number of bytes above the high water mark that the segment of the database object is to have after deallocation.

- If you omit `KEEP` and the high water mark is above the size of `INITIAL` and `MINEXTENTS`, then all unused space above the high water mark is freed. When the high water mark is less than the size of `INITIAL` or `MINEXTENTS`, then all unused space above `MINEXTENTS` is freed.
- If you specify `KEEP`, then the specified amount of space is kept and the remaining space is freed. When the remaining number of extents is less than `MINEXTENTS`, then Oracle adjusts `MINEXTENTS` to the new number of extents. If the initial extent becomes smaller than `INITIAL`, then Oracle adjusts `INITIAL` to the new size.
- In either case, Oracle sets the value of the `NEXT` storage parameter to the size of the last extent that was deallocated.

file_specification

Purpose

Use one of the `file_specification` forms to specify a file as a data file or temp file, or to specify a group of one or more files as a redo log file group. If you are storing your files in Oracle Automatic Storage Management (Oracle ASM) disk groups, then you can further specify the file as a disk group file.

A `file_specification` can appear in the following statements:

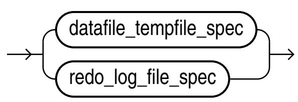
- `CREATE CONTROLFILE` (see CREATE CONTROLFILE)
- `CREATE DATABASE` (see CREATE DATABASE)
- `ALTER DATABASE` (see ALTER DATABASE)
- `CREATE TABLESPACE` (see CREATE TABLESPACE)
- `ALTER TABLESPACE` (see ALTER TABLESPACE)
- `ALTER DISKGROUP` (see ALTER DISKGROUP)

Prerequisites

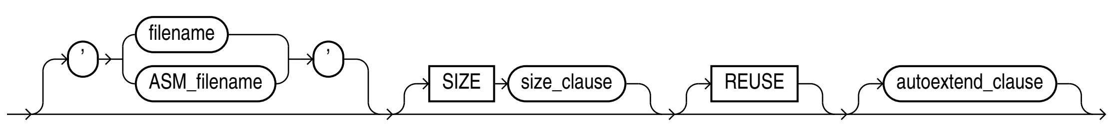
You must have the privileges necessary to issue the statement in which the file specification appears.

Syntax

***file_specification*::=**

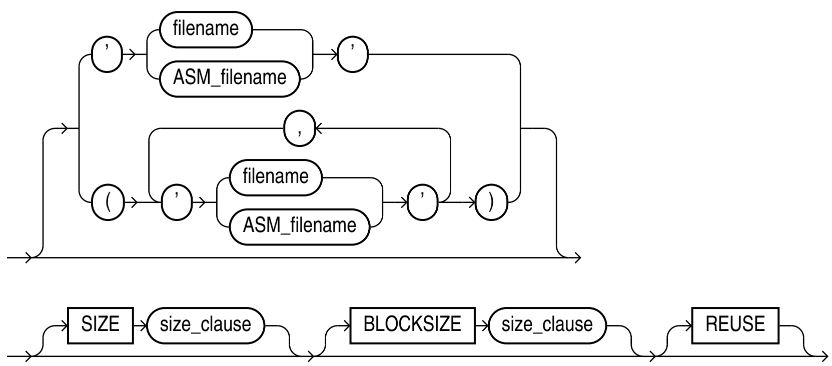


***datafile_tempfile_spec*::=**



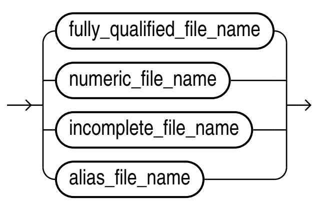
(*size_clause*::=)

***redo_log_file_spec*::=**

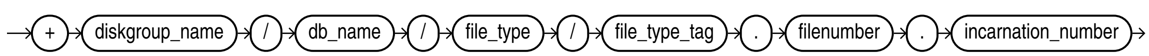


(*size_clause::=*)

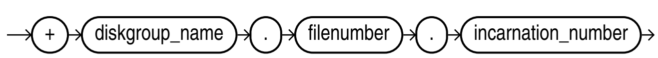
***ASM_filename*::=**



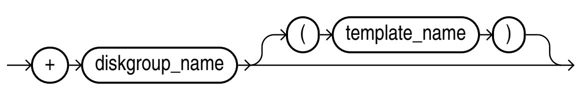
***fully_qualified_file_name*::=**



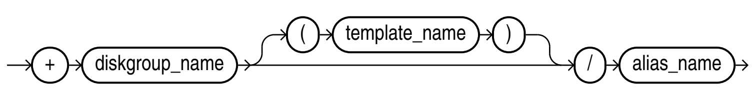
***numeric_file_name*::=**



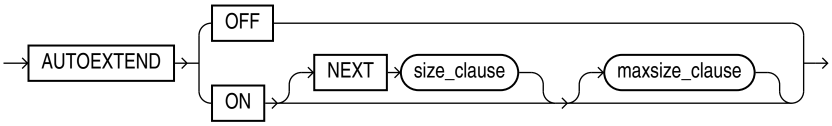
***incomplete_file_name*::=**



***alias_file_name*::=**

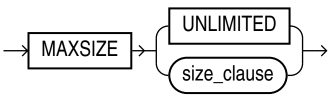


***autoextend_clause*::=**



(*size_clause*::=)

***maxsize_clause*::=**



(*size_clause*::=)

Semantics

This section describes the semantics of *file_specification*. For additional information, refer to the SQL statement in which you specify a data file, temp file, redo log file, or Oracle ASM disk group or disk group file.

datafile_tempfile_spec

Use this clause to specify the attributes of data files and temp files if your database storage is in a file system or in Oracle ASM disk groups.

redo_log_file_spec

Use this clause to specify the attributes of redo log files if your database storage is in a file system or in Oracle ASM disk groups.

filename

Use *filename* for files stored in a file system. The *filename* can specify either a new file or an existing file. For a *new* file:

- If you are *not* using Oracle Managed Files, then you must specify both *filename* and the SIZE clause or the statement fails. When you specify a filename without a size, Oracle attempts to reuse an existing file and returns an error if the file does not exist.
- If you *are* using Oracle Managed Files, then *filename* is optional, as are the remaining clauses of the specification. In this case, Oracle Database creates a unique name for the file and saves it in the directory specified by one of the following initialization parameters:
  – The DB_RECOVERY_FILE_DEST (for logfiles and control files)
  – The DB_CREATE_FILE_DEST initialization parameter (for any type of file)
  – The DB_CREATE_ONLINE_LOG_DEST_*n* initialization parameter, which takes precedence over DB_CREATE_FILE_DEST and DB_RECOVERY_FILE_DEST for log files.

For an *existing* file, specify the name of either a data file, temp file, or a redo log file member. The `filename` can contain only single-byte characters from 7-bit ASCII or EBCDIC character sets. Multibyte characters are not valid.

The filename can include a path prefix. If you do not specify such a path prefix, then the database adds the path prefix for the default storage location, which is platform dependent.

A redo log file group can have one or more members (copies). Each `filename` must be fully specified according to the conventions for your operating system.

The way the database interprets `filename` also depends on whether you specify it with the `SIZE` and `REUSE` clauses.

- If you specify `filename` only, or with the `REUSE` clause but without the `SIZE` clause, then the file must already exist.
- If you specify `filename` with `SIZE` but without `REUSE`, then the file must be a new file.
- If you specify `filename` with both `SIZE` and `REUSE`, then the file can be either new or existing. If the file exists, then it is reused with the new size. If it does not exist, then the database ignores the `REUSE` keyword and creates a new file of the specified size.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information on Oracle Managed Files, "Specifying a Data File: Example", and "Specifying a Log File: Example"

ASM_filename

Use a form of `ASM_filename` for files stored in Oracle ASM disk groups. You can create or refer to data files, temp files, and redo log files with this syntax.

All forms of `ASM_filename` begin with the plus sign (+) followed by the name of the disk group. You can determine the names of all Oracle ASM disk groups by querying the `V$ASM_DISKGROUP` view.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for information on using Oracle ASM

fully_qualified_file_name

When you create a file in an Oracle ASM disk group, the file receives a system-generated fully qualified Oracle ASM filename. You can use this form only when referring to an existing Oracle ASM file. Therefore, if you are using this form during file creation, you must also specify `REUSE`.

- `db_name` is the value of the `DB_UNIQUE_NAME` initialization parameter. This name is equivalent to the name of the database on which the file resides, but the parameter distinguishes between primary and standby databases, if both exist.

- *file_type* and *file_type_tag* indicate the type of database file. Table 8-1 lists all of the file types and their corresponding Oracle ASM tags.
- *filenumber* and *incarnation_number* are system-generated identifiers to guarantee uniqueness.

You can determine the fully qualified names of Oracle ASM files by querying the dynamic performance view appropriate for the file type (for example V$DATAFILE for data files, V$CONTROLFILE for control files, and so on). You can also obtain the *filenumber* and *incarnation_number* portions of the fully qualified names by querying the V$ASM_FILE view.

Table 8-1 Oracle File Types and Oracle ASM File Type Tags

Oracle ASM *file_type*	Description	Oracle ASM *file_type_tag*	Comments
CONTROLFILE	Control files and backup control files	Current Backup	—
DATAFILE	Data files and data file copies	*tsname*	Tablespace into which the file is added
ONLINELOG	Online logs	group_*group#*	—
ARCHIVELOG	Archive logs	thread_*thread#*_seq_*sequence#*	—
TEMPFILE	Temp files	*tsname*	Tablespace into which the file is added
BACKUPSET	Data file and archive log backup pieces; data file incremental backup pieces	*hasspfile_timestamp*	*hasspfile* can take one of two values: *s* indicates that the backup set includes the spfile; *n* indicates that the backup set does not include the spfile.
PARAMETERFILE	Persistent parameter files	spfile	—
DATAGUARDCONFIG	Data Guard configuration file	*db_unique_name*	Data Guard uses the value of the DB_UNIQUE_NAME initialization parameter.
FLASHBACK	Flashback logs	log_*log#*	—
CHANGETRACKING	Block change tracking data	ctf	Used during incremental backups
DUMPSET	Data Pump dumpset	user_*obj#*_*file#*	Dump set files encode the user name, the job number that created the dump set, and the file number as part of the tag.
XTRANSPORT	Data file convert	*tsname*	—
AUTOBACKUP	Automatic backup files	*hasspfile_timestamp*	*hasspfile* can take one of two values: *s* indicates that the backup set includes the spfile; *n* indicates that the backup set does not include the spfile.

numeric_file_name

A numeric Oracle ASM filename is similar to a fully qualified filename except that it uses only the unique *filenumber.incarnation_number* string. You can use this form only to refer to an existing file. Therefore, if you are using this form during file creation, you must also specify REUSE.

incomplete_file_name

Incomplete Oracle ASM filenames are used during file creation only. If you specify the disk group name alone, then Oracle ASM uses the appropriate default template for the file type. For example, if you are creating a data file in a CREATE TABLESPACE statement, Oracle ASM uses the default DATAFILE template to create an Oracle ASM data file. If you specify the disk group name with a template, then Oracle ASM uses the specified template to create the file. In both cases, Oracle ASM also creates a fully qualified filename.

template_name

A template is a named collection of attributes. You can create templates and apply them to files in a disk group. You can determine the names of all Oracle ASM template names by querying the V$ASM_TEMPLATE data dictionary view. Refer to *diskgroup_template_clauses* for instructions on creating Oracle ASM templates.

You can specify *template* only during file creation. It appears in the incomplete and alias name forms of the *ASM_filename* diagram:

- If you specify *template* immediately after the disk group name, then Oracle ASM uses the specified template to create the file, and gives the file a fully qualified filename.
- If you specify template after specifying an alias, then Oracle ASM uses the specified template to create the file, gives the file a fully qualified filename, and also creates the alias so that you can subsequently use it to refer to the file. If the alias you specify refers to an existing file, then Oracle ASM ignores the template specification unless you also specify REUSE.

> **See Also:**
>
> *diskgroup_template_clauses* for information about the default templates

alias_file_name

An alias is a user-friendly name for an Oracle ASM file. You can use alias filenames during file creation or reference. You can specify a template with an alias, but only during file creation. To determine the alias names for Oracle ASM files, query the V$ASM_ALIAS data dictionary view.

If you are specifying an alias during file creation, then refer to *diskgroup_directory_clauses* and *diskgroup_alias_clauses* for instructions on specifying the full alias name.

SIZE Clause

Specify the size of the file in bytes. Use K, M, G, or T to specify the size in kilobytes, megabytes, gigabytes, or terabytes.

- For undo tablespaces, you must specify the `SIZE` clause for each data file. For other tablespaces, you can omit this parameter if the file already exists, or if you are creating an Oracle Managed File.
- If you omit this clause when creating an Oracle Managed File, then Oracle creates a 100M file.
- The size of a tablespace must be one block greater than the sum of the sizes of the objects contained in it.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces and "Adding a Log File: Example"

BLOCKSIZE Clause

Specify `BLOCKSIZE` to override the operating system-dependent sector size. If you omit this clause, then the database uses the operating system-dependent sector size as the block size.

When you add a redo log file to a 512-byte sector disk or to a 4KB sector disk with 512-byte emulation, the blocksize of the new file must be the original platform base block size or 4KB.

- If the redo log file is being added to a 512-byte sector disk, then you must specify 512 or 1024 (or 1K) as the block size, depending on your platform.
- If the redo log file is being added to a 4KB sector disk (native), then you must specify either 4096 or 4K as the block size.
- If the redo log file is being added to a 4KB sector disk with 512-byte emulation, then you can specify either 512, 1024 (or 1K), or 4096 (or 4K) as the block size, depending on your platform.

All logs within a log group must have the same block size. Two log groups created on separate disks can have different block sizes. However, the mixed configuration introduces overhead at every log switch. Oracle recommends that you create all log files with the same block size.

This clause is useful when the 4K sector size is in use, but you want to optimize disk space use rather than performance. In such a case you can override the operating system sector size by specifying `BLOCKSIZE` 512 or, for HP-UX, `BLOCKSIZE` 1024.

> **See Also:**
>
> "Adding a Log File: Example"

REUSE

Specify `REUSE` to allow Oracle to reuse an existing file.

- If the file already exists, then Oracle reuses the filename and applies the new size (if you specify `SIZE`) or retains the original size.
- If the file does not exist, then Oracle ignores this clause and creates the file.

Restriction on the REUSE Clause

You cannot specify `REUSE` unless you have specified *`filename`*.

Whenever Oracle uses an existing file, the previous contents of the file are lost.

> **See Also:**
>
> "Adding a Data File: Example" and "Adding a Log File: Example"

autoextend_clause

The *`autoextend_clause`* is valid for data files and temp files but not for redo log files. Use this clause to enable or disable the automatic extension of a new or existing data file or temp file. If you omit this clause, then:

- For Oracle Managed Files:
  - If you specify `SIZE`, then Oracle Database creates a file of the specified size with `AUTOEXTEND` disabled.
  - If you do not specify `SIZE`, then the database creates a 100M file with `AUTOEXTEND` enabled. When autoextension is required, the database extends the file by its original size or 100MB, whichever is smaller. You can override this default behavior by specifying the `NEXT` clause.
- For user-managed files, with or without `SIZE` specified, Oracle creates a file with `AUTOEXTEND` disabled.

ON

Specify `ON` to enable autoextend.

OFF

Specify `OFF` to turn off autoextend if is turned on. When you turn off autoextend, the values of `NEXT` and `MAXSIZE` are set to zero. If you turn autoextend back on in a subsequent statement, then you must reset these values.

NEXT

Use the `NEXT` clause to specify the size in bytes of the next increment of disk space to be allocated automatically when more extents are required. The default is the size of one data block.

MAXSIZE

Use the `MAXSIZE` clause to specify the maximum disk space allowed for automatic extension of the data file.

UNLIMITED

Use the `UNLIMITED` clause if you do not want to limit the disk space that Oracle can allocate to the data file or temp file.

Restriction on the *autoextend_clause*

You cannot specify this clause as part of the *datafile_tempfile_spec* in a CREATE CONTROLFILE statement or in an ALTER DATABASE CREATE DATAFILE clause.

Examples

Specifying a Log File: Example

The following statement creates a database named payable that has two redo log file groups, each with two members, and one data file:

```
CREATE DATABASE payable
   LOGFILE GROUP 1 ('diska:log1.log', 'diskb:log1.log') SIZE 50K,
           GROUP 2 ('diska:log2.log', 'diskb:log2.log') SIZE 50K
   DATAFILE 'diskc:dbone.dbf' SIZE 30M;
```

The first file specification in the LOGFILE clause specifies a redo log file group with the GROUP value 1. This group has members named 'diska:log1.log' and 'diskb:log1.log', each 50 kilobytes in size.

The second file specification in the LOGFILE clause specifies a redo log file group with the GROUP value 2. This group has members named 'diska:log2.log' and 'diskb:log2.log', also 50 kilobytes in size.

The file specification in the DATAFILE clause specifies a data file named 'diskc:dbone.dbf', 30 megabytes in size.

Each file specification specifies a value for the SIZE parameter and omits the REUSE clause, so none of these files can already exist. Oracle must create them.

Adding a Log File: Example

The following statement adds another redo log file group with two members to the payable database:

```
ALTER DATABASE payable
   ADD LOGFILE GROUP 3 ('diska:log3.log', 'diskb:log3.log')
   SIZE 50K REUSE;
```

The file specification in the ADD LOGFILE clause specifies a new redo log file group with the GROUP value 3. This new group has members named 'diska:log3.log' and 'diskb:log3.log', each 50 kilobytes in size. Because the file specification specifies the REUSE clause, each member can (but need not) already exist.

The following statement adds a logfile group 5 with member log files on migration target disks 4k_disk_a and 4k_disk_b. After executing this statement, you can switch existing log files on disks with 512-byte block size to logs with 4K block size using the *switch_logfile_clause*.

```
ALTER DATABASE ADD LOGFILE GROUP 5
     ('4k_disk_a:log5.log', '4k_disk_b:log5.log')
     SIZE 100M BLOCKSIZE 4096 REUSE;
```

Specifying a Data File: Example

The following statement creates a tablespace named stocks that has three data files:

```
CREATE TABLESPACE stocks
   DATAFILE 'stock1.dbf' SIZE 10M,
```

```
'stock2.dbf' SIZE 10M,
'stock3.dbf' SIZE 10M;
```

The file specifications for the data files specify files named 'diskc:stock1.dbf', 'diskc:stock2.dbf', and 'diskc:stock3.dbf'.

Adding a Data File: Example

The following statement alters the stocks tablespace and adds a new data file:

```
ALTER TABLESPACE stocks
   ADD DATAFILE 'stock4.dbf' SIZE 10M REUSE;
```

The file specification specifies a data file named 'stock4.dbf'. If the filename does not exist, then Oracle simply ignores the REUSE keyword.

Using a Fully Qualified Oracle ASM Data File Name: Example

When using Oracle ASM, the following syntax shows how to use the *fully_qualified_file_name* clause to bring online a data file in a hypothetical database, testdb:

```
ALTER DATABASE testdb
   DATAFILE '+dgroup_01/testdb/datafile/system.261.1' ONLINE;
```

logging_clause

Purpose

The *logging_clause* lets you specify whether certain DML operations will be logged in the redo log file (LOGGING) or not (NOLOGGING).

You can specify the *logging_clause* in the following statements:

- CREATE TABLE and ALTER TABLE: for logging of the table, a table partition, a LOB segment, or the overflow segment of an index-organized table (see CREATE TABLE and ALTER TABLE).

  > **Note:**
  >
  > Logging specified for a LOB column can differ from logging set at the table level. If you specify LOGGING at the table level and NOLOGGING for a LOB column, then DML changes to the base table row are logged, but DML changes to the LOB data are not logged.

- CREATE INDEX and ALTER INDEX: for logging of the index or an index partition (see CREATE INDEX and ALTER INDEX).
- CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW: for logging of the materialized view, one of its partitions, or a LOB segment (see CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW).
- CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG: for logging of the materialized view log or one of its partitions (see CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG).

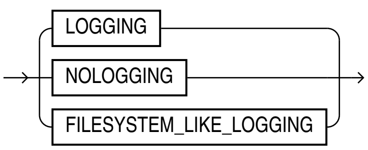
- `CREATE TABLESPACE` and `ALTER TABLESPACE`: to set or modify the default logging characteristics for all objects created in the tablespace (see CREATE TABLESPACE and ALTER TABLESPACE).
- `CREATE PLUGGABLE DATABASE` and `ALTER PLUGGABLE DATABASE`: to set or modify the default logging characteristics for all tablespaces created in the pluggable database (PDB) (see CREATE PLUGGABLE DATABASE and ALTER PLUGGABLE DATABASE).

You can also specify `LOGGING` or `NOLOGGING` for the following operations:

- Rebuilding an index (using `CREATE INDEX` ... `REBUILD`)
- Moving a table (using `ALTER TABLE` ... `MOVE`)

Syntax

***logging_clause*::=**



Semantics

This section describes the semantics of the *`logging_clause`*. For additional information, refer to the SQL statement in which you set or reset logging characteristics for a particular database object.

- If you specify `LOGGING`, then the creation of a database object, as well as subsequent inserts into the object, will be logged in the redo log file.
- If you specify `NOLOGGING`, then the creation of a database object, as well as subsequent conventional inserts, will be logged in the redo log file. Direct-path inserts will not be logged.
  - For a **nonpartitioned object**, the value specified for this clause is the actual physical attribute of the segment associated with the object.
  - For **partitioned objects**, the value specified for this clause is the default physical attribute of the segments associated with all partitions specified in the `CREATE` statement (and in subsequent `ALTER` ... `ADD PARTITION` statements), unless you specify the logging attribute in the `PARTITION` description.
  - For SecureFiles LOBs, the `NOLOGGING` setting is converted internally to `FILESYSTEM_LIKE_LOGGING`.
  - `CACHE NOLOGGING` is not allowed for BasicFiles LOBs.
- The `FILESYSTEM_LIKE_LOGGING` clause is valid only for logging of SecureFiles LOB segments. You cannot specify this setting for BasicFiles LOBs. Specify this setting if you want to log only metadata changes. This setting is similar to the metadata journaling of file systems, which reduces mean time to recovery from failures. The `LOGGING` setting, for SecureFiles LOBs, is similar to the data journaling of file systems. Both the `LOGGING` and `FILESYSTEM_LIKE_LOGGING` settings provide a complete transactional file system by way of SecureFiles.

> **Note:**
>
> For LOB segments, with the `NOLOGGING` and `FILESYSTEM_LIKE_LOGGING` settings it is possible for data to be changed on disk during a backup operation, resulting in an inconsistent backup. To avoid this situation, ensure that changes to LOB segments are saved in the redo log file by setting `LOGGING` for LOB storage. Alternatively, change the database to `FORCE LOGGING` mode so that changes to *all* LOB segments are saved in the redo.

If the object for which you are specifying the logging attributes resides in a database or tablespace in force logging mode, then Oracle Database ignores any `NOLOGGING` setting until the database or tablespace is taken out of force logging mode.

If the database is running in `ARCHIVELOG` mode, then media recovery from a backup made before the `LOGGING` operation re-creates the object. However, media recovery from a backup made before the `NOLOGGING` operation does not re-create the object.

The size of a redo log generated for an operation in `NOLOGGING` mode is significantly smaller than the log generated in `LOGGING` mode.

In `NOLOGGING` mode, data is modified with minimal logging (to mark new extents `INVALID` and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not fully logged. Therefore, if you cannot afford to lose the database object, then you should take a backup after the `NOLOGGING` operation.

`NOLOGGING` is supported in only a subset of the locations that support `LOGGING`. Only the following operations support the `NOLOGGING` mode:

DML:

- Direct-path `INSERT` (serial or parallel) resulting either from an `INSERT` or a `MERGE` statement. `NOLOGGING` is not applicable to any `UPDATE` operations resulting from the `MERGE` statement.
- Direct Loader (SQL*Loader)

DDL:

- `CREATE TABLE` ... `AS SELECT` (In `NOLOGGING` mode, the creation of the table will be logged, but direct-path inserts will not be logged.)
- `CREATE TABLE` ... *`LOB_storage_clause`* ... *`LOB_parameters`* ... `CACHE` | `NOCACHE` | `CACHE READS`
- `ALTER TABLE` ... *`LOB_storage_clause`* ... *`LOB_parameters`* ... `CACHE` | `NOCACHE` | `CACHE READS` (to specify logging of newly created LOB columns)
- `ALTER TABLE` ... *`modify_LOB_storage_clause`* ... *`modify_LOB_parameters`* ... `CACHE` | `NOCACHE` | `CACHE READS` (to change logging of existing LOB columns)
- `ALTER TABLE` ... `MOVE`
- `ALTER TABLE` ... (all partition operations that involve data movement)
  - `ALTER TABLE` ... `ADD PARTITION` (hash partition only)

  - `ALTER TABLE ... MERGE PARTITIONS`
  - `ALTER TABLE ... SPLIT PARTITION`
  - `ALTER TABLE ... MOVE PARTITION`
  - `ALTER TABLE ... MODIFY PARTITION ... ADD SUBPARTITION`
  - `ALTER TABLE ... MODIFY PARTITION ... COALESCE SUBPARTITION`
- `CREATE INDEX`
- `ALTER INDEX ... REBUILD`
- `ALTER INDEX ... REBUILD [SUB]PARTITION`
- `ALTER INDEX ... SPLIT PARTITION`

For **objects other than LOBs**, if you omit this clause, then the logging attribute of the object defaults to the logging attribute of the tablespace in which it resides.

For **LOBs**, if you omit this clause, then:

- If you specify `CACHE`, then `LOGGING` is used (because you cannot have `CACHE NOLOGGING`).
- If you specify `NOCACHE` or `CACHE READS`, then the logging attribute defaults to the logging attribute of the tablespace in which it resides.

`NOLOGGING` does not apply to LOBs that are stored internally (in the table with row data). If you specify `NOLOGGING` for LOBs with values less than 4000 bytes and you have not disabled `STORAGE IN ROW`, then Oracle ignores the `NOLOGGING` specification and treats the LOB data the same as other table data.

parallel_clause

Purpose

The *`parallel_clause`* lets you parallelize the creation of a database object and set the default degree of parallelism for subsequent queries of and DML operations on the object.

You can specify the *`parallel_clause`* in the following statements:

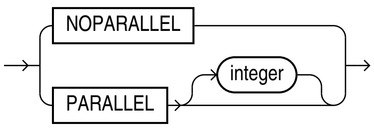
- `CREATE TABLE`: to set parallelism for the table (see CREATE TABLE).
- `ALTER TABLE` (see ALTER TABLE):
  - To change parallelism for the table
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a table partition
- `CREATE CLUSTER` and `ALTER CLUSTER`: to set or alter parallelism for a cluster (see CREATE CLUSTER and ALTER CLUSTER).
- `CREATE INDEX`: to set parallelism for the index (see CREATE INDEX).
- `ALTER INDEX` (see ALTER INDEX):
  - To change parallelism for the index
  - To parallelize the rebuilding of the index or the splitting of an index partition

- CREATE MATERIALIZED VIEW: to set parallelism for the materialized view (see CREATE MATERIALIZED VIEW).
- ALTER MATERIALIZED VIEW (see ALTER MATERIALIZED VIEW):
  - To change parallelism for the materialized view
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a materialized view partition
  - To parallelize the operations of adding or moving materialized view subpartitions
- CREATE MATERIALIZED VIEW LOG: to set parallelism for the materialized view log (see CREATE MATERIALIZED VIEW LOG).
- ALTER MATERIALIZED VIEW LOG (see ALTER MATERIALIZED VIEW LOG):
  - To change parallelism for the materialized view log
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a materialized view log partition
- ALTER DATABASE ... RECOVER: to recover the database (see ALTER DATABASE).
- ALTER DATABASE ... *standby_database_clauses*: to parallelize operations on the standby database (see ALTER DATABASE).

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the DBMS_PARALLEL_EXECUTE package, which provides methods to apply table changes in chunks of rows. Changes to each chunk are independently committed when there are no errors.

Syntax

***parallel_clause*::=**



Semantics

This section describes the semantics of the *parallel_clause*. For additional information, refer to the SQL statement in which you set or reset parallelism for a particular database object or operation.

> **Note:**
>
> The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. The superseded syntax is still supported for backward compatibility, but may result in slightly different behavior from that documented.

The database interprets the *parallel_clause* based on the setting of the PARALLEL_DEGREE_POLICY initialization parameter. When that parameter is set to AUTO, the *parallel_clause* is ignored entirely, and the optimizer determines the best degree of parallelism for all statements. When PARALLEL_DEGREE_POLICY is set to either MANUAL or LIMITED, the *parallel_clause* is interpreted as follows:

NOPARALLEL

Specify NOPARALLEL for serial execution. This is the default.

PARALLEL

Specify PARALLEL for parallel execution.

- If PARALLEL_DEGREE_POLICY is set to MANUAL, then the optimizer calculates a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
- If PARALLEL_DEGREE_POLICY is set to LIMITED, then the optimizer determines the best degree of parallelism.

PARALLEL *integer*

Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers.

Notes on the *parallel_clause*

The following notes apply to the *parallel_clause*:

- Parallelism is disabled for DML operations on tables on which you have defined a trigger or referential integrity constraint.
- Parallelism is not supported for UPDATE or DELETE operations on index-organized tables.
- When you specify the *parallel_clause* during creation of a table, if the table contains any columns of LOB or user-defined object type, then subsequent INSERT, UPDATE, DELETE or MERGE operations that modify the LOB or object type column are executed serially without notification. Subsequent queries, however, will be executed in parallel.
- A parallel hint overrides the effect of the *parallel_clause*.
- DML statements and CREATE TABLE ... AS SELECT statements that reference remote objects can run in parallel. However, the remote object must really be on a remote database. The reference cannot loop back to an object on the local database, for example, by way of a synonym on the remote database pointing back to an object on the local database.
- DML operations on tables with LOB columns can be parallelized. However, intrapartition parallelism is not supported.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on parallelized operations, and "Creating a Table: Parallelism Examples"

physical_attributes_clause

Purpose

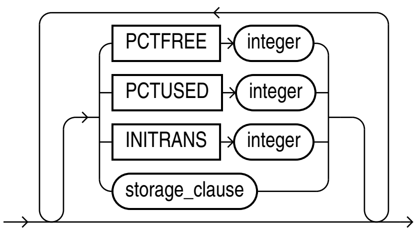
The `physical_attributes_clause` lets you specify the value of the `PCTFREE`, `PCTUSED`, and `INITRANS` parameters and the storage characteristics of a table, cluster, index, or materialized view.

You can specify the `physical_attributes_clause` in the following statements:

- `CREATE CLUSTER` and `ALTER CLUSTER`: to set or change the physical attributes of the cluster and all tables in the cluster (see CREATE CLUSTER and ALTER CLUSTER).
- `CREATE TABLE`: to set the physical attributes of the table, a table partition, the `OIDINDEX` of an object table, or the overflow segment of an index-organized table (see CREATE TABLE).
- `ALTER TABLE`: to change the physical attributes of the table, the default physical attributes of future table partitions, or the physical attributes of existing table partitions (see ALTER TABLE). The following restrictions apply:
  - You cannot specify physical attributes for a temporary table.
  - You cannot specify physical attributes for a clustered table. Tables in a cluster inherit the physical attributes of the cluster.
- `CREATE INDEX`: to set the physical attributes of an index or index partition (see CREATE INDEX).
- `ALTER INDEX`: to change the physical attributes of the index, the default physical attributes of future index partitions, or the physical attributes of existing index partitions (see ALTER INDEX).
- `CREATE MATERIALIZED VIEW`: to set the physical attributes of the materialized view, one of its partitions, or the index Oracle Database generates to maintain the materialized view (see CREATE MATERIALIZED VIEW).
- `ALTER MATERIALIZED VIEW`: to change the physical attributes of the materialized view, the default physical attributes of future partitions, the physical attributes of an existing partition, or the index Oracle creates to maintain the materialized view (see ALTER MATERIALIZED VIEW).
- `CREATE MATERIALIZED VIEW LOG` and `ALTER MATERIALIZED VIEW LOG`: to set or change the physical attributes of the materialized view log (see CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG).

Syntax

***physical_attributes_clause*::=**



(*storage_clause*::=)

Semantics

This section describes the parameters of the *physical_attributes_clause*. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

PCTFREE *integer*

Specify a whole number representing the percentage of space in each data block of the database object reserved for future updates to rows of the object. The value of PCTFREE must be a value from 0 to 99. A value of 0 means that the entire block can be filled by inserts of new rows. The default value is 10. This value reserves 10% of each block for updates to existing rows and allows inserts of new rows to fill a maximum of 90% of each block.

PCTFREE has the same function in the statements that create and alter tables, partitions, clusters, indexes, materialized views, materialized view logs, and zone maps. The combination of PCTFREE and PCTUSED determines whether new rows will be inserted into existing data blocks or into new blocks. See "How PCTFREE and PCTUSED Work Together".

Restriction on the PCTFREE Clause

When altering an index, you can specify this parameter only in the *modify_index_default_attrs* clause and the *split_index_partition* clause.

PCTUSED *integer*

Specify a whole number representing the minimum percentage of used space that Oracle maintains for each data block of the database object. PCTUSED is specified as a positive integer from 0 to 99 and defaults to 40.

PCTUSED has the same function in the statements that create and alter tables, partitions, clusters, materialized views, materialized view logs, and zone maps.

PCTUSED is not a valid table storage characteristic for an index-organized table.

The sum of PCTFREE and PCTUSED must be equal to or less than 100. You can use PCTFREE and PCTUSED together to utilize space within a database object more efficiently. See "How PCTFREE and PCTUSED Work Together".

Restrictions on the PCTUSED Clause

The PCTUSED parameter is subject to the following restrictions:

- You cannot specify this parameter for an index or for the index segment of an index-organized table.
- This parameter is not useful and is ignored for objects with automatic segment-space management.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for information on the performance effects of different values of PCTUSED and PCTFREE and CREATE TABLESPACE
> *segment_management_clause* for information on automatic segment-space management

How PCTFREE and PCTUSED Work Together

In a newly allocated data block, the space available for inserts is the block size minus the sum of the block overhead and free space (PCTFREE). Updates to existing data can use any available space in the block. Therefore, updates can reduce the available space of a block to less than PCTFREE.

After a data block is filled to the limit determined by PCTFREE, Oracle Database considers the block unavailable for the insertion of new rows until the percentage of that block falls beneath the parameter PCTUSED. Until this value is achieved, Oracle Database uses the free space of the data block only for updates to rows already contained in the data block. A block becomes a candidate for row insertion when its used space falls below PCTUSED.

> **See Also:**
>
> FREELISTS for information on how PCTUSED and PCTFREE work with freelist segment space management

INITRANS *integer*

Specify the initial number of concurrent transaction entries allocated within each data block allocated to the database object. This value can range from 1 to 255 and defaults to 1, with the following exceptions:

- The default INITRANS value for a cluster is 2 or the default INITRANS value of the tablespace in which the cluster resides, whichever is greater.
- The default value for an index is 2.

In general, you should not change the INITRANS value from its default.

Each transaction that updates a block requires a transaction entry in the block. This parameter ensures that a minimum number of concurrent transactions can update the block and helps avoid the overhead of dynamically allocating a transaction entry.

The INITRANS parameter serves the same purpose in the statements that create and alter tables, partitions, clusters, indexes, materialized views, and materialized view logs.

MAXTRANS Parameter

In earlier releases, the MAXTRANS parameter determined the maximum number of concurrent update transactions allowed for each data block in the segment. This parameter has been deprecated. Oracle now automatically allows up to 255 concurrent update transactions for any data block, depending on the available space in the block.

Existing objects for which a value of MAXTRANS has already been set retain that setting. However, if you attempt to change the value for MAXTRANS, Oracle ignores the new specification and substitutes the value 255 without returning an error.

storage_clause

The *storage_clause* lets you specify storage characteristics for the table, object table OIDINDEX, partition, LOB data segment, or index-organized table overflow data segment. This clause has performance ramifications for large tables. Storage should be allocated to minimize dynamic allocation of additional space. Refer to the *storage_clause* for more information.

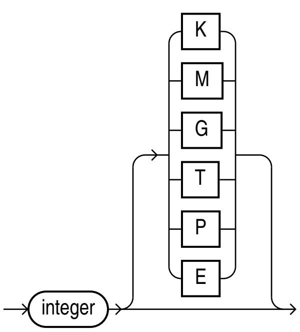
size_clause

Purpose

The *size_clause* lets you specify a number of bytes, kilobytes (K), megabytes (M), gigabytes (G), terabytes (T), petabytes (P), or exabytes (E) in any statement that lets you establish amounts of disk or memory space.

Syntax

***size_clause*::=**



Semantics

Use the *size_clause* to specify a number or multiple of bytes. If you do not specify any of the multiple abbreviations, then the *integer* is interpreted as bytes.

> **Note:**
>
> Not all multiples of bytes are appropriate in all cases, and context-sensitive limitations may apply. In the latter case, Oracle issues an error message.

storage_clause

Purpose

The *storage_clause* lets you specify how Oracle Database should store a permanent database object. Storage parameters for temporary segments always use the default storage parameters for the associated tablespace. Storage parameters affect both how long it takes to access data stored in the database and how efficiently space in the database is used.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for a discussion of the effects of the storage parameters

When you create a cluster, index, materialized view, materialized view log, rollback segment, table, LOB, varray, nested table, or partition, you can specify values for the storage parameters for the segments allocated to these objects. If you omit any storage parameter, then Oracle uses the value of that parameter specified for the tablespace in which the object resides. If no value was specified for the tablespace, then the database uses default values.

> **Note:**
>
> The specification of storage parameters for objects in locally managed tablespaces is supported for backward compatibility. If you are using locally managed tablespaces, then you can omit these storage parameter when creating objects in those tablespaces.

When you alter a cluster, index, materialized view, materialized view log, rollback segment, table, varray, nested table, or partition, you can change the values of storage parameters. The new values affect only future extent allocations.

The *storage_clause* is part of the *physical_attributes_clause*, so you can specify this clause in any of the statements where you can specify the physical attributes clause (see *physical_attributes_clause*). In addition, you can specify the *storage_clause* in the following statements:

- CREATE CLUSTER and ALTER CLUSTER: to set or change the storage characteristics of the **cluster and all tables in the cluster** (see CREATE CLUSTER and ALTER CLUSTER).
- CREATE INDEX and ALTER INDEX: to set or change the storage characteristics of an index segment created for a **table index or index partition** or an **index segment created for an index used to enforce a primary key or unique constraint** (see CREATE INDEX and ALTER INDEX).
- The ENABLE ... USING INDEX clause of CREATE TABLE or ALTER TABLE: to set or change the storage characteristics of an **index created by the system to enforce a primary key or unique constraint**.
- CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW: to set or change the storage characteristics of a **materialized view, one of its partitions, or the index Oracle**

generates to maintain the materialized view (see CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW).

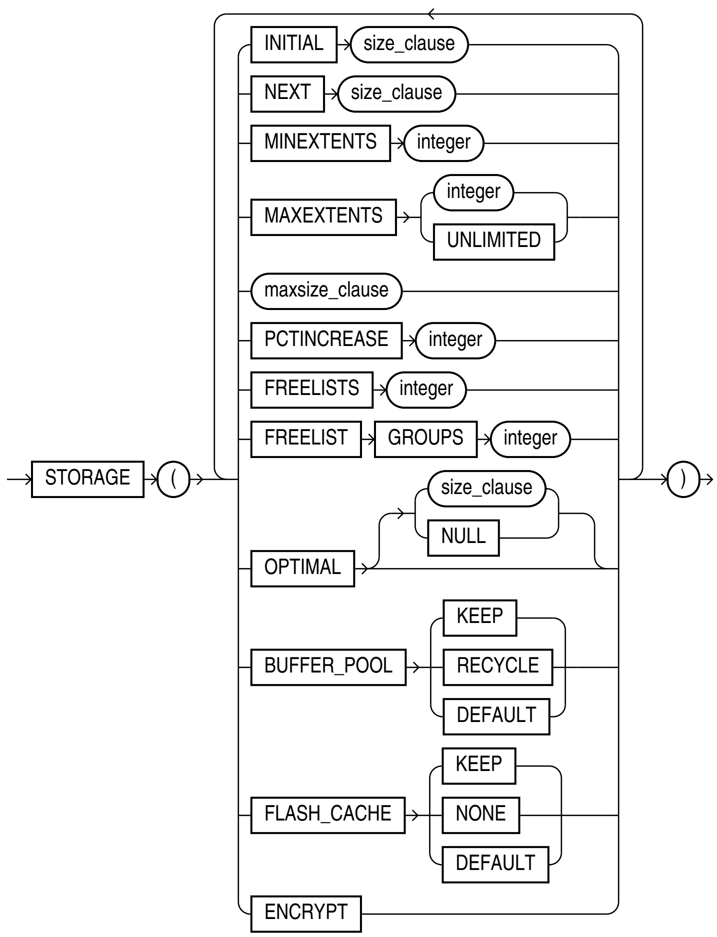
- `CREATE MATERIALIZED VIEW LOG` and `ALTER MATERIALIZED VIEW LOG`: to set or change the storage characteristics of **the materialized view log** (see CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG).
- `CREATE ROLLBACK SEGMENT` and `ALTER ROLLBACK SEGMENT`: to set or change the storage characteristics of a **rollback segment** (see CREATE ROLLBACK SEGMENT and ALTER ROLLBACK SEGMENT).
- `CREATE TABLE` and `ALTER TABLE`: to set the storage characteristics of a **LOB or varray data segment** of the nonclustered table or one of its partitions or subpartitions, or the **storage table of a nested table** (see CREATE TABLE and ALTER TABLE).
- `CREATE TABLESPACE` and `ALTER TABLESPACE`: to set or change the default storage characteristics for **objects created in the tablespace** (see CREATE TABLESPACE and ALTER TABLESPACE). Changes to tablespace storage parameters affect only new objects created in the tablespace or new extents allocated for a segment.
- *`constraint`*: to specify storage for the **index (and its partitions, if it is a partitioned index) used to enforce the constraint** (see *constraint*).

Prerequisites

To change the value of a `STORAGE` parameter, you must have the privileges necessary to use the appropriate `CREATE` or `ALTER` statement.

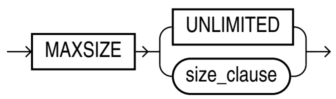
Syntax

***storage_clause*::=**



(*size_clause*::=)

***maxsize_clause*::=**



(*size_clause*::=)

Semantics

This section describes the parameters of the `storage_clause`. For additional information, refer to the SQL statement in which you set or reset these storage parameters for a particular database object.

Note:

The *storage_clause* is interpreted differently for locally managed tablespaces. For locally managed tablespaces, Oracle Database uses INITIAL, NEXT, PCTINCREASE, and MINEXTENTS to compute how many extents are allocated when the object is first created. After object creation, these parameters are ignored. For more information, see CREATE TABLESPACE .

See Also:

"Specifying Table Storage Attributes: Example"

INITIAL

Specify the size of the first extent of the object. Oracle allocates space for this extent when you create the schema object. Refer to *size_clause* for information on that clause.

In locally managed tablespaces, Oracle uses the value of INITIAL, in conjunction with the type of local management—AUTOALLOCATE or UNIFORM—and the values of MINEXTENTS, NEXT and PCTINCREASE, to determine the initial size of the segment.

- With AUTOALLOCATE extent management, Oracle uses the INITIAL setting to optimize the number of extents allocated. Extents of 64K, 1M, 8M, and 64M can be allocated. During segment creation, the system chooses the greatest of these four sizes that is equal to or smaller than INITIAL, and allocates as many extents of that size as are needed to reach the INITIAL setting. For example, if you set INITIAL to 4M, then the database creates four 1M extents.
- For UNIFORM extent management, the number of extents is determined from initial segment size and the uniform extent size specified at tablespace creation time. For example, in a uniform locally managed tablespace with 1M extents, if you specify an INITIAL value of 5M, then Oracle creates five 1M extents.

  Consider this comparison: With AUTOALLOCATE, if you set INITAL to 72K, then the initial segment size will be 128K (greater than INITIAL). The database cannot allocate an extent smaller than 64K, so it must allocate two 64K extents. If you set INITIAL to 72K with a UNIFORM extent size of 24K, then the database will allocate three 24K extents to equal 72K.

In dictionary managed tablespaces, the default initial extent size is 5 blocks, and all subsequent extents are rounded to 5 blocks. If MINIMUM EXTENT was specified at tablespace creation time, then the extent sizes are rounded to the value of MINIMUM EXTENT.

Restriction on INITIAL

You cannot specify INITIAL in an ALTER statement.

NEXT

Specify in bytes the size of the next extent to be allocated to the object. Refer to *size_clause* for information on that clause.

In locally managed tablespaces, any user-supplied value for `NEXT` is ignored and the size of `NEXT` is determined by Oracle if the tablespace is set for autoallocate extent management. In `UNIFORM` tablespaces, the size of `NEXT` is the uniform extent size specified at tablespace creation time.

In dictionary-managed tablespaces, the default value is the size of 5 data blocks. The minimum value is the size of 1 data block. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation.

> **See Also:**
>
> *Oracle Database Concepts* for information on how Oracle minimizes fragmentation

PCTINCREASE

In locally managed tablespaces, Oracle Database uses the value of `PCTINCREASE` during segment creation to determine the initial segment size and ignores this parameter during subsequent space allocation.

In dictionary-managed tablespaces, specify the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, meaning that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system. Oracle rounds the calculated size of each new extent to the nearest multiple of the data block size. If you change the value of the `PCTINCREASE` parameter by specifying it in an `ALTER` statement, then Oracle calculates the size of the next extent using this new value and the size of the most recently allocated extent.

Restriction on PCTINCREASE

You cannot specify `PCTINCREASE` for rollback segments. Rollback segments always have a `PCTINCREASE` value of 0.

MINEXTENTS

In locally managed tablespaces, Oracle Database uses the value of `MINEXTENTS` in conjunction with `PCTINCREASE`, `INITIAL` and `NEXT` to determine the initial segment size.

In dictionary-managed tablespaces, specify the total number of extents to allocate when the object is created. The default and minimum value is 1, meaning that Oracle allocates only the initial extent, except for rollback segments, for which the default and minimum value is 2. The maximum value depends on your operating system.

- In a locally managed tablespace, `MINEXTENTS` is used to compute the initial amount of space allocated, which is equal to `INITIAL` * `MINEXTENTS`. Thereafter this value is set to 1, which is reflected in the `DBA_SEGMENTS` view.
- In a dictionary-managed tablespace, `MINEXTENTS` is simply the minimum number of extents that must be allocated to the segment.

If the `MINEXTENTS` value is greater than 1, then Oracle calculates the size of subsequent extents based on the values of the `INITIAL`, `NEXT`, and `PCTINCREASE` storage parameters.

When changing the value of MINEXTENTS by specifying it in an ALTER statement, you can reduce the value from its current value, but you cannot increase it. Resetting MINEXTENTS to a smaller value might be useful, for example, before a TRUNCATE ... DROP STORAGE statement, if you want to ensure that the segment will maintain a minimum number of extents after the TRUNCATE operation.

Restrictions on MINEXTENTS

The MINEXTENTS storage parameter is subject to the following restrictions:

- MINEXTENTS is not applicable at the tablespace level.
- You cannot change the value of MINEXTENTS in an ALTER statement or for an object that resides in a locally managed tablespace.

MAXEXTENTS

This storage parameter is valid only for objects in dictionary-managed tablespaces. Specify the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1 except for rollback segments, which always have a minimum of 2. The default value depends on your data block size.

Restriction on MAXEXTENTS

MAXEXTENTS is ignored for objects residing in a locally managed tablespace, unless the value of ALLOCATION_TYPE is USER for the tablespace in the DBA_TABLESPACES data dictionary view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the DBA_TABLESPACES data dictionary view

UNLIMITED

Specify UNLIMITED if you want extents to be allocated automatically as needed. Oracle recommends this setting as a way to minimize fragmentation.

Do not use this clause for rollback segments. Doing so allows the possibility that long-running rogue DML transactions will continue to create new extents until a disk is full.

> **Note:**
>
> A rollback segment that you create without specifying the *storage_clause* has the same storage parameters as the tablespace in which the rollback segment is created. Thus, if you create a tablespace with MAXEXTENTS UNLIMITED, then the rollback segment will have this same default.

MAXSIZE

The MAXSIZE clause lets you specify the maximum size of the storage element. For LOB storage, MAXSIZE has the following effects

- If you specify RETENTION MAX in *LOB_parameters*, then the LOB segment increases to the specified size before any space can be reclaimed from undo space.
- If you specify RETENTION AUTO, MIN, or NONE in *LOB_parameters*, then the specified size is a hard limit on the LOB segment size and has no bearing on undo retention.

UNLIMITED

Use the UNLIMITED clause if you do not want to limit the disk space of the storage element. This clause is not compatible with a specification of RETENTION MAX in *LOB_parameters*. If you specify both, then the database uses RETENTION AUTO and MAXSIZE UNLIMITED.

FREELISTS

In tablespaces with manual segment-space management, Oracle Database uses the FREELISTS storage parameter to improve performance of space management in OLTP systems by increasing the number of insert points in the segment. In tablespaces with automatic segment-space management, this parameter is ignored, because the database adapts to varying workload.

In tablespaces with manual segment-space management, for objects other than tablespaces and rollback segments, specify the number of free lists for each of the free list groups for the table, partition, cluster, or index. The default and minimum value for this parameter is 1, meaning that each free list group contains one free list. The maximum value of this parameter depends on the data block size. If you specify a FREELISTS value that is too large, then Oracle returns an error indicating the maximum value.

This clause is not valid or useful if you have specified the SECUREFILE parameter of *LOB_parameters*. If you specify both the SECUREFILE parameter and FREELISTS, then the database silently ignores the FREELISTS specification.

Restriction on FREELISTS

You can specify FREELISTS in the *storage_clause* of any statement except when creating or altering a tablespace or rollback segment.

FREELIST GROUPS

In tablespaces with manual segment-space management, Oracle Database uses the value of this storage parameter to statically partition the segment free space in an Oracle Real Application Clusters environment. This partitioning improves the performance of space allocation and deallocation by avoiding inter instance transfer of segment metadata. In tablespaces with automatic segment-space management, this parameter is ignored, because Oracle dynamically adapts to inter instance workload.

In tablespaces with manual segment-space management, specify the number of groups of free lists for the database object you are creating. The default and minimum value for this parameter is 1. Oracle uses the instance number of Oracle Real Application Clusters (Oracle RAC) instances to map each instance to one free list group.

Each free list group uses one database block. Therefore:

- If you do not specify a large enough value for INITIAL to cover the minimum value plus one data block for each free list group, then Oracle increases the value of INITIAL the necessary amount.
- If you are creating an object in a uniform locally managed tablespace, and the extent size is not large enough to accommodate the number of freelist groups, then the create operation will fail.

This clause is not valid or useful if you have specified the SECUREFILE parameter of *LOB_parameters*. If you specify both the SECUREFILE parameter and FREELIST GROUPS, then the database silently ignores the FREELIST GROUPS specification.

Restriction on FREELIST GROUPS

You can specify the FREELIST GROUPS parameter only in CREATE TABLE, CREATE CLUSTER, CREATE MATERIALIZED VIEW, CREATE MATERIALIZED VIEW LOG, and CREATE INDEX statements.

OPTIMAL

The OPTIMAL keyword is relevant only to rollback segments. It specifies an optimal size in bytes for a rollback segment. Refer to *size_clause* for information on that clause.

Oracle tries to maintain this size for the rollback segment by dynamically deallocating extents when their data is no longer needed for active transactions. Oracle deallocates as many extents as possible without reducing the total size of the rollback segment below the OPTIMAL value.

The value of OPTIMAL cannot be less than the space initially allocated by the MINEXTENTS, INITIAL, NEXT, and PCTINCREASE parameters. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size.

NULL

Specify NULL for no optimal size for the rollback segment, meaning that Oracle never deallocates the extents of the rollback segment. This is the default behavior.

BUFFER_POOL

The BUFFER_POOL clause lets you specify a default buffer pool or cache for a schema object. All blocks for the object are stored in the specified cache.

- If you define a buffer pool for a partitioned table or index, then the partitions inherit the buffer pool from the table or index definition unless overridden by a partition-level definition.
- For an index-organized table, you can specify a buffer pool separately for the index segment and the overflow segment.

Restrictions on the BUFFER_POOL Parameter

BUFFER_POOL is subject to the following restrictions:

- You cannot specify this clause for a cluster table. However, you can specify it for a cluster.
- You cannot specify this clause for a tablespace or a rollback segment.

KEEP

Specify KEEP to put blocks from the segment into the KEEP buffer pool. Maintaining an appropriately sized KEEP buffer pool lets Oracle retain the schema object in memory to avoid I/O operations. KEEP takes precedence over any NOCACHE clause you specify for a table, cluster, materialized view, or materialized view log.

RECYCLE

Specify RECYCLE to put blocks from the segment into the RECYCLE pool. An appropriately sized RECYCLE pool reduces the number of objects whose default pool is the RECYCLE pool from taking up unnecessary cache space.

DEFAULT

Specify DEFAULT to indicate the default buffer pool. This is the default for objects not assigned to KEEP or RECYCLE.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for more information about using multiple buffer pools

FLASH_CACHE

The FLASH_CACHE clause lets you override the automatic buffer cache policy and specify how specific schema objects are cached in flash memory. To use this clause, Database Smart Flash Cache (flash cache) must be configured on your system. The flash cache is an extension of the database buffer cache that is stored on a flash disk, a storage device that uses flash memory. Because flash memory is faster than magnetic disks, the database can improve performance by caching buffers in the flash cache instead of reading from magnetic disk.

KEEP

Specify KEEP if you want the schema object buffers to remain cached in the flash cache as long as the flash cache is large enough.

NONE

Specify NONE to ensure that the schema object buffers are never cached in the flash cache. This allows you to reserve the flash cache space for more frequently accessed objects.

DEFAULT

Specify DEFAULT if you want the schema object buffers to be written to the flash cache when they are aged out of main memory, and then be aged out of the flash cache with the standard buffer cache replacement algorithm. This is the default if flash cache is configured and you do not specify KEEP or NONE.

> **Note:**
>
> Database Smart Flash Cache is available only in Solaris and Oracle Linux.

See Also:

- *Oracle Database Concepts* for more information about Database Smart Flash Cache
- *Oracle Database Administrator's Guide* to learn how to configure Database Smart Flash Cache

ENCRYPT

This clause is valid only when you are creating a tablespace. Specify `ENCRYPT` to encrypt the entire tablespace. You must also specify the `ENCRYPTION` clause in the `CREATE TABLESPACE` statement.

Note:

The `ENCRYPT` clause is supported for backward compatibility. However, beginning with Oracle Database 12*c* Release 2 (12.2), you can instead specify `ENCRYPT` in the *`tablespace_encryption_clause`*. Refer to the *tablespace_encryption_clause* of `CREATE TABLESPACE` for more information.

Example

Live SQL:

View and run a related example on Oracle Live SQL at *Specifying Table Storage Attributes*.

Specifying Table Storage Attributes: Example

The following statement creates a table and provides storage parameter values:

```
CREATE TABLE divisions
    (div_no     NUMBER(2),
     div_name   VARCHAR2(14),
     location   VARCHAR2(13) )
     STORAGE  ( INITIAL 8M MAXSIZE 1G );
```

Oracle allocates space for the table based on the `STORAGE` parameter values as follows:

- The `INITIAL` value is 8M, so the size of the first extent is 8 megabytes.
- The `MAXSIZE` value is 1G, so the maximum size of the storage element is 1 gigabyte.

9

SQL Queries and Subqueries

This chapter describes SQL queries and subqueries.

This chapter contains these sections:

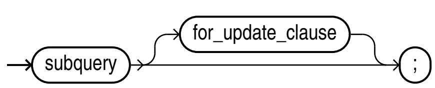
- About Queries and Subqueries
- Creating Simple Queries
- Hierarchical Queries
- The UNION [ALL], INTERSECT, MINUS Operators
- Sorting Query Results
- Joins
- Using Subqueries
- Unnesting of Nested Subqueries
- Selecting from the DUAL Table
- Distributed Queries

About Queries and Subqueries

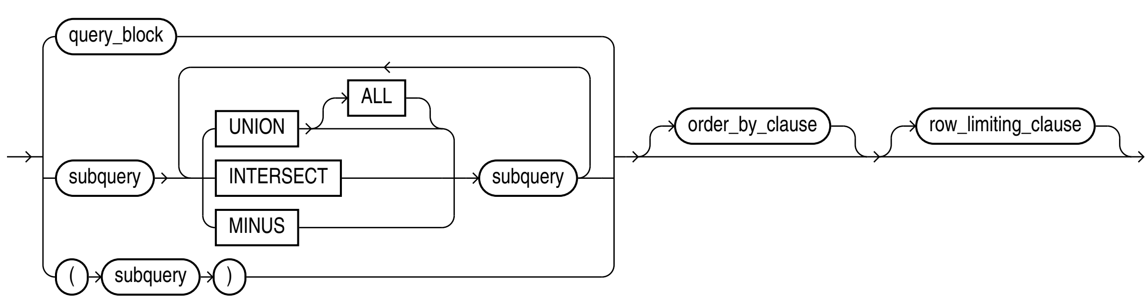
A **query** is an operation that retrieves data from one or more tables or views. In this reference, a top-level `SELECT` statement is called a **query**, and a query nested within another SQL statement is called a **subquery**.

This section describes some types of queries and subqueries and how to use them. The top level of the syntax is shown in this chapter. Refer to SELECT for the full syntax of all the clauses and the semantics of this statement.

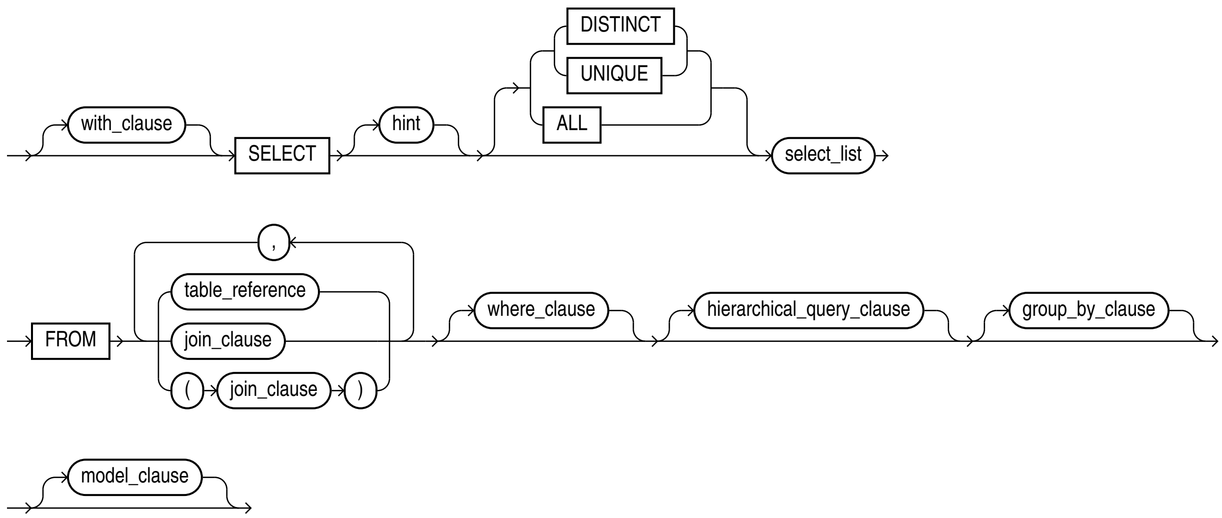
select::=



subquery::=



***query_block*::=**



Creating Simple Queries

The list of expressions that appears after the `SELECT` keyword and before the `FROM` clause is called the **select list**. Within the select list, you specify one or more columns in the set of rows you want Oracle Database to return from one or more tables, views, or materialized views. The number of columns, as well as their data type and length, are determined by the elements of the select list.

If two or more tables have some column names in common, then you must qualify column names with names of tables. Otherwise, fully qualified column names are optional. However, it is always a good idea to qualify table and column references explicitly. Oracle often does less work with fully qualified table and column names.

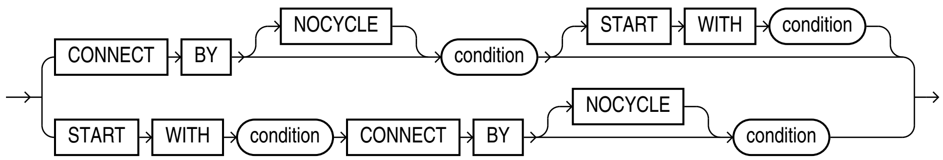
You can use a column alias, *`c_alias`*, to label the immediately preceding expression in the select list so that the column is displayed with a new heading. The alias effectively renames the select list item for the duration of the query. The alias can be used in the `ORDER BY` clause, but not other clauses in the query.

You can use comments in a `SELECT` statement to pass instructions, or **hints**, to the Oracle Database optimizer. The optimizer uses hints to choose an execution plan for the statement. Refer to "Hints " for more information on hints.

Hierarchical Queries

If a table contains hierarchical data, then you can select rows in a hierarchical order using the hierarchical query clause:

***hierarchical_query_clause*::=**



`condition` can be any condition as described in Conditions.

`START WITH` specifies the root row(s) of the hierarchy.

`CONNECT BY` specifies the relationship between parent rows and child rows of the hierarchy.

- The `NOCYCLE` parameter instructs Oracle Database to return rows from a query even if a `CONNECT BY` loop exists in the data. Use this parameter along with the `CONNECT_BY_ISCYCLE` pseudocolumn to see which rows contain the loop. Refer to CONNECT_BY_ISCYCLE Pseudocolumn for more information.
- In a hierarchical query, one expression in *`condition`* must be qualified with the `PRIOR` operator to refer to the parent row. For example,

  ```
  ... PRIOR expr = expr
  or
  ... expr = PRIOR expr
  ```

  If the `CONNECT BY` *`condition`* is compound, then only one condition requires the `PRIOR` operator, although you can have multiple `PRIOR` conditions. For example:

  ```
  CONNECT BY last_name != 'King' AND PRIOR employee_id = manager_id ...
  CONNECT BY PRIOR employee_id = manager_id and
             PRIOR account_mgr_id = customer_id ...
  ```

  `PRIOR` is a unary operator and has the same precedence as the unary + and - arithmetic operators. It evaluates the immediately following expression for the parent row of the current row in a hierarchical query.

  `PRIOR` is most commonly used when comparing column values with the equality operator. (The `PRIOR` keyword can be on either side of the operator.) `PRIOR` causes Oracle to use the value of the parent row in the column. Operators other than the equal sign (=) are theoretically possible in `CONNECT BY` clauses. However, the conditions created by these other operators can result in an infinite loop through the possible combinations. In this case Oracle detects the loop at run time and returns an error.

Both the `CONNECT BY` condition and the `PRIOR` expression can take the form of an uncorrelated subquery. However, `CURRVAL` and `NEXTVAL` are not valid `PRIOR` expressions, so the `PRIOR` expression cannot refer to a sequence.

You can further refine a hierarchical query by using the `CONNECT_BY_ROOT` operator to qualify a column in the select list. This operator extends the functionality of the `CONNECT BY` [`PRIOR`] condition of hierarchical queries by returning not only the immediate parent row but all ancestor rows in the hierarchy.

> **See Also:**
>
> CONNECT_BY_ROOT for more information about this operator and "Hierarchical Query Examples"

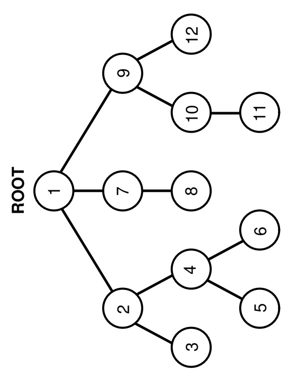
Oracle processes hierarchical queries as follows:

- A join, if present, is evaluated first, whether the join is specified in the `FROM` clause or with `WHERE` clause predicates.
- The `CONNECT BY` condition is evaluated.
- Any remaining `WHERE` clause predicates are evaluated.

Oracle then uses the information from these evaluations to form the hierarchy using the following steps:

1. Oracle selects the root row(s) of the hierarchy—those rows that satisfy the `START WITH` condition.
2. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the `CONNECT BY` condition with respect to one of the root rows.
3. Oracle selects successive generations of child rows. Oracle first selects the children of the rows returned in step 2, and then the children of those children, and so on. Oracle always selects children by evaluating the `CONNECT BY` condition with respect to a current parent row.
4. If the query contains a `WHERE` clause without a join, then Oracle eliminates all rows from the hierarchy that do not satisfy the condition of the `WHERE` clause. Oracle evaluates this condition for each row individually, rather than removing all the children of a row that does not satisfy the condition.
5. Oracle returns the rows in the order shown in Figure 9-1. In the diagram, children appear below their parents. For an explanation of hierarchical trees, see Figure 3-1.

Figure 9-1 Hierarchical Queries



To find the children of a parent row, Oracle evaluates the `PRIOR` expression of the `CONNECT BY` condition for the parent row and the other expression for each row in the table. Rows for which the condition is true are the children of the parent. The `CONNECT`

BY condition can contain other conditions to further filter the rows selected by the query.

If the CONNECT BY condition results in a loop in the hierarchy, then Oracle returns an error. A loop occurs if one row is both the parent (or grandparent or direct ancestor) and a child (or a grandchild or a direct descendent) of another row.

> **Note:**
>
> In a hierarchical query, do not specify either ORDER BY or GROUP BY, as they will override the hierarchical order of the CONNECT BY results. If you want to order rows of siblings of the same parent, then use the ORDER SIBLINGS BY clause. See *order_by_clause*.

Hierarchical Query Examples

CONNECT BY Example

The following hierarchical query uses the CONNECT BY clause to define the relationship between employees and managers:

```
SELECT employee_id, last_name, manager_id
   FROM employees
   CONNECT BY PRIOR employee_id = manager_id;

EMPLOYEE_ID LAST_NAME                 MANAGER_ID
----------- ------------------------- ----------
        101 Kochhar                          100
        108 Greenberg                        101
        109 Faviet                           108
        110 Chen                             108
        111 Sciarra                          108
        112 Urman                            108
        113 Popp                             108
        200 Whalen                           101
        203 Mavris                           101
        204 Baer                             101
. . .
```

LEVEL Example

The next example is similar to the preceding example, but uses the LEVEL pseudocolumn to show parent and child rows:

```
SELECT employee_id, last_name, manager_id, LEVEL
   FROM employees
   CONNECT BY PRIOR employee_id = manager_id;

EMPLOYEE_ID LAST_NAME                 MANAGER_ID      LEVEL
----------- ------------------------- ---------- ----------
        101 Kochhar                          100          1
        108 Greenberg                        101          2
        109 Faviet                           108          3
        110 Chen                             108          3
        111 Sciarra                          108          3
        112 Urman                            108          3
        113 Popp                             108          3
```

```
        200 Whalen                          101          2
        203 Mavris                          101          2
        204 Baer                            101          2
        205 Higgins                         101          2
        206 Gietz                           205          3
        102 De Haan                         100          1
...
```

START WITH Examples

The next example adds a `START WITH` clause to specify a root row for the hierarchy and an `ORDER BY` clause using the `SIBLINGS` keyword to preserve ordering within the hierarchy:

```
SELECT last_name, employee_id, manager_id, LEVEL
      FROM employees
      START WITH employee_id = 100
      CONNECT BY PRIOR employee_id = manager_id
      ORDER SIBLINGS BY last_name;

LAST_NAME                 EMPLOYEE_ID MANAGER_ID      LEVEL
------------------------- ----------- ---------- ----------
King                              100                     1
Cambrault                         148        100          2
Bates                             172        148          3
Bloom                             169        148          3
Fox                               170        148          3
Kumar                             173        148          3
Ozer                              168        148          3
Smith                             171        148          3
De Haan                           102        100          2
Hunold                            103        102          3
Austin                            105        103          4
Ernst                             104        103          4
Lorentz                           107        103          4
Pataballa                         106        103          4
Errazuriz                         147        100          2
Ande                              166        147          3
Banda                             167        147          3
...
```

In the `hr.employees` table, the employee Steven King is the head of the company and has no manager. Among his employees is John Russell, who is the manager of department 80. If you update the `employees` table to set Russell as King's manager, you create a loop in the data:

```
UPDATE employees SET manager_id = 145
   WHERE employee_id = 100;

SELECT last_name "Employee",
   LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   WHERE level <= 3 AND department_id = 80
   START WITH last_name = 'King'
   CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 4;

ERROR:
ORA-01436: CONNECT BY loop in user data
```

The `NOCYCLE` parameter in the `CONNECT BY` condition causes Oracle to return the rows in spite of the loop. The `CONNECT_BY_ISCYCLE` pseudocolumn shows you which rows contain the cycle:

```
SELECT last_name "Employee", CONNECT_BY_ISCYCLE "Cycle",
   LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   WHERE level <= 3 AND department_id = 80
   START WITH last_name = 'King'
   CONNECT BY NOCYCLE PRIOR employee_id = manager_id AND LEVEL <= 4
   ORDER BY "Employee", "Cycle", LEVEL, "Path";

Employee                       Cycle      LEVEL Path
------------------------- ---------- ---------- -------------------------
Abel                               0          3 /King/Zlotkey/Abel
Ande                               0          3 /King/Errazuriz/Ande
Banda                              0          3 /King/Errazuriz/Banda
Bates                              0          3 /King/Cambrault/Bates
Bernstein                          0          3 /King/Russell/Bernstein
Bloom                              0          3 /King/Cambrault/Bloom
Cambrault                          0          2 /King/Cambrault
Cambrault                          0          3 /King/Russell/Cambrault
Doran                              0          3 /King/Partners/Doran
Errazuriz                          0          2 /King/Errazuriz
Fox                                0          3 /King/Cambrault/Fox
...
```

CONNECT_BY_ISLEAF Example

The following statement shows how you can use a hierarchical query to turn the values in a column into a comma-delimited list:

```
SELECT LTRIM(SYS_CONNECT_BY_PATH (warehouse_id,','),',') FROM
   (SELECT ROWNUM r, warehouse_id FROM warehouses)
   WHERE CONNECT_BY_ISLEAF = 1
   START WITH r = 1
   CONNECT BY r = PRIOR r + 1
   ORDER BY warehouse_id;

LTRIM(SYS_CONNECT_BY_PATH(WAREHOUSE_ID,','),',')
--------------------------------------------------------------------------------
1,2,3,4,5,6,7,8,9
```

CONNECT_BY_ROOT Examples

The following example returns the last name of each employee in department 110, each manager at the highest level above that employee in the hierarchy, the number of levels between manager and employee, and the path between the two:

```
SELECT last_name "Employee", CONNECT_BY_ROOT last_name "Manager",
   LEVEL-1 "Pathlen", SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   WHERE LEVEL > 1 and department_id = 110
   CONNECT BY PRIOR employee_id = manager_id
   ORDER BY "Employee", "Manager", "Pathlen", "Path";

Employee        Manager            Pathlen Path
--------------- --------------- ---------- ------------------------------
Gietz           Higgins                  1 /Higgins/Gietz
Gietz           King                     3 /King/Kochhar/Higgins/Gietz
Gietz           Kochhar                  2 /Kochhar/Higgins/Gietz
```

```
Higgins         King                      2 /King/Kochhar/Higgins
Higgins         Kochhar                   1 /Kochhar/Higgins
```

The following example uses a `GROUP BY` clause to return the total salary of each employee in department 110 and all employees above that employee in the hierarchy:

```
SELECT name, SUM(salary) "Total_Salary" FROM (
   SELECT CONNECT_BY_ROOT last_name as name, Salary
      FROM employees
      WHERE department_id = 110
      CONNECT BY PRIOR employee_id = manager_id)
      GROUP BY name
   ORDER BY name, "Total_Salary";

NAME                      Total_Salary
------------------------- ------------
Gietz                             8300
Higgins                          20300
King                             20300
Kochhar                          20300
```

See Also:

- LEVEL Pseudocolumn and CONNECT_BY_ISCYCLE Pseudocolumn for a discussion of how these pseudocolumns operate in a hierarchical query
- SYS_CONNECT_BY_PATH for information on retrieving the path of column values from root to node
- *order_by_clause* for more information on the `SIBLINGS` keyword of `ORDER BY` clauses
- subquery_factoring_clause, which supports recursive subquery factoring (recursive WITH) and lets you query hierarchical data. This feature is more powerful than `CONNECT BY` in that it provides depth-first search and breadth-first search, and supports multiple recursive branches.

The UNION [ALL], INTERSECT, MINUS Operators

You can combine multiple queries using the set operators `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`. All set operators have equal precedence. If a SQL statement contains multiple set operators, then Oracle Database evaluates them from the left to right unless parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and must be in the same data type group (such as numeric or character).

If component queries select character data, then the data type of the return values are determined as follows:

- If both queries select values of data type `CHAR` of equal length, then the returned values have data type `CHAR` of that length. If the queries select values of `CHAR` with different lengths, then the returned value is VARCHAR2 with the length of the larger CHAR value.

- If either or both of the queries select values of data type VARCHAR2, then the returned values have data type VARCHAR2.

If component queries select numeric data, then the data type of the return values is determined by numeric precedence:

- If any query selects values of type BINARY_DOUBLE, then the returned values have data type BINARY_DOUBLE.
- If no query selects values of type BINARY_DOUBLE but any query selects values of type BINARY_FLOAT, then the returned values have data type BINARY_FLOAT.
- If all queries select values of type NUMBER, then the returned values have data type NUMBER.

In queries using set operators, Oracle does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, Oracle returns an error.

> **See Also:**
>
> Table 2-8 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Examples

The following query is valid:

```
SELECT 3 FROM DUAL
   INTERSECT
SELECT 3f FROM DUAL;
```

This is implicitly converted to the following compound query:

```
SELECT TO_BINARY_FLOAT(3) FROM DUAL
   INTERSECT
SELECT 3f FROM DUAL;
```

The following query returns an error:

```
SELECT '3' FROM DUAL
   INTERSECT
SELECT 3f FROM DUAL;
```

Restrictions on the Set Operators

The set operators are subject to the following restrictions:

- The set operators are not valid on columns of type BLOB, CLOB, BFILE, VARRAY, or nested table.
- The UNION, INTERSECT, and MINUS operators are not valid on LONG columns.
- If the select list preceding the set operator contains an expression, then you must provide a column alias for the expression in order to refer to it in the *order_by_clause*.

- You cannot also specify the *for_update_clause* with the set operators.
- You cannot specify the *order_by_clause* in the *subquery* of these operators.
- You cannot use these operators in SELECT statements containing TABLE collection expressions.

> **Note:**
>
> To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

UNION Example

The following statement combines the results of two queries with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match data type (using the TO_CHAR function) when columns do not exist in one or the other table:

```
SELECT location_id, department_name "Department",
   TO_CHAR(NULL) "Warehouse"  FROM departments
   UNION
   SELECT location_id, TO_CHAR(NULL) "Department", warehouse_name
   FROM warehouses;

LOCATION_ID Department                     Warehouse
----------- ------------------------------ ---------------------------
       1400 IT
       1400                                Southlake, Texas
       1500 Shipping
       1500                                San Francisco
       1600                                New Jersey
       1700 Accounting
       1700 Administration
       1700 Benefits
       1700 Construction
       1700 Contracting
       1700 Control And Credit
...
```

UNION ALL Example

The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows:

```
SELECT product_id FROM order_items
UNION
SELECT product_id FROM inventories
ORDER BY product_id;

SELECT location_id  FROM locations
UNION ALL
SELECT location_id  FROM departments
ORDER BY location_id;
```

A location_id value that appears multiple times in either or both queries (such as '1700') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example

The following statement combines the results with the INTERSECT operator, which returns only those unique rows returned by both queries:

```
SELECT product_id FROM inventories
INTERSECT
SELECT product_id FROM order_items
ORDER BY product_id;
```

MINUS Example

The following statement combines results with the MINUS operator, which returns only unique rows returned by the first query but not by the second:

```
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items
ORDER BY product_id;
```

Sorting Query Results

Use the ORDER BY clause to order the rows selected by a query. Sorting by position is useful in the following cases:

- To order by a lengthy select list expression, you can specify its position in the ORDER BY clause rather than duplicate the entire expression.
- For compound queries containing set operators UNION, INTERSECT, MINUS, or UNION ALL, the ORDER BY clause must specify positions or aliases rather than explicit expressions. Also, the ORDER BY clause can appear only in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The mechanism by which Oracle Database sorts character values for the ORDER BY clause, also known as the collation, is specified by the NLS_SORT session parameter. If this parameter is not set, then its default is derived from the NLS_LANGUAGE session parameter. You can change the collation dynamically using the ALTER SESSION SET NLS_SORT statement. You can also apply a specific collation by including the character expressions to be sorted as arguments to the NLSSORT function, with the collation specified in the second parameter.

When character values are compared linguistically for the ORDER BY clause, they are first transformed to collation keys and then compared like RAW values. The collation keys are generated either explicitly as specified in NLSSORT or implicitly using the same method that NLSSORT uses. Both explicitly and implicitly generated collation keys are subject to the same restrictions that are described in "NLSSORT ". As a result of these restrictions, two values may compare as linguistically equal if they do not differ in the prefix that was used to produce the collation key, even if they differ in the rest of the value.

> **See Also:**
>
> NLSSORT and *Oracle Database Globalization Support Guide* for information on the NLS parameters

Joins

A **join** is a query that combines rows from two or more tables, views, or materialized views. Oracle Database performs a join whenever multiple tables appear in the `FROM` clause of the query. The select list of the query can select any columns from any of these tables. If any two of these tables have a column name in common, then you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain at least one **join condition**, either in the `FROM` clause or in the `WHERE` clause. The join condition compares two columns, each from a different table. To execute a join, Oracle Database combines pairs of rows, each containing one row from each table, for which the join condition evaluates to `TRUE`. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, any available statistics for the tables.

A `WHERE` clause that contains a join condition can also contain other conditions that refer to columns of only one table. These conditions can further restrict the rows returned by the join query.

> **Note:**
>
> You cannot specify LOB columns in the `WHERE` clause if the `WHERE` clause contains the join condition. The use of LOBs in `WHERE` clauses is also subject to other restrictions. See *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

Equijoins

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. Depending on the internal algorithm the optimizer chooses to execute the join, the total size of the columns in the equijoin condition in a single table may be limited to the size of a data block minus some overhead. The size of a data block is specified by the initialization parameter `DB_BLOCK_SIZE`.

> **See Also:**
>
> "Using Join Queries: Examples"

Band Joins

A **band join** is a special type of nonequijoin in which key values in one data set must fall within the specified range ("band") of the second data set. The same table can serve as both the first and second data sets.

> **See Also:**
>
> - *Database SQL Tuning Guide* for more information on band joins
> - USE_BAND Hint
> - NO_USE_BAND Hint

Self Joins

A **self join** is a join of a table to itself. This table appears twice in the `FROM` clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle Database combines and returns rows of the table that satisfy the join condition.

> **See Also:**
>
> "Using Self Joins: Example"

Cartesian Products

If two tables in a join query have no join condition, then Oracle Database returns their **Cartesian product**. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product. If a query joins three or more tables and you do not specify a join condition for a specific pair, then the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Inner Joins

An **inner join** (sometimes called a **simple join**) is a join of two or more tables that returns only those rows that satisfy the join condition.

Outer Joins

An **outer join** extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

- To write a query that performs an outer join of tables A and B and returns all rows from A (a **left outer join**), use the `LEFT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to all columns of B in the join condition in the `WHERE` clause. For all rows in A that have no matching rows in B, Oracle Database returns null for any select list expressions containing columns of B.
- To write a query that performs an outer join of tables A and B and returns all rows from B (a **right outer join**), use the `RIGHT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to all columns of A in the join condition in the `WHERE` clause. For all rows in B that have no matching rows in A, Oracle returns null for any select list expressions containing columns of A.
- To write a query that performs an outer join and returns all rows from A and B, extended with nulls if they do not satisfy the join condition (a **full outer join**), use the `FULL [OUTER] JOIN` syntax in the `FROM` clause.

You cannot compare a column with a subquery in the `WHERE` clause of any outer join, regardless which form you specify.

You can use outer joins to fill gaps in sparse data. Such a join is called a **partitioned outer join** and is formed using the *`query_partition_clause`* of the *`join_clause`* syntax. Sparse data is data that does not have rows for all possible values of a dimension such as time or department. For example, tables of sales data typically do not have rows for products that had no sales on a given date. Filling data gaps is useful in situations where data sparsity complicates analytic computation or where some data might be missed if the sparse data is queried directly.

> **See Also:**
>
> - *join_clause* for more information about using outer joins to fill gaps in sparse data
> - *Oracle Database Data Warehousing Guide* for a complete discussion of group outer joins and filling gaps in sparse data

Oracle recommends that you use the `FROM` clause `OUTER JOIN` syntax rather than the Oracle join operator. Outer join queries that use the Oracle join operator (+) are subject to the following rules and restrictions, which do not apply to the `FROM` clause `OUTER JOIN` syntax:

- You cannot specify the (+) operator in a query block that also contains `FROM` clause join syntax.
- The (+) operator can appear only in the `WHERE` clause or, in the context of left-correlation (when specifying the `TABLE` clause) in the `FROM` clause, and can be applied only to a column of a table or view.

- If A and B are joined by multiple join conditions, then you must use the (+) operator in all of these conditions. If you do not, then Oracle Database will return only the rows resulting from a simple join, but without a warning or error to advise you that you do not have the results of an outer join.
- The (+) operator does not produce an outer join if you specify one table in the outer query and the other table in an inner query.
- You cannot use the (+) operator to outer-join a table to itself, although self joins are valid. For example, the following statement is **not** valid:

  ```
  -- The following statement is not valid:
  SELECT employee_id, manager_id
     FROM employees
     WHERE employees.manager_id(+) = employees.employee_id;
  ```

  However, the following self join is valid:

  ```
  SELECT e1.employee_id, e1.manager_id, e2.employee_id
     FROM employees e1, employees e2
     WHERE e1.manager_id(+) = e2.employee_id
     ORDER BY e1.employee_id, e1.manager_id, e2.employee_id;
  ```

- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain one or more columns marked with the (+) operator.
- A `WHERE` condition containing the (+) operator cannot be combined with another condition using the `OR` logical operator.
- A `WHERE` condition cannot use the `IN` comparison condition to compare a column marked with the (+) operator with an expression.

If the `WHERE` clause contains a condition that compares a column from table B with a constant, then the (+) operator must be applied to the column so that Oracle returns the rows from table A for which it has generated nulls for this column. Otherwise Oracle returns only the results of a simple join.

In previous releases of Oracle Database, in a query that performed outer joins of more than two pairs of tables, a single table could be the null-generated table for only one other table. Beginning with Oracle Database 12*c*, a single table can be the null-generated table for multiple tables. For example, the following statement is allowed in Oracle Database 12*c*:

```
SELECT * FROM A, B, D
  WHERE A.c1 = B.c2(+) and D.c3 = B.c4(+);
```

In this example, `B`, the null-generated table, is outer-joined to two tables, `A` and `D`. Refer to SELECT for the syntax for an outer join.

Antijoins

An antijoin returns rows from the left side of the predicate for which there are no corresponding rows on the right side of the predicate. It returns rows that fail to match (`NOT IN`) the subquery on the right side.

> **See Also:**
>
> "Using Antijoins: Example"

Semijoins

A semijoin returns rows that match an EXISTS subquery without duplicating rows from the left side of the predicate when multiple rows on the right side satisfy the criteria of the subquery.

Semijoin and antijoin transformation cannot be done if the subquery is on an OR branch of the WHERE clause.

> **See Also:**
>
> "Using Semijoins: Example"

Using Subqueries

A **subquery** answers multiple-part questions. For example, to determine who works in Taylor's department, you can first use a subquery to determine the department in which Taylor works. You can then answer the original question with the parent SELECT statement. A subquery in the FROM clause of a SELECT statement is also called an **inline view**. you can nest any number of subqueries in an inline view. A subquery in the WHERE clause of a SELECT statement is also called a **nested subquery**. You can nest up to 255 levels of subqueries in the a nested subquery.

A subquery can contain another subquery. Oracle Database imposes no limit on the number of subquery levels in the FROM clause of the top-level query. You can nest up to 255 levels of subqueries in the WHERE clause.

If columns in a subquery have the same name as columns in the containing statement, then you must prefix any reference to the column of the table from the containing statement with the table name or alias. To make your statements easier to read, always qualify the columns in a subquery with the name or alias of the table, view, or materialized view.

Oracle performs a **correlated subquery** when a nested subquery references a column from a table referred to a parent statement one or more levels above the subquery or nested subquery. The parent statement can be a SELECT, UPDATE, or DELETE statement in which the subquery is nested. A correlated subquery conceptually is evaluated once for each row processed by the parent statement. However, the optimizer may choose to rewrite the query as a join or use some other technique to formulate a query that is semantically equivalent. Oracle resolves unqualified columns in the subquery by looking in the tables named in the subquery and then in the tables named in the parent statement.

A correlated subquery answers a multiple-part question whose answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

> **See Also:**
>
> "Using Correlated Subqueries: Examples"

Use subqueries for the following purposes:

- To define the set of rows to be inserted into the target table of an `INSERT` or `CREATE TABLE` statement
- To define the set of rows to be included in a view or materialized view in a `CREATE VIEW` or `CREATE MATERIALIZED VIEW` statement
- To define one or more values to be assigned to existing rows in an `UPDATE` statement
- To provide values for conditions in a `WHERE` clause, `HAVING` clause, or `START WITH` clause of `SELECT`, `UPDATE`, and `DELETE` statements
- To define a table to be operated on by a containing query

  You do this by placing the subquery in the `FROM` clause of the containing query as you would a table name. You may use subqueries in place of tables in this way as well in `INSERT`, `UPDATE`, and `DELETE` statements.

  Subqueries so used can employ correlation variables, both defined within the subquery itself and those defined in query blocks containing the subquery. Refer to *table_collection_expression* for more information.

  Scalar subqueries, which return a single column value from a single row, are a valid form of expression. You can use scalar subquery expressions in most of the places where *`expr`* is called for in syntax. Refer to "Scalar Subquery Expressions " for more information.

Unnesting of Nested Subqueries

Subqueries are **nested** when they appear in the `WHERE` clause of the parent statement. When Oracle Database evaluates a statement with a nested subquery, it must evaluate the subquery portion multiple times and may overlook some efficient access paths or joins.

Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins. The optimizer can unnest most subqueries, with some exceptions. Those exceptions include hierarchical subqueries and subqueries that contain a `ROWNUM` pseudocolumn, one of the set operators, a nested aggregate function, or a correlated reference to a query block that is not the immediate outer query block of the subquery.

Assuming no restrictions exist, the optimizer automatically unnests some (but not all) of the following nested subqueries:

- Uncorrelated `IN` subqueries
- `IN` and `EXISTS` correlated subqueries, as long as they do not contain aggregate functions or a `GROUP BY` clause

You can enable **extended subquery unnesting** by instructing the optimizer to unnest additional types of subqueries:

- You can unnest an uncorrelated `NOT IN` subquery by specifying the `HASH_AJ` or `MERGE_AJ` hint in the subquery.
- You can unnest other subqueries by specifying the `UNNEST` hint in the subquery.

> **See Also:**
>
> "Hints " for information on hints

Selecting from the DUAL Table

`DUAL` is a table automatically created by Oracle Database along with the data dictionary. `DUAL` is in the schema of the user `SYS` but is accessible by the name `DUAL` to all users. It has one column, `DUMMY`, defined to be `VARCHAR2(1)`, and contains one row with a value `X`. Selecting from the `DUAL` table is useful for computing a constant expression with the `SELECT` statement. Because `DUAL` has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table. Refer to "About SQL Functions " for many examples of selecting a constant value from `DUAL`.

> **Note:**
>
> Beginning with Oracle Database 10*g* Release 1, logical I/O is not performed on the `DUAL` table when computing an expression that does not include the `DUMMY` column. This optimization is listed as `FAST DUAL` in the execution plan. If you `SELECT` the `DUMMY` column from `DUAL`, then this optimization does not take place and logical I/O occurs.

Distributed Queries

The Oracle distributed database management system architecture lets you access data in remote databases using Oracle Net and an Oracle Database server. You can identify a remote table, view, or materialized view by appending @*`dblink`* to the end of its name. The *`dblink`* must be a complete or partial name for a database link to the database containing the remote table, view, or materialized view.

> **See Also:**
>
> References to Objects in Remote Databases for more information on referring to database links

Restrictions on Distributed Queries

Distributed queries are currently subject to the restriction that all tables locked by a `FOR UPDATE` clause and all tables with `LONG` columns selected by the query must be located on the same database. In addition, Oracle Database currently does not support distributed queries that select user-defined types or object `REF` data types on remote tables.

10 SQL Statements: ADMINISTER KEY MANAGEMENT to ALTER JAVA

This chapter lists the various types of SQL statements and then describes the first set (in alphabetical order) of SQL statements. The remaining SQL statements appear in alphabetical order in the subsequent chapters.

This chapter contains the following sections:

- Types of SQL Statements
- How the SQL Statement Chapters are Organized
- ADMINISTER KEY MANAGEMENT
- ALTER ANALYTIC VIEW
- ALTER ATTRIBUTE DIMENSION
- ALTER AUDIT POLICY (Unified Auditing)
- ALTER CLUSTER
- ALTER DATABASE
- ALTER DATABASE LINK
- ALTER DIMENSION
- ALTER DISKGROUP
- ALTER FLASHBACK ARCHIVE
- ALTER FUNCTION
- ALTER HIERARCHY
- ALTER INDEX
- ALTER INDEXTYPE
- ALTER INMEMORY JOIN GROUP
- ALTER JAVA

Types of SQL Statements

The lists in the following sections provide a functional summary of SQL statements and are divided into these categories:

- Data Definition Language (DDL) Statements
- Data Manipulation Language (DML) Statements
- Transaction Control Statements
- Session Control Statements
- System Control Statement

- Embedded SQL Statements

Data Definition Language (DDL) Statements

Data definition language (DDL) statements let you to perform these tasks:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster
- Establish auditing options
- Add comments to the data dictionary

The `CREATE`, `ALTER`, and `DROP` commands require exclusive access to the specified object. For example, an `ALTER TABLE` statement fails if another user has an open transaction on the specified table.

The `GRANT`, `REVOKE`, `ANALYZE`, `AUDIT`, and `COMMENT` commands do not require exclusive access to the specified object. For example, you can analyze a table while other users are updating the table.

Oracle Database implicitly commits the current transaction before and after every DDL statement.

A DDL statement is either blocking or nonblocking, and both types of DDL statements require exclusive locks on internal structures.

> **See Also:**
>
> *Oracle Database Development Guide* to learn about the difference between blocking and nonblocking DDL

Many DDL statements may cause Oracle Database to recompile or reauthorize schema objects. For information on how Oracle Database recompiles and reauthorizes schema objects and the circumstances under which a DDL statement would cause this, see *Oracle Database Concepts*.

DDL statements are supported by PL/SQL with the use of the `DBMS_SQL` package.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about this package

The DDL statements are:

`ALTER` ... (All statements beginning with `ALTER`, except `ALTER SESSION` and `ALTER SYSTEM`—see "Session Control Statements " and "System Control Statement ")

`ANALYZE`

`ASSOCIATE STATISTICS`

- `AUDIT`
- `COMMENT`
- `CREATE` ... (All statements beginning with `CREATE`)
- `DISASSOCIATE STATISTICS`
- `DROP` ... (All statements beginning with `DROP`)
- `FLASHBACK` ... (All statements beginning with `FLASHBACK`)
- `GRANT`
- `NOAUDIT`
- `PURGE`
- `RENAME`
- `REVOKE`
- `TRUNCATE`

Data Manipulation Language (DML) Statements

Data manipulation language (DML) statements access and manipulate data in existing schema objects. These statements do not implicitly commit the current transaction. The data manipulation language statements are:

- `CALL`
- `DELETE`
- `EXPLAIN PLAN`
- `INSERT`
- `LOCK TABLE`
- `MERGE`
- `SELECT`
- `UPDATE`

The `SELECT` statement is a limited form of DML statement in that it can only access data in the database. It cannot manipulate data stored in the database, although it can manipulate the accessed data before returning the results of the query.

The `SELECT` statement is supported in PL/SQL only when executed dynamically. However, you can use the similar PL/SQL statement `SELECT INTO` in PL/SQL code, and you do not have to execute it dynamically. The `CALL` and `EXPLAIN PLAN` statements are supported in PL/SQL only when executed dynamically. All other DML statements are fully supported in PL/SQL.

Transaction Control Statements

Transaction control statements manage changes made by DML statements. The transaction control statements are:

- `COMMIT`
- `ROLLBACK`
- `SAVEPOINT`
- `SET TRANSACTION`
- `SET CONSTRAINT`

All transaction control statements, except certain forms of the `COMMIT` and `ROLLBACK` commands, are supported in PL/SQL. For information on the restrictions, see COMMIT and ROLLBACK .

Session Control Statements

Session control statements dynamically manage the properties of a user session. These statements do not implicitly commit the current transaction.

PL/SQL does not support session control statements. The session control statements are:

```
ALTER SESSION
SET ROLE
```

System Control Statement

The single system control statement, `ALTER SYSTEM`, dynamically manages the properties of an Oracle Database instance. This statement does not implicitly commit the current transaction and is not supported in PL/SQL.

Embedded SQL Statements

Embedded SQL statements place DDL, DML, and transaction control statements within a procedural language program. Embedded SQL is supported by the Oracle precompilers and is documented in the following books:

- *Pro*COBOL Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*

How the SQL Statement Chapters are Organized

All SQL statements in this book are organized into the following sections:

Syntax

The syntax diagrams show the keywords and parameters that make up the statement.

> **Note:**
>
> Not all keywords and parameters are valid in all circumstances. Be sure to refer to the "Semantics" section of each statement and clause to learn about any restrictions on the syntax.

Purpose

The "Purpose" section describes the basic uses of the statement.

Prerequisites

The "Prerequisites" section lists privileges you must have and steps that you must take before using the statement. In addition to the prerequisites listed, most statements also require that the database be opened by your instance, unless otherwise noted.

Semantics

The "Semantics" section describes the purpose of the keywords, parameters, and clauses that make up the syntax, as well as restrictions and other usage notes that may apply to them. (The conventions for keywords and parameters used in this chapter are explained in the "Preface" of this reference.)

Examples

The "Examples" section shows how to use the various clauses and parameters of the statement.

ADMINISTER KEY MANAGEMENT

Purpose

The `ADMINISTER KEY MANAGEMENT` statement provides a unified key management interface for Transparent Data Encryption. Use this statement to:

- Manage software and hardware keystores
- Manage encryption keys
- Manage secrets

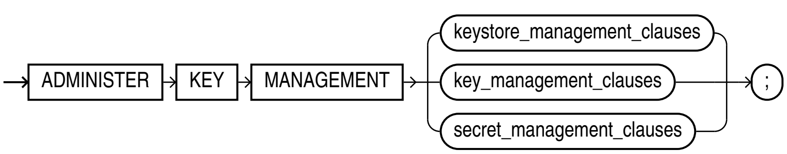
Prerequisites

You must have the `ADMINISTER KEY MANAGEMENT` or `SYSKM` system privilege.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root and you must have the commonly granted `ADMINISTER KEY MANAGEMENT` or `SYSKM` privilege.

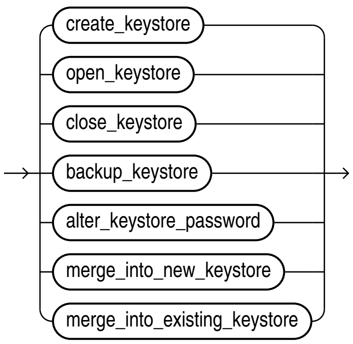
Syntax

***administer_key_management*::=**



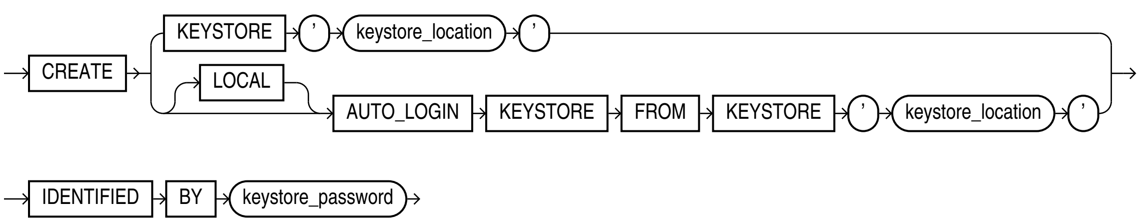
(*keystore_management_clauses*::=, *key_management_clauses*::=, *secret_management_clauses*::=)

***keystore_management_clauses*::=**

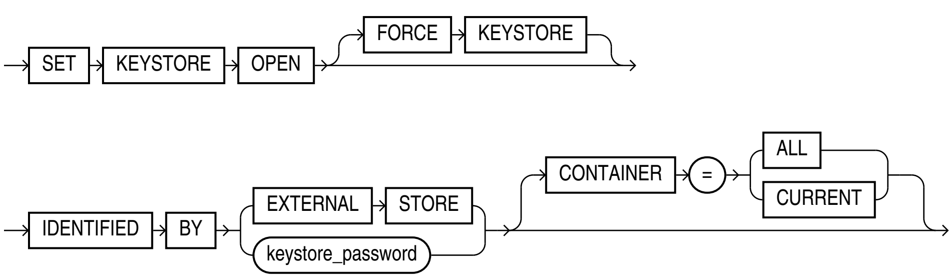


(*create_keystore*::=, *open_keystore*::=, *close_keystore*::=, *backup_keystore*::=, *alter_keystore_password*::=, *merge_into_new_keystore*::=, *merge_into_existing_keystore*::=)

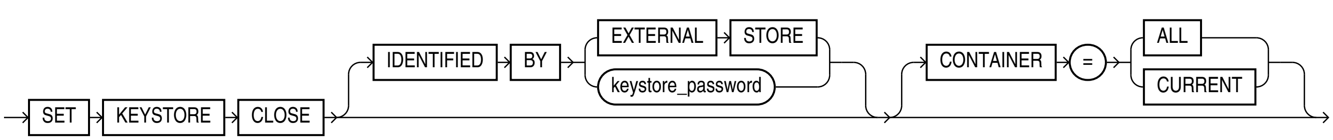
***create_keystore*::=**



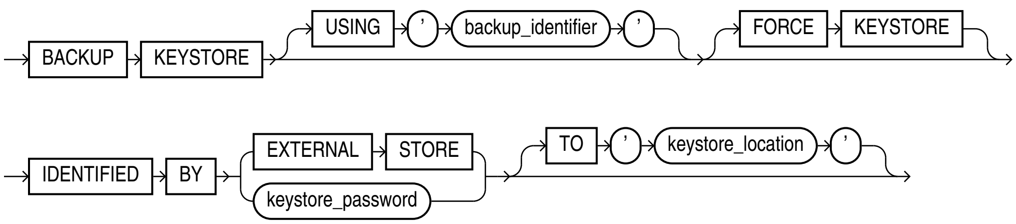
***open_keystore*::=**



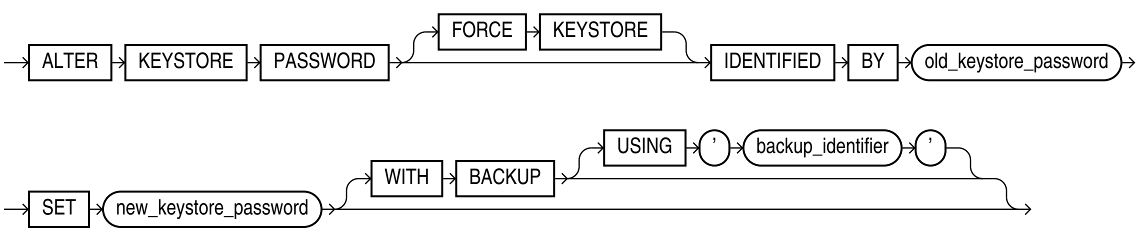
***close_keystore*::=**



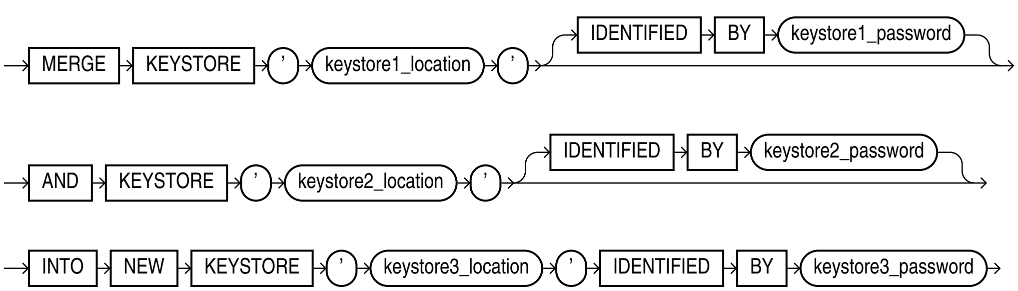
***backup_keystore*::=**



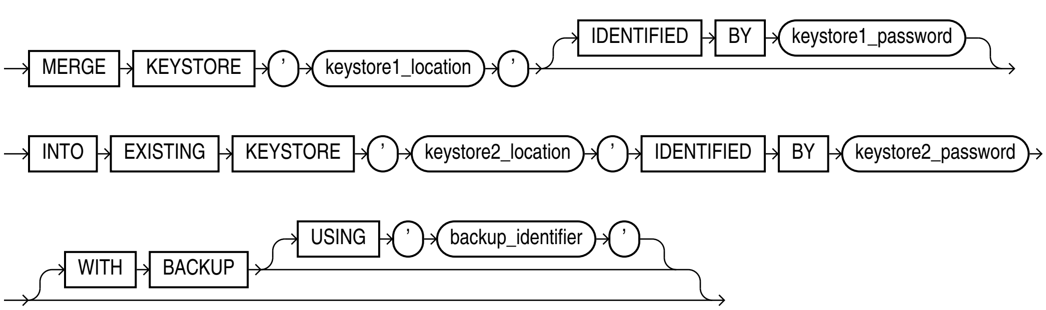
***alter_keystore_password*::=**



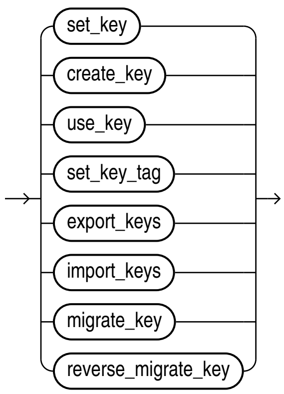
***merge_into_new_keystore*::=**



***merge_into_existing_keystore*::=**

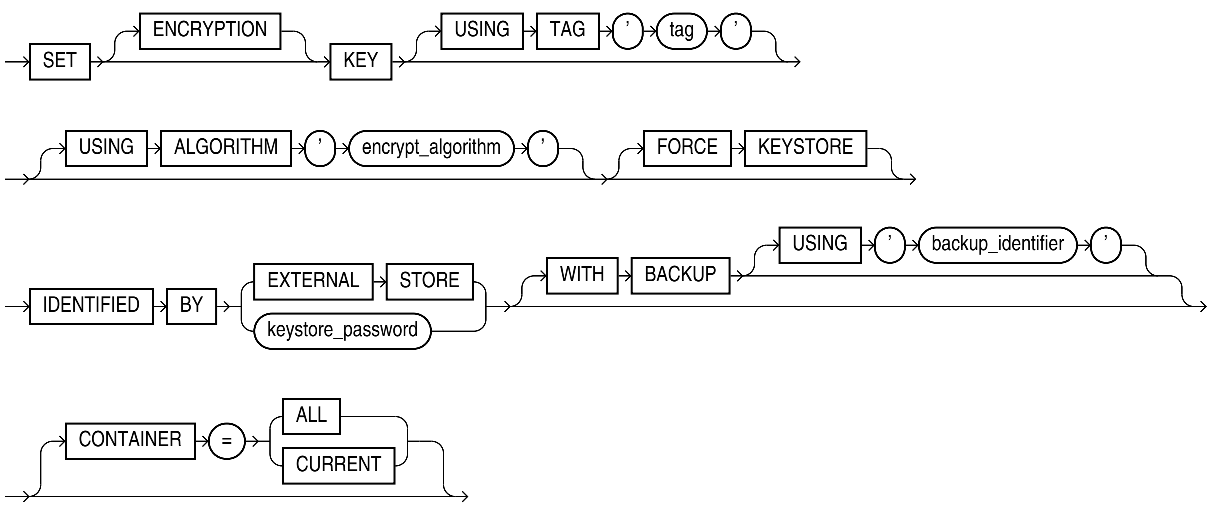


***key_management_clauses*::=**

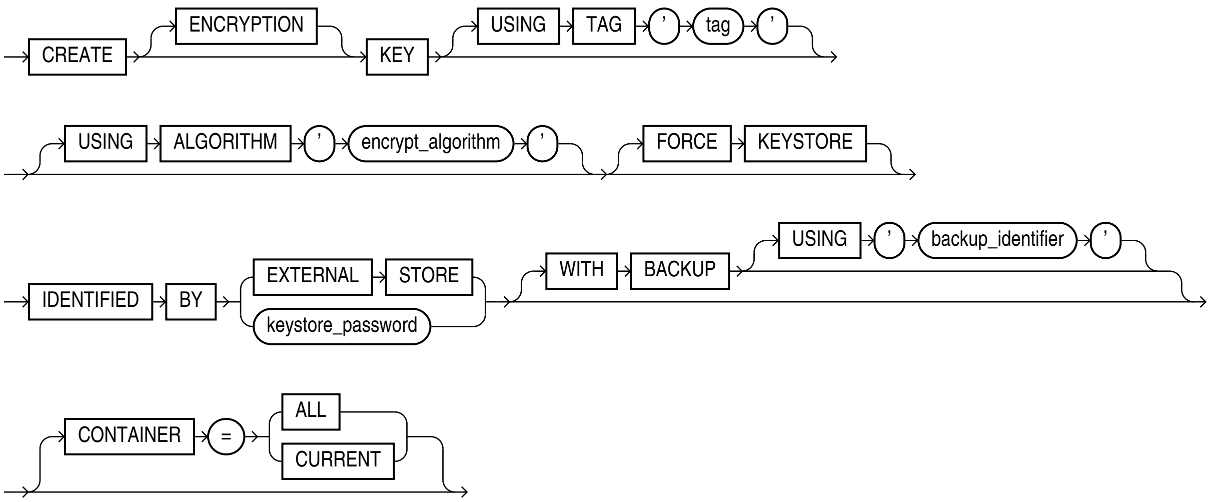


(*set_key*::=, *create_key*::=, *use_key*::=, *set_key_tag*::=, *export_keys*::=, *import_keys*::=, *migrate_key*::=, *reverse_migrate_key*::=)

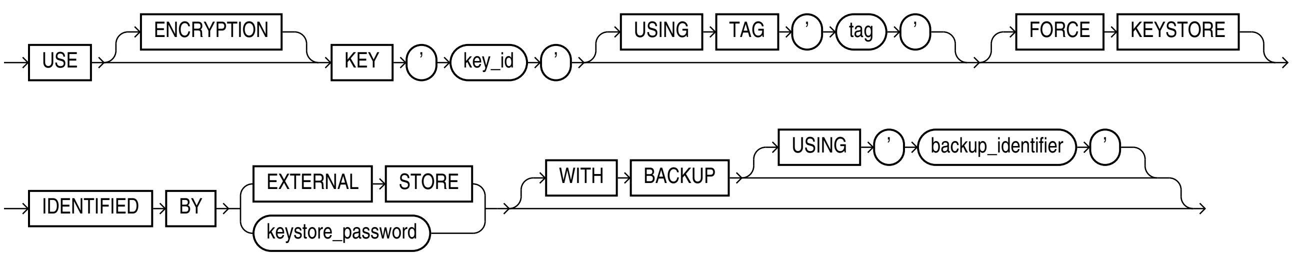
***set_key*::=**



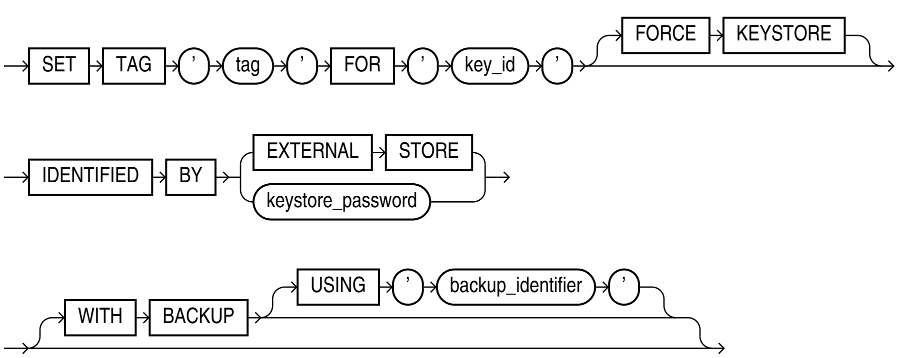
***create_key*::=**



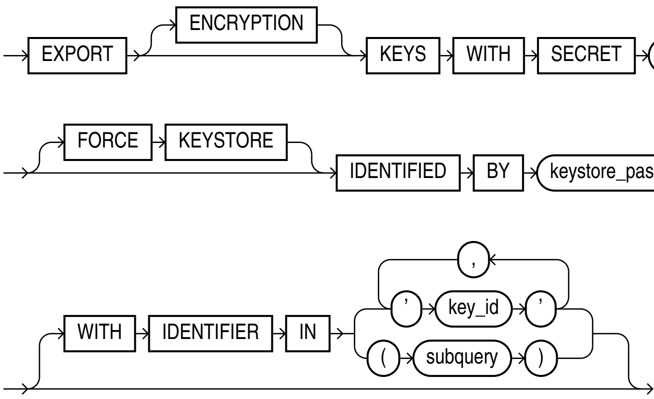
***use_key*::=**



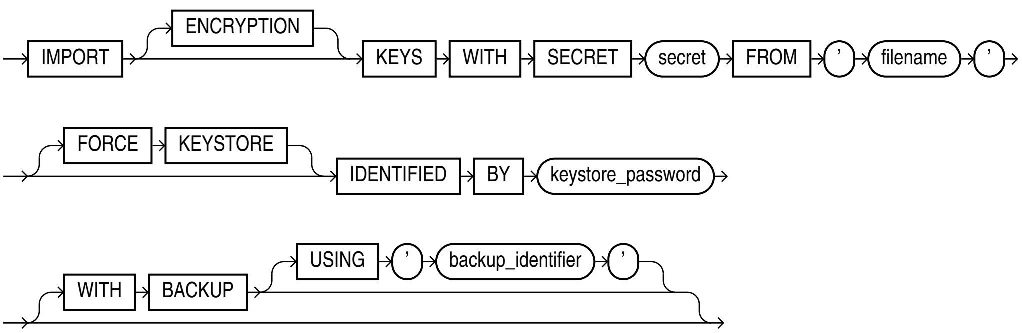
***set_key_tag*::=**



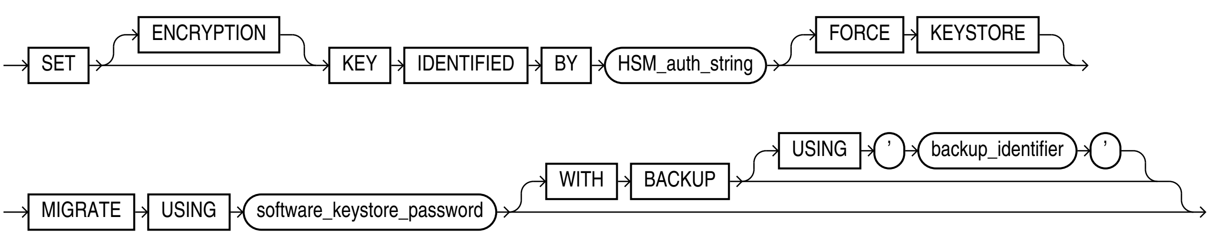
***export_keys*::=**



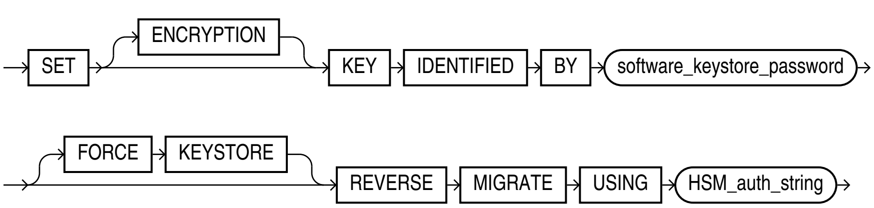
***import_keys*::=**



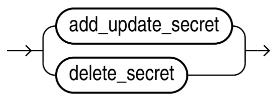
***migrate_key*::=**



***reverse_migrate_key*::=**

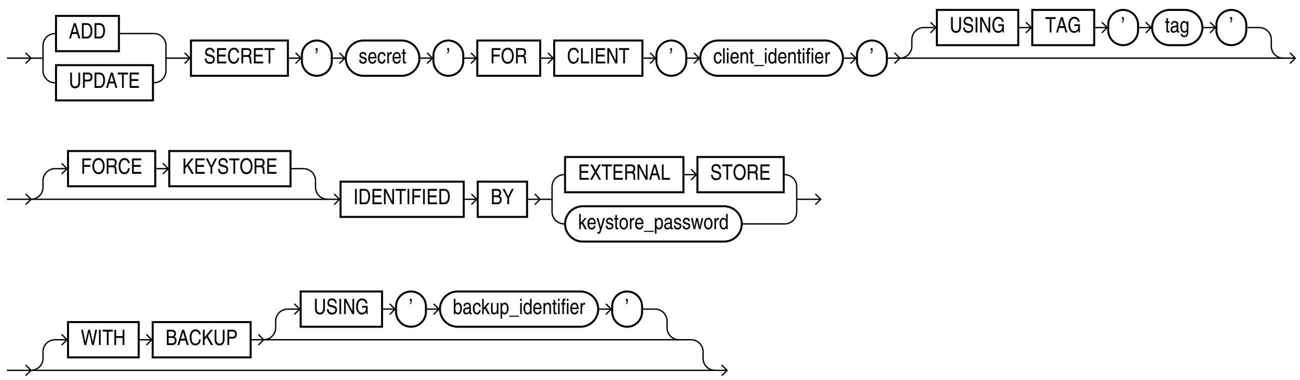


***secret_management_clauses*::=**

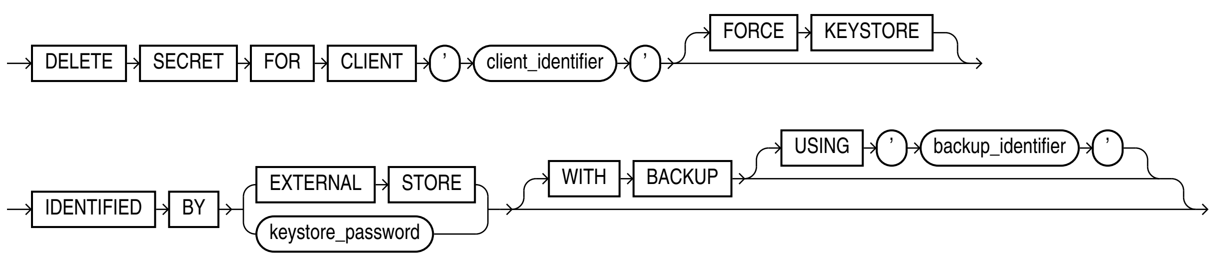


(*add_update_secret*::=, *delete_secret*::=)

***add_update_secret*::=**



***delete_secret*::=**



Semantics

keystore_management_clauses

Use these clauses to perform the following keystore management operations:

- Create a software keystore
- Open and close a software keystore or a hardware keystore
- Back up a password-based software keystore
- Change the password of a password-based software keystore
- Merge two existing software keystores into a new password-based software keystore
- Merge one existing software keystore into an existing password-based software keystore

create_keystore

This clause lets you create the following types of software keystores: password-based software keystores and auto-login software keystores. To issue this clause in a multitenant environment, you must be connected to the root.

CREATE KEYSTORE

Specify this clause to create a password-based software keystore.

- For `keystore_location`, specify the full path name of the software keystore directory. The keystore will be created in this directory in a file named `ewallet.p12`. Refer to *Oracle Database Advanced Security Guide* to learn how to determine the software keystore directory for your system.
- Use the `IDENTIFIED BY` clause to set the password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.

CREATE [LOCAL] AUTO_LOGIN KEYSTORE

Specify this clause to create an auto-login software keystore. An auto-login software keystore is created from an existing password-based software keystore. The auto-login keystore has a system-generated password. It is stored in a PKCS#12-based file named `cwallet.sso` in the same directory as the password-based software keystore.

- By default, Oracle creates an auto-login keystore, which can be opened from computers other than the computer on which the keystore resides. If you specify the `LOCAL` keyword, then Oracle Database creates a local auto-login keystore, which can be opened only from the computer on which the keystore resides.
- For `keystore_location`, specify the full path name of the directory in which the existing password-based software keystore resides. The password-based software keystore can be open or closed.
- Use the `IDENTIFIED BY` clause to specify the password for the existing password-based software keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.

Restriction on Creating Keystores

You can create at most one password-based software keystore and one auto-login software keystore, either local or not, in any single directory.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on creating software keystores

open_keystore

This clause lets you open a password-based software keystore or a hardware keystore.

> **Note:**
>
> You do not need to use this clause to open auto-login and local auto-login software keystores because they are opened automatically when they are required—that is, when the master encryption key is accessed.

- The `FORCE KEYSTORE` clause is useful when opening a keystore in a PDB. It ensures that the CDB root keystore is open before opening the PDB keystore. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the `IDENTIFIED BY` clause to specify the password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- The `CONTAINER` clause applies when you are connected to a CDB.

  If the current container is a pluggable database (PDB), then specify `CONTAINER = CURRENT` to open the keystore in the PDB. The keystore must be open in the root before you open it in the PDB.

  If the current container is the root, then specify `CONTAINER = CURRENT` to open the keystore in the root, or specify `CONTAINER = ALL` to open the keystore in the root and in all PDBs.

  If you omit this clause, then `CONTAINER = CURRENT` is the default.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on opening password-based software keystores and hardware keystores

close_keystore

This clause lets you close a password-based software keystore, an auto-login software keystore, or a hardware keystore. Closing a keystore disables all encryption and decryption operations. Any attempt to encrypt or decrypt data or access encrypted data results in an error.

- To close a password-based software keystore or a hardware keystore, specify the `IDENTIFIED BY` clause. Refer to "Notes on Specifying Keystore Passwords" for more information.
- To close an auto-login keystore, do not specify the `IDENTIFIED BY` clause. Before you close an auto-login keystore, check the `WALLET_TYPE` column of the `V$ENCRYPTION_WALLET` view. If it returns `AUTOLOGIN`, then you can close the keystore. Otherwise, if you attempt to close the keystore, then an error occurs.
- The `CONTAINER` clause applies when you are connected to a CDB.

  If the current container is a PDB, then specify `CONTAINER = CURRENT` to close the keystore in the PDB.

  If the current container is the root, then the `CONTAINER = CURRENT` and `CONTAINER = ALL` clauses have the same effect; both clauses close the keystore in the root and in all PDBs.

  If you omit this clause, then `CONTAINER = CURRENT` is the default.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on closing keystores

backup_keystore

This clause lets you back up a password-based software keystore. The keystore must be open.

- By default, Oracle Database creates a backup file with a name of the form `ewallet_`*`timestamp`*`.p12`, where *`timestamp`* is the file creation timestamp in UTC format. The optional `USING '`*`backup_identifier`*`'` clause lets you specify a backup identifier which is added to the backup file name. For example, if you specify a backup identifier of `'Backup1'`, then Oracle Database creates a backup file with a name of the form `ewallet_`*`timestamp`*`_Backup1.p12`.
- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the `IDENTIFIED BY` clause to specify the password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- The optional `TO` '*`keystore_location`*' clause lets you specify the directory in which the backup file is created. If you omit this clause, then the backup is created in the same directory as the keystore that you are backing up.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on backing up password-based software keystores

alter_keystore_password

This clause lets you change the password for a password-based software keystore. The keystore must be open.

- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- For *`old_keystore_password`*, specify the old password for the keystore. For *`new_keystore_password`*, specify the new password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- The optional `WITH BACKUP` clause instructs the database to create a backup of the keystore before changing the password. Refer to "Notes on the WITH BACKUP Clause" for more information.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on changing a password-based software keystore password

merge_into_new_keystore

This clause lets you merge two software keystores into a new keystore. The keys and attributes in the two constituent keystores are added to the new keystore. The constituent keystores can be password-based or auto-login (including local auto-login) software keystores; they can be open or closed. The new keystore is a password-based software keystore. It is in a closed state when the merge completes. Any or none of the keystores specified in this clause can be the keystore configured for use by the database.

- For *keystore1_location*, specify the full path name of the directory in which the first keystore resides.
- Specify IDENTIFIED BY *keystore1_password* only if the first keystore is a password-based software keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- For *keystore2_location*, specify the full path name of the directory in which the second keystore resides.
- Specify IDENTIFIED BY *keystore2_password* only if the second keystore is a password-based software keystore.
- For *keystore3_location*, specify the full path name of the directory in which the new keystore is created.
- For *keystore3_password*, specify the password for the new keystore.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on merging software keystores

merge_into_existing_keystore

This clause lets you merge a software keystore into another existing software keystore. The keys and attributes in the keystore from which you merge are added to the keystore into which you merge. The keystore from which you merge can be a password-based or auto-login (including local auto-login) software keystore; it can be open or closed. The keystore into which you merge must be a password-based software keystore. It can be open or closed when the merge begins. However, it will be in a closed state when the merge completes. Either or neither of the keystores specified in this clause can be the keystore configured for use by the database.

- For *keystore1_location*, specify the full path name of the directory in which the keystore from which you merge resides.
- Specify IDENTIFIED BY *keystore1_password* only if the keystore from which you merge is a password-based software keystore.

- For *keystore2_location*, specify the full path name of the directory in which the keystore into which you merge resides.
- For *keystore2_password*, specify the password for the keystore into which you merge.
- The optional `WITH BACKUP` clause instructs the database to create a backup of the keystore into which you merge before performing the merge. Refer to "Notes on the WITH BACKUP Clause" for more information.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on merging software keystores

key_management_clauses

Use these clauses to perform the following key management operations:

- Create and activate a master encryption key
- Set the tag for an encryption key
- Export encryption keys from a keystore into a file
- Import encryption keys from a file into a keystore
- Migrate from a password-based software keystore to a hardware keystore
- Migrate from a hardware keystore to a password-based software keystore

set_key

This clause creates a new master encryption key and activates it. You can use this clause to create the first master encryption key in a keystore or to rotate (change) the master encryption key. If a master encryption key is active when you use this clause, then it is deactivated before the new master encryption key is activated. The keystore that contains the key can be a password-based software keystore or a hardware keystore. The keystore must be open.

- The `ENCRYPTION` keyword is optional and is provided for semantic clarity.
- Specify the optional `USING TAG` clause to associate a tag to the new master encryption key. Refer to "Notes on the USING TAG Clause" for more information.
- If you specify the `USING ALGORITHM` clause, then the database creates a master encryption key that conforms to the specified encryption algorithm. For *encrypt_algorithm*, you can specify `AES256`, `ARIA256`, `GOST256`, or `SEED128`. To specify this clause, the `COMPATIBLE` initialization parameter must be set to `12.2` or higher. If you omit this clause, then the default is `AES256`.

  The ARIA, SEED, and GOST algorithms are country-specific national and government standards for encryption and hashing. See *Oracle Database Security Guide* for more information.
- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.

- Use the `IDENTIFIED BY` clause to specify the password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the `WITH BACKUP` clause, and optionally the `USING 'backup_identifier'` clause, to create a backup of the keystore before the new master encryption key is created. Refer to "Notes on the WITH BACKUP Clause" for more information.
- The `CONTAINER` clause applies when you are connected to a CDB.

  If the current container is a PDB, then specify `CONTAINER = CURRENT` to create and activate a new master encryption key in the PDB. A master encryption key must exist in the root before you create a master encryption key in the PDB.

  If the current container is the root, then specify `CONTAINER = CURRENT` to create and activate a new master encryption key in the root, or specify `CONTAINER = ALL` to create and activate new master encryption keys in the root and in all PDBs.

  If you omit this clause, then `CONTAINER = CURRENT` is the default.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on creating and activating a master encryption key

create_key

This clause lets you create a master encryption key for later use. You can subsequently activate the key by using the *use_key* clause. The keystore that contains the key can be a password-based software keystore or a hardware keystore. The keystore must be open.

- The `ENCRYPTION` keyword is optional and is provided for semantic clarity.
- Specify the optional `USING TAG` clause to associate a tag to the encryption key. Refer to "Notes on the USING TAG Clause" for more information.
- If you specify the `USING ALGORITHM` clause, then the database creates a master encryption key that conforms to the specified encryption algorithm. For *`encrypt_algorithm`*, you can specify `AES256`, `ARIA256`, `GOST256`, or `SEED128`. To specify this clause, the `COMPATIBLE` initialization parameter must be set to `12.2` or higher. If you omit this clause, then the default is `AES256`.

  The ARIA, SEED, and GOST algorithms are country-specific national and government standards for encryption and hashing. See *Oracle Database Security Guide* for more information.
- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the `IDENTIFIED BY` clause to specify the password for the keystore in which the key will be created. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the `WITH BACKUP` clause, and optionally the `USING 'backup_identifier'` clause, to create a backup of the keystore before the key is created. Refer to "Notes on the WITH BACKUP Clause" for more information.
- The `CONTAINER` clause applies when you are connected to a CDB.

If the current container is a PDB, then specify `CONTAINER = CURRENT` to create a master encryption key in the PDB. A master encryption key must exist in the root before you create a master encryption key in the PDB

If the current container is the root, then specify `CONTAINER = CURRENT` to create a master encryption key in the root, or specify `CONTAINER = ALL` to create master encryption keys in the root and in all PDBs.

If you omit this clause, then `CONTAINER = CURRENT` is the default.

See Also:

Oracle Database Advanced Security Guide for more information on creating a master encryption key for later use

use_key

This clause lets you activate a master encryption key that has already been created. If a master encryption key is active when you use this clause, then it is deactivated before the new master encryption key is activated. The keystore that contains the key can be a password-based software keystore or a hardware keystore. The keystore must be open

- The `ENCRYPTION` keyword is optional and is provided for semantic clarity.
- For *`key_id`*, specify the identifier of the key that you want to activate. You can find the key identifier by querying the `KEY_ID` column of the `V$ENCRYPTION_KEYS` view.
- Specify the optional `USING TAG` clause to associate a tag to the encryption key. Refer to "Notes on the USING TAG Clause" for more information.
- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the `IDENTIFIED BY` clause to specify the password for the keystore that contains the key. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the `WITH BACKUP` clause, and optionally the `USING '`*`backup_identifier`*`'` clause, to create a backup of the keystore before the key is activated. Refer to "Notes on the WITH BACKUP Clause" for more information.

See Also:

Oracle Database Advanced Security Guide for more information on activating a master encryption key

set_key_tag

This clause lets you set the tag for the specified encryption key. The tag is an optional, user-defined descriptor for the key. If the key has no tag, then use this clause to create a tag. If the key already has a tag, then use this clause to replace the tag. You can view encryption key tags by querying the TAG column of the V$ENCRYPTION_KEYS view. The keystore must be open.

- For *tag*, specify an alphanumeric string. Enclose *tag* in single quotation marks.
- For *key_id*, specify the identifier of the encryption key. You can find the key identifier by querying the KEY_ID column of the V$ENCRYPTION_KEYS view.
- The FORCE KEYSTORE clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the IDENTIFIED BY clause to specify the password for the keystore that contains the key. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the WITH BACKUP clause, and optionally the USING '*backup_identifier*' clause, to create a backup of the keystore before you set the key tag. Refer to "Notes on the WITH BACKUP Clause" for more information.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on setting a key tag

export_keys

Use this clause to export one or more encryption keys from a password-based software keystore into a file. The keystore must be open. Each encryption key is exported together with its key identifier and key attributes. The exported keys are protected in the file with a password (secret). You can subsequently import one or more of the keys into a password-based software keystore by using the *import_keys* clause.

- The ENCRYPTION keyword is optional and is provided for semantic clarity.
- Specify *secret* to set the password (secret) that protects the keys in the file. The secret is an alphanumeric string. You can optionally enclose the secret in double quotation marks. Quoted and nonquoted secrets are case sensitive.
- For *filename*, specify the full path name of the file to which the keys are to be exported. Enclose *filename* in single quotation marks.
- The FORCE KEYSTORE clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the IDENTIFIED BY clause to specify the password for the keystore that contains the keys you want to export. Refer to "Notes on the WITH BACKUP Clause" for more information.
- Use the WITH IDENTIFIER IN clause to specify one or more encryption keys that you would like to export using one of the following methods:
  - Use *key_id* to specify the identifier of the encryption key you would like to export. You can specify more than one *key_id* in a comma-separated list. You can find key identifiers by querying the KEY_ID column of the V$ENCRYPTION_KEYS view.
  - Use *subquery* to specify a query that returns a list of key identifiers for the encryption keys you would like to export. For example, the following *subquery* returns the key identifiers for all encryption keys in the database whose tags begin with the string mytag:

```
SELECT KEY_ID FROM V$ENCRYPTION_KEYS WHERE TAG LIKE 'mytag%'
```

Be aware that Oracle Database executes *subquery* within the current user's rights and not with definer's rights.

– If you omit the WITH IDENTIFIER IN clause, then all encryption keys in the database are exported.

Restriction on the WITH IDENTIFIER IN Clause

In a multitenant environment, you cannot specify WITH IDENTIFIER IN when exporting keys from a PDB. This ensures that all of the keys in the PDB are exported, along with metadata about the active encryption key. If you subsequently clone the PDB, or unplug and plug in the PDB, then you can use the export file to import the keys into the cloned or newly plugged-in PDB and preserve information about the active encryption key.

See Also:

Oracle Database Advanced Security Guide for more information on exporting encryption keys

import_keys

Use this clause to import one or more encryption keys from a file into a password-based software keystore. The keystore must be open. Each encryption key is imported together with its key identifier and key attributes. The keys must have been previously exported to the file by using the *export_keys* clause. You cannot re-import keys that have already been imported into the keystore.

- The ENCRYPTION keyword is optional and is provided for semantic clarity.
- For *secret*, specify the password (secret) that protects the keys in the file. The secret is an alphanumeric string. You can optionally enclose the secret in double quotation marks. Quoted and nonquoted secrets are case sensitive.
- For *filename*, specify the full path name of the file from which the keys are to be imported. Enclose *filename* in single quotation marks.
- The FORCE KEYSTORE clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the IDENTIFIED BY clause to specify the password for the keystore into which you want to import the keys. Refer to "Notes on the WITH BACKUP Clause" for more information.
- Specify the WITH BACKUP clause, and optionally the USING '*backup_identifier*' clause, to create a backup of the keystore before the keys are imported. Refer to "Notes on the WITH BACKUP Clause" for more information.

See Also:

Oracle Database Advanced Security Guide for more information on importing encryption keys

migrate_key

Use this clause to migrate from a password-based software keystore to a hardware keystore. This clause decrypts existing table encryption keys and tablespace encryption keys with the master encryption key in the software keystore and then re-encrypts them with the newly created master encryption key in the hardware keystore.

> **Note:**
>
> The use of this clause is only one step in a series of steps for migrating from a password-based software keystore to a hardware keystore. Refer to *Oracle Database Advanced Security Guide* for the complete set of steps before you use this clause.

- The `ENCRYPTION` keyword is optional and is provided for semantic clarity.
- For *`HSM_auth_string`*, specify the hardware keystore password. Refer to "Notes on Specifying Keystore Passwords" for more information.
- The `FORCE KEYSTORE` clause enables this operation even if the keystores are closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- For *`software_keystore_password`*., specify the password-based software keystore password. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the `WITH BACKUP` clause, and optionally the `USING '`*`backup_identifier`*`'` clause, to create a backup of the keystore before the migration occurs. Refer to "Notes on the WITH BACKUP Clause" for more information.

reverse_migrate_key

Use this clause to migrate from a hardware keystore to a password-based software keystore. This clause decrypts existing table encryption keys and tablespace encryption keys with the master encryption key in the hardware keystore and then re-encrypts them with the newly created master encryption key in the password-based software keystore.

> **Note:**
>
> The use of this clause is only one step in a series of steps for migrating from a hardware keystore to a password-based software keystore. Refer to *Oracle Database Advanced Security Guide* for the complete set of steps before you use this clause.

- The `ENCRYPTION` keyword is optional and is provided for semantic clarity.
- For *`software_keystore_password`*., specify the password-based software keystore password. Refer to "Notes on Specifying Keystore Passwords" for more information.
- The `FORCE KEYSTORE` clause enables this operation even if the keystores are closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.

- For *HSM_auth_string*, specify the hardware keystore password. Refer to "Notes on Specifying Keystore Passwords" for more information.

secret_management_clauses

Use these clauses to add, update, and delete secrets in a password-based software keystores or a hardware keystore.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on adding, updating, and deleting secrets

add_update_secret

This clause lets you add a secret to a keystore or update an existing secret in a keystore. The keystore must be open.

- Specify `ADD` to add a secret to a keystore.
- Specify `UPDATE` to update an existing secret in a keystore.
- For *secret*, specify the secret to be added or updated. The secret is an alphanumeric string. Enclose the secret in single quotation marks.
- For *client_identifier*, specify an alphanumeric string used to identify the secret. Enclose *client_identifier* in single quotation marks. This value is case-sensitive. You can enter any of the following fixed values:
  - `TDE_WALLET` if the keystore was configured as `FILE`
  - `OKV_WALLET` if the keystore is for an Oracle Key Vault HSM
  - `HSM_WALLET` if the keystore is for a third-party HSM
- Specify the optional `USING TAG` clause to associate a tag to *secret*. The *tag* is an optional, user-defined descriptor for the secret. Enclose the tag in single quotation marks. You can view secret tags by querying the `SECRET_TAG` column of the `V$CLIENT_SECRETS` view.
- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the `IDENTIFIED BY` clause to specify the password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the `WITH BACKUP` clause, and optionally the `USING '`*backup_identifier*`'` clause, to create a backup of the keystore before adding or updating the secret in a password-based software keystore. Refer to "Notes on the WITH BACKUP Clause" for more information.

delete_secret

This clause lets you delete a secret from a keystore. The keystore must be open.

- For *client_identifier*, specify an alphanumeric string used to identify the secret. Enclose *client_identifier* in single quotation marks. You can view client identifiers by querying the `CLIENT` column of the `V$CLIENT_SECRETS` view.

- The `FORCE KEYSTORE` clause enables this operation even if the keystore is closed. Refer to "Notes on the FORCE KEYSTORE Clause" for more information.
- Use the `IDENTIFIED BY` clause to specify the password for the keystore. Refer to "Notes on Specifying Keystore Passwords" for more information.
- Specify the `WITH BACKUP` clause, and optionally the `USING 'backup_identifier'` clause, to create a backup of the keystore before deleting the secret from a password-based software keystore. Refer to "Notes on the WITH BACKUP Clause" for more information.

Notes on the USING TAG Clause

Many `ADMINISTER KEY MANAGEMENT` operations include the `USING TAG` clause, which lets you associate a tag to an encryption key. The *tag* is an optional, user-defined descriptor for the key. It is a character string enclosed in single quotation marks.

You can view encryption key tags by querying the `TAG` column of the `V$ENCRYPTION_KEYS` view.

Notes on the FORCE KEYSTORE Clause

The `FORCE KEYSTORE` clause enables a keystore operation even if the keystore is closed. The behavior of this clause depends on whether you are connected to a non-CDB, a CDB root, or a PDB:

- When you are connected to a non-CDB:
  - If the password-based software or hardware keystore is closed, then the database opens the password-based software or hardware keystore while the operation is performed and leaves it open, and then updates the auto-login keystore, if one exists, with the new information.
  - If the auto-login keystore is open, then the database opens the password-based software or hardware keystore temporarily while the operation is performed and updates the auto-login keystore with the new information, without switching out the auto-login keystore.
  - If the password-based software or hardware keystore is open, then the `FORCE KEYSTORE` clause is not necessary and has no effect.
- When you are connected to the CDB root:
  - To perform an operation on the CDB root keystore (`CONTAINER=CURRENT`), the CDB root keystore must be open. Therefore, the behavior described for a non-CDB applies to the CDB root.
  - To perform an operation on the CDB root keystore and all PDB keystores (`CONTAINER=ALL`), the CDB root keystore and all PDB keystores must be open. Therefore, the behavior described for a non-CDB applies to the CDB root and each PDB.
- When you are connected to a PDB:
  - To perform an operation on a PDB keystore, the CDB root keystore and the keystore for that PDB must be open. Therefore, the behavior described for a non-CDB applies to the CDB root and that PDB.

Notes on Specifying Keystore Passwords

Specify keystore passwords as follows:

- For a password-based software keystore, specify the password as a character string. You can optionally enclose the password in double quotation marks. Quoted and nonquoted passwords are case sensitive. Keystore passwords adhere to the same rules as database user passwords. Refer to the BY *password* clause of CREATE USER for the complete details.

- For a hardware keystore, specify the password as a string of the form "*user_id*:*password*" where:
  - *user_id* is the user ID created for the database using the HSM management interface
  - *password* is the password created for the user ID using the HSM management interface

  Enclose the *user_id*:*password* string in double quotation marks (" ") and separate *user_id* and *password* with a colon (:).
- If you specify EXTERNAL STORE, then the database uses the keystore password stored in the external store to perform the operation. This feature enables you to store the password in a separate location where it can be centrally managed and accessed. To use this functionality, you must first set the EXTERNAL_KEYSTORE_CREDENTIAL_LOCATION initialization parameter to a location where the keystore password will be stored. Refer to *Oracle Database Advanced Security Guide* for more information on configuring an external store for a keystore password.

Notes on the WITH BACKUP Clause

Many ADMINISTER KEY MANAGEMENT operations include the WITH BACKUP clause. This clause applies only to password-based software keystores. It indicates that the keystore must be backed up before the operation is performed. Therefore, you must either specify the WITH BACKUP clause when performing the operation, or issue the ADMINISTER KEY MANAGEMENT *backup_clause* statement immediately *before* performing the operation.

When you specify the WITH BACKUP clause, Oracle Database creates a backup file with a name of the form ewallet_*timestamp*.p12, where *timestamp* is the file creation timestamp in UTC format. The backup file is created in the same directory as the keystore you are backing up.

The optional USING '*backup_identifier*' clause lets you specify a backup identifier, which is added to the backup file name. For example, if you specify a backup identifier of 'Backup1', then Oracle Database creates a backup file with a name of the form ewallet_*timestamp*_Backup1.p12.

Examples

Creating a Keystore: Examples

The following statement creates a password-based software keystore in directory /etc/ORACLE/WALLETS/orcl:

```
ADMINISTER KEY MANAGEMENT
  CREATE KEYSTORE '/etc/ORACLE/WALLETS/orcl'
  IDENTIFIED BY password;
```

The following statement creates an auto-login software keystore from the keystore created in the previous statement:

```
ADMINISTER KEY MANAGEMENT
  CREATE AUTO_LOGIN KEYSTORE FROM KEYSTORE '/etc/ORACLE/WALLETS/orcl'
  IDENTIFIED BY password;
```

Opening a Keystore: Examples

The following statement opens a password-based software keystore:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE OPEN
  IDENTIFIED BY password;
```

If you are connected to a CDB, then the following statement opens a password-based software keystore in the current container:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE OPEN
  IDENTIFIED BY password
  CONTAINER = CURRENT;
```

The following statement opens a hardware keystore:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE OPEN
  IDENTIFIED BY "user_id:password";
```

The following statement opens a keystore whose password is stored in the external store:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE OPEN
  IDENTIFIED BY EXTERNAL STORE;
```

Closing a Keystore: Examples

The following statement closes a password-based software keystore:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE CLOSE
  IDENTIFIED BY password;
```

The following statement closes an auto-login software keystore:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE CLOSE;
```

The following statement closes a hardware keystore:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE CLOSE
  IDENTIFIED BY "user_id:password";
```

The following statement closes a keystore whose password is stored in the external store:

```
ADMINISTER KEY MANAGEMENT
  SET KEYSTORE CLOSE
  IDENTIFIED BY EXTERNAL STORE;
```

Backing Up a Keystore: Example

The following statement creates a backup of a password-based software keystore. The backup is stored in directory `/etc/ORACLE/KEYSTORE/DB1` and the backup file name contains the tag `hr.emp_keystore`.

```
ADMINISTER KEY MANAGEMENT
  BACKUP KEYSTORE USING 'hr.emp_keystore'
  IDENTIFIED BY password
  TO '/etc/ORACLE/KEYSTORE/DB1/';
```

Changing a Keystore Password: Example

The following statement changes the password for a password-based software keystore. It also creates a backup of the keystore, with the tag `pwd_change`, before changing the password.

```
ADMINISTER KEY MANAGEMENT
  ALTER KEYSTORE PASSWORD IDENTIFIED BY old_password
  SET new_password WITH BACKUP USING 'pwd_change';
```

Merging Two Keystores Into a New Keystore: Example

The following statement merges an auto-login software keystore with a password-based software keystore to create a new password-based software keystore at a new location:

```
ADMINISTER KEY MANAGEMENT
  MERGE KEYSTORE '/etc/ORACLE/KEYSTORE/DB1'
  AND KEYSTORE '/etc/ORACLE/KEYSTORE/DB2'
    IDENTIFIED BY existing_keystore_password
  INTO NEW KEYSTORE '/etc/ORACLE/KEYSTORE/DB3'
    IDENTIFIED BY new_keystore_password;
```

Merging a Keystore Into an Existing Keystore: Example

The following statement merges an auto-login software keystore into a password-based software keystore. It also creates a backup of the password-based software keystore before performing the merge.

```
ADMINISTER KEY MANAGEMENT
  MERGE KEYSTORE '/etc/ORACLE/KEYSTORE/DB1'
  INTO EXISTING KEYSTORE '/etc/ORACLE/KEYSTORE/DB2'
    IDENTIFIED BY existing_keystore_password
  WITH BACKUP;
```

Creating and Activating a Master Encryption Key: Examples

The following statement creates and activates a master encryption key in a password-based software keystore. It encrypts the key using the `SEED128` algorithm. It also creates a backup of the keystore before creating the new master encryption key.

```
ADMINISTER KEY MANAGEMENT
  SET KEY USING ALGORITHM 'SEED128'
  IDENTIFIED BY password
  WITH BACKUP;
```

The following statement creates a master encryption key in a password-based software keystore, but does not activate the key. It also creates a backup of the keystore before creating the new master encryption key.

```
ADMINISTER KEY MANAGEMENT
  CREATE KEY USING TAG 'mykey1'
  IDENTIFIED BY password
  WITH BACKUP;
```

The following query displays the key identifier for the master encryption key that was created in the previous statement:

```
SELECT TAG, KEY_ID
  FROM V$ENCRYPTION_KEYS
  WHERE TAG = 'mykey1';

TAG     KEY_ID
---     ----------------------------------------------------
mykey1  ARgEtzPxpE/Nv8WdPu8LJJUAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

The following statement activates the master encryption key that was queried in the previous statement. It also creates a backup of the keystore before activating the new master encryption key.

```
ADMINISTER KEY MANAGEMENT
  USE KEY 'ARgEtzPxpE/Nv8WdPu8LJJUAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
  IDENTIFIED BY password
  WITH BACKUP;
```

Setting a Key Tag: Example

This example assumes that the keystore is closed. The following statement temporarily opens the keystore and changes the tag to `mykey2` for the master encryption key that was activated in the previous example. It also creates a backup of the keystore before changing the tag.

```
ADMINISTER KEY MANAGEMENT
  SET TAG 'mykey2' FOR 'ARgEtzPxpE/Nv8WdPu8LJJUAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
  FORCE KEYSTORE
  IDENTIFIED BY password
  WITH BACKUP;
```

Exporting Keys: Examples

The following statement exports two master encryption keys from a password-based software keystore to file `/etc/TDE/export.exp`. The statement encrypts the master encryption keys in the file using the secret `my_secret`. The identifiers of the master encryption keys to be exported are provided as a comma-separated list.

```
ADMINISTER KEY MANAGEMENT
  EXPORT KEYS WITH SECRET "my_secret"
  TO '/etc/TDE/export.exp'
  IDENTIFIED BY password
  WITH IDENTIFIER IN 'AdoxnJ0uH08cv7xkz83ovwsAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',
                     'AW5z3CoyKE/yv3cNT5CWCXUAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
```

The following statement exports master encryption keys from a password-based software keystore to file `/etc/TDE/export.exp`. Only the keys whose tags are `mytag1` or `mytag2` are exported. The master encryption keys in the file are encrypted using the secret `my_secret`. The key identifiers are found by querying the `V$ENCRYPTION_KEYS` view.

```
ADMINISTER KEY MANAGEMENT
  EXPORT KEYS WITH SECRET "my_secret"
  TO '/etc/TDE/export.exp'
  IDENTIFIED BY password
  WITH IDENTIFIER IN
    (SELECT KEY_ID FROM V$ENCRYPTION_KEYS WHERE TAG IN ('mytag1', 'mytag2'));
```

The following statement exports all master encryption keys of the database to file `/etc/TDE/export.exp`. The master encryption keys in the file are encrypted using the secret `my_secret`.

```
ADMINISTER KEY MANAGEMENT
  EXPORT KEYS WITH SECRET "my_secret"
  TO '/etc/TDE/export.exp'
  IDENTIFIED BY password;
```

In a multitenant environment, the following statements exports all master encryption keys of the PDB `salespdb`, along with metadata, to file `/etc/TDE/salespdb.exp`. The master encryption keys in the file are encrypted using the secret `my_secret`. If the PDB is subsequently cloned, or unplugged and plugged back in, then the export file created by this statement can be used to import the keys into the cloned or newly plugged-in PDB.

```
ALTER SESSION SET CONTAINER = salespdb;
ADMINISTER KEY MANAGEMENT
  EXPORT KEYS WITH SECRET "my_secret"
  TO '/etc/TDE/salespdb.exp'
  IDENTIFIED BY password;
```

Importing Keys: Example

The following statement imports the master encryption keys, encrypted with secret `my_secret`, from file `/etc/TDE/export.exp` to a password-based software keystore. It also creates a backup of the password-based software keystore before importing the keys.

```
ADMINISTER KEY MANAGEMENT
  IMPORT KEYS WITH SECRET "my_secret"
  FROM '/etc/TDE/export.exp'
  IDENTIFIED BY password
  WITH BACKUP;
```

Migrating a Keystore: Example

The following statement migrates from a password-based software keystore to a hardware keystore. It also creates a backup of the password-based software keystore before performing the migration.

```
ADMINISTER KEY MANAGEMENT
  SET ENCRYPTION KEY IDENTIFIED BY "user_id:password"
  MIGRATE USING software_keystore_password
  WITH BACKUP;
```

Reverse Migrating a Keystore: Example

The following statement reverse migrates from a hardware keystore to a password-based software keystore:

```
ADMINISTER KEY MANAGEMENT
  SET ENCRYPTION KEY IDENTIFIED BY software_keystore_password
  REVERSE MIGRATE USING "user_id:password";
```

Adding a Secret to a Keystore: Examples

The following statement adds secret `secret1`, with the tag `My first secret`, for client `client1` to a password-based software keystore. It also creates a backup of the password-based software keystore before adding the secret.

```
ADMINISTER KEY MANAGEMENT
  ADD SECRET 'secret1' FOR CLIENT 'client1'
  USING TAG 'My first secret'
  IDENTIFIED BY password
  WITH BACKUP;
```

The following statement adds a similar secret to a hardware keystore:

```
ADMINISTER KEY MANAGEMENT
  ADD SECRET 'secret2' FOR CLIENT 'client2'
  USING TAG 'My second secret'
  IDENTIFIED BY "user_id:password";
```

Updating a Secret in a Keystore: Examples

The following statement updates the secret that was created in the previous example in a password-based software keystore. It also creates a backup of the password-based software keystore before updating the secret.

```
ADMINISTER KEY MANAGEMENT
  UPDATE SECRET 'secret1' FOR CLIENT 'client1'
  USING TAG 'New Tag 1'
  IDENTIFIED BY password
  WITH BACKUP;
```

The following statement updates the secret that was created in the previous example in a hardware keystore:

```
ADMINISTER KEY MANAGEMENT
  UPDATE SECRET 'secret2' FOR CLIENT 'client2'
  USING TAG 'New Tag 2'
  IDENTIFIED BY "user_id:password";
```

Deleting a Secret from a Keystore: Examples

The following statement deletes the secret that was updated in the previous example from a password-based software keystore. It also creates a backup of the password-based software keystore before deleting the secret.

```
ADMINISTER KEY MANAGEMENT
  DELETE SECRET FOR CLIENT 'client1'
  IDENTIFIED BY password
  WITH BACKUP;
```

The following statement deletes the secret that was updated in the previous example from a hardware keystore:

```
ADMINISTER KEY MANAGEMENT
  DELETE SECRET FOR CLIENT 'client2'
  IDENTIFIED BY "user_id:password";
```

ALTER ANALYTIC VIEW

Purpose

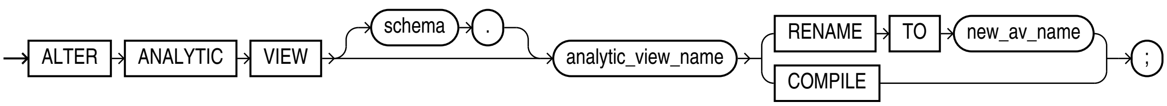
Use the `ALTER ANALYTIC VIEW` statement to rename or compile an analytic view. For other alterations, use `CREATE OR REPLACE ANALYTIC VIEW`.

Prerequisites

To alter an analytic view in your own schema, you must have the ALTER ANALYTIC VIEW system privilege. To alter an analytic view in another user's schema, you must have the ALTER ANY ANALYTIC VIEW system privilege or ALTER ANY TABLE granted on the analytic view.

Syntax

***alter_analytic_view*::=**



Semantics

schema

Specify the schema in which the analytic view exists. If you do not specify a schema, then Oracle Database looks for the analytic view in your own schema.

analytic_view_name

Specify the name of the analytic view.

RENAME TO

Specify RENAME TO to change the name of the analytic view. For *new_av_name*, specify a new name for the analytic view.

COMPILE

Specify COMPILE to compile the analytic view.

Example

The following statement changes the name of an analytic view:

```
ALTER ANALYTIC VIEW sales_av RENAME TO mysales_av;
```

ALTER ATTRIBUTE DIMENSION

Purpose

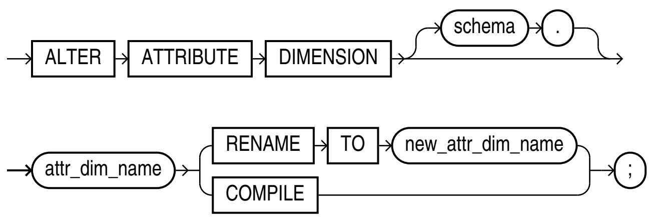
Use the ALTER ATTRIBUTE DIMENSION statement to rename or compile an attribute dimension. For other alterations, use CREATE OR REPLACE ATTRIBUTE DIMENSION.

Prerequisites

To alter an attribute dimension in your own schema, you must have the ALTER ATTRIBUTE DIMENSION system privilege. To alter an attribute dimension in another user's schema, you must have the ALTER ANY ATTRIBUTE DIMENSION system privilege or have been granted ALTER on the attribute dimension directly.

Syntax

***alter_attribute_dimension*::=**



Semantics

schema

Specify the schema in which the attribute dimension exists. If you do not specify a schema, then Oracle Database looks for the attribute dimension in your own schema.

attr_dim_name

Specify the name of the attribute dimension.

RENAME TO

Specify RENAME TO to change the name of the attribute dimension. For *new_attr_dim_name*, specify a new name for the attribute dimension.

COMPILE

Specify COMPILE to compile the attribute dimension.

Example

The following statement changes the name of an attribute dimension:

```
ALTER ATTRIBUTE DIMENSION product_attr_dim RENAME TO my_product_attr_dim;
```

ALTER AUDIT POLICY (Unified Auditing)

This section describes the ALTER AUDIT POLICY statement for **unified auditing**. This type of auditing is new beginning with Oracle Database 12*c* and provides a full set of enhanced auditing features. Refer to *Oracle Database Security Guide* for more information on unified auditing.

Purpose

Use the ALTER AUDIT POLICY statement to modify a unified audit policy.

> **See Also:**
>
> - CREATE AUDIT POLICY (Unified Auditing)
> - DROP AUDIT POLICY (Unified Auditing)
> - AUDIT (Unified Auditing)
> - NOAUDIT (Unified Auditing)

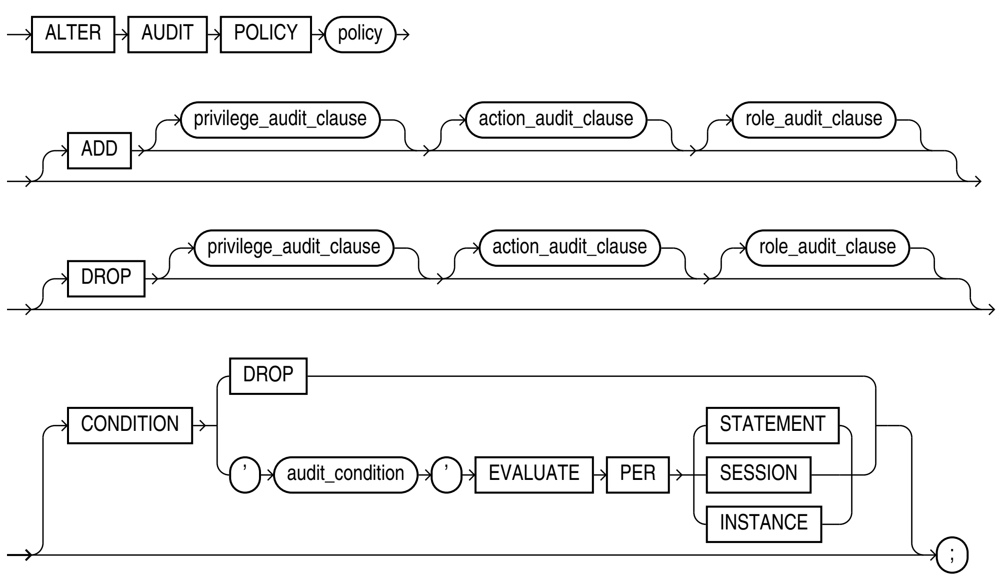
Prerequisites

You must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.

If you are connected to a multitenant container database (CDB), then to modify a common unified audit policy, the current container must be the root and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role. To modify a local unified audit policy, the current container must be the container in which the audit policy was created and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the container.

Syntax

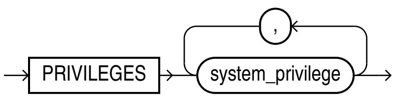
***alter_audit_policy*::=**



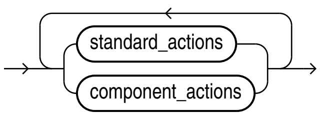
> **Note:**
>
> If you specify the `ADD` or `DROP` clause, then you must specify at least one of the clauses *`privilege_audit_clause`*, *`action_audit_clause`*, or *`role_audit_clause`*.

(*privilege_audit_clause*::=, *action_audit_clause*::=, *role_audit_clause*::=)

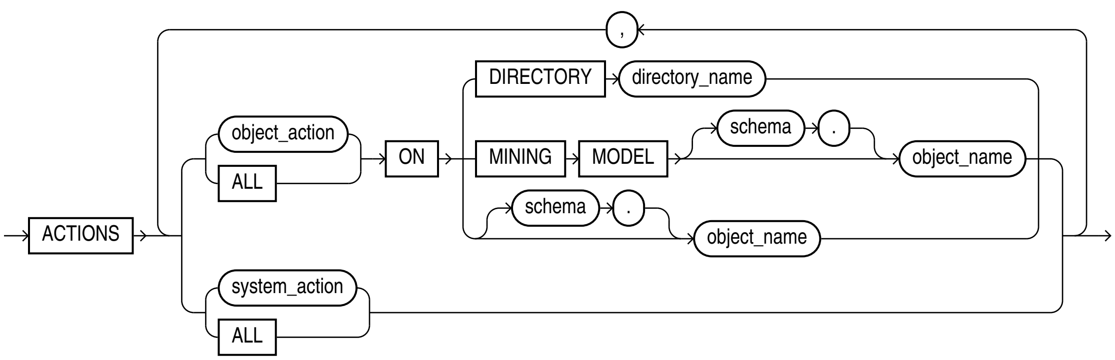
***privilege_audit_clause*::=**



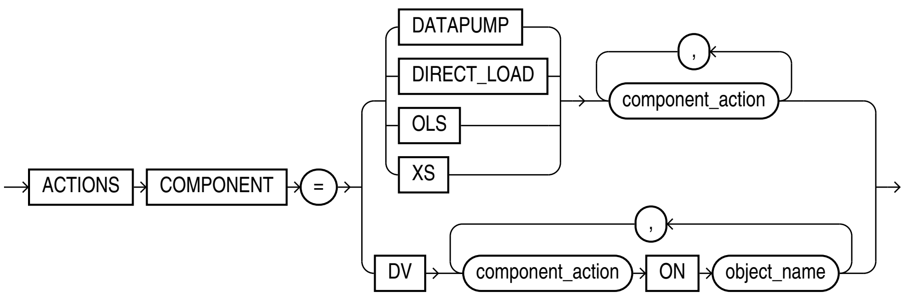
***action_audit_clause*::=**



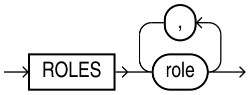
***standard_actions*::=**



***component_actions*::=**



***role_audit_clause*::=**



Semantics

policy

Specify the name of the unified audit policy to be modified. The policy must have been created using the `CREATE AUDIT POLICY` statement. You can find descriptions of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view.

> **See Also:**
>
> - CREATE AUDIT POLICY (Unified Auditing)
> - *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES` view

ADD | DROP

Use the `ADD` clause to add privileges to be audited to *`policy`*.

Use the `DROP` clause to remove privileges to be audited from *`policy`*.

Refer to *privilege_audit_clause*, *action_audit_clause*, and *role_audit_clause* of `CREATE AUDIT POLICY` for the full semantics of these clauses.

CONDITION

Use this clause to drop, add, or replace the audit condition for *`policy`*.

Specify `DROP` to drop the audit condition from *`policy`*.

Specify `'`*`audit_condition`*`'` ... to add or replace the audit condition for *`policy`*.

Refer to audit_condition, EVALUATE PER STATEMENT, EVALUATE PER SESSION, and EVALUATE PER INSTANCE of `CREATE AUDIT POLICY` for the full semantics of these clauses.

Examples

The following examples modify unified audit policies that were created in the `CREATE AUDIT POLICY` "Examples".

Adding Privileges, Actions, and Roles to a Unified Audit Policy: Examples

The following statement adds the system privileges `CREATE ANY TABLE` and `DROP ANY TABLE` to unified audit policy `dml_pol`:

```
ALTER AUDIT POLICY dml_pol
  ADD PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE;
```

The following statement adds the system actions `CREATE JAVA`, `ALTER JAVA`, and `DROP JAVA` to unified audit policy `java_pol`:

```
ALTER AUDIT POLICY java_pol
  ADD ACTIONS CREATE JAVA, ALTER JAVA, DROP JAVA;
```

The following statement adds the role `dba` to unified audit policy `table_pol`:

```
ALTER AUDIT POLICY table_pol
  ADD ROLES dba;
```

The following statement adds multiple system privileges, actions, and roles to unified audit policy `security_pol`:

```
ALTER AUDIT POLICY security_pol
  ADD PRIVILEGES CREATE ANY LIBRARY, DROP ANY LIBRARY
      ACTIONS DELETE on hr.employees,
              INSERT on hr.employees,
              UPDATE on hr.employees,
              ALL on hr.departments
      ROLES dba, connect;
```

Dropping Privileges, Actions, and Roles from a Unified Audit Policy: Examples

The following statement drops the system privilege `CREATE ANY TABLE` from unified audit policy `table_pol`:

```
ALTER AUDIT POLICY table_pol
  DROP PRIVILEGES CREATE ANY TABLE;
```

The following statement drops the `INSERT` and `UPDATE` actions on `hr.employees` from unified audit policy `dml_pol`:

```
ALTER AUDIT POLICY dml_pol
  DROP ACTIONS INSERT on hr.employees,
               UPDATE on hr.employees;
```

The following statement drops the role `java_deploy` from unified audit policy `java_pol`:

```
ALTER AUDIT POLICY java_pol
  DROP ROLES java_deploy;
```

The following statement drops a system privilege, an action, and a role from unified audit policy `hr_admin_pol`:

```
ALTER AUDIT POLICY hr_admin_pol
  DROP PRIVILEGES CREATE ANY TABLE
       ACTIONS LOCK TABLE
       ROLES audit_viewer;
```

Adding and Dropping Actions for a Unified Audit Policy: Example

The following statement adds `EXPORT` actions for Oracle Data Pump to unified audit policy `dp_actions_pol` and drops `IMPORT` actions for Oracle Data Pump:

```
ALTER AUDIT POLICY dp_actions_pol
  ADD ACTIONS COMPONENT = datapump EXPORT
  DROP ACTIONS COMPONENT = datapump IMPORT;
```

Dropping the Audit Condition from a Unified Audit Policy: Example

The following statement drops the audit condition from unified audit policy `order_updates_pol`:

```
ALTER AUDIT POLICY order_updates_pol
  CONDITION DROP;
```

Modifying the Audit Condition for a Unified Audit Policy: Example

The following statement modifies the audit condition for unified audit policy `emp_updates_pol` so that the policy is enforced only when the auditable statement is issued by a user whose UID is 102:

```
ALTER AUDIT POLICY emp_updates_pol
  CONDITION 'UID = 102'
  EVALUATE PER STATEMENT;
```

ALTER CLUSTER

Purpose

Use the `ALTER CLUSTER` statement to redefine storage and parallelism characteristics of a cluster.

> **Note:**
>
> You cannot use this statement to change the number or the name of columns in the cluster key, and you cannot change the tablespace in which the cluster is stored.

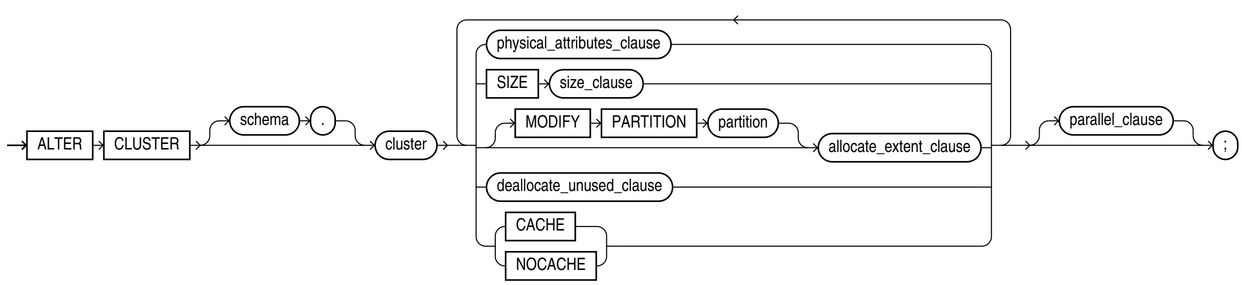
> **See Also:**
>
> CREATE CLUSTER for information on creating a cluster, DROP CLUSTER and DROP TABLE for information on removing tables from a cluster, and CREATE TABLE ... *physical_properties* for information on adding a table to a cluster

Prerequisites

The cluster must be in your own schema or you must have the `ALTER ANY CLUSTER` system privilege.

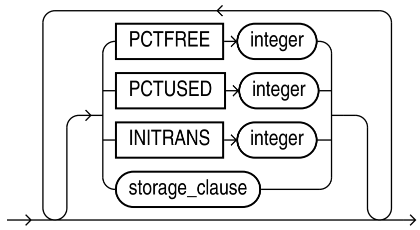
Syntax

***alter_cluster*::=**



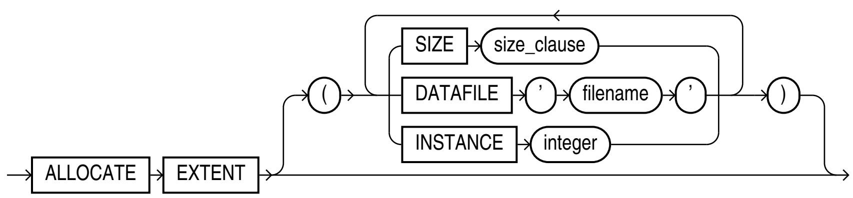
(*physical_attributes_clause*::, *size_clause*::=, MODIFY PARTITION, *allocate_extent_clause*::=, *deallocate_unused_clause*::=, *parallel_clause*::=)

***physical_attributes_clause*::**



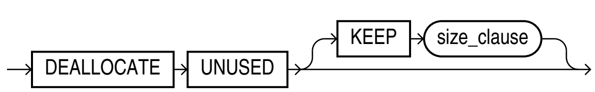
(*storage_clause*::=)

***allocate_extent_clause*::=**



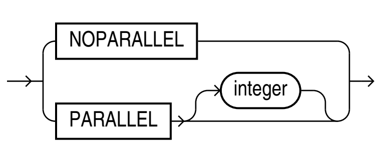
(*size_clause*::=)

***deallocate_unused_clause*::=**



(*size_clause*::=)

***parallel_clause*::=**



Semantics

schema

Specify the schema containing the cluster. If you omit *schema*, then Oracle Database assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be altered.

physical_attributes_clause

Use this clause to change the values of the PCTUSED, PCTFREE, and INITRANS parameters of the cluster.

Use the STORAGE clause to change the storage characteristics of the cluster.

> **See Also:**
>
> - *physical_attributes_clause* for information on the parameters
> - storage_clause for a full description of that clause

Restriction on Physical Attributes

You cannot change the values of the storage parameters INITIAL and MINEXTENTS for a cluster.

SIZE

integer

Use the SIZE clause to specify the number of cluster keys that will be stored in data blocks allocated to the cluster.

Restriction on SIZE

You can change the SIZE parameter only for an indexed cluster, not for a hash cluster.

> **See Also:**
>
> CREATE CLUSTER for a description of the SIZE parameter and "Modifying a Cluster: Example"

MODIFY PARTITION

Specify MODIFY PARTITION *partition allocate_extent_clause* to explicitly allocate a new extent for a cluster partition. This operation is valid only for range-partitioned hash clusters. For *partition*, specify the cluster partition name.

allocate_extent_clause

Specify `allocate_extent_clause` to explicitly allocate a new extent for a cluster. This operation is valid only for indexed clusters and nonpartitioned hash clusters.

When you explicitly allocate an extent with the `allocate_extent_clause`, Oracle Database does not evaluate the storage parameters of the cluster and determine a new size for the next extent to be allocated (as it does when you create a table). Therefore, specify `SIZE` if you do not want Oracle Database to use a default value.

> **See Also:**
>
> *allocate_extent_clause* for a full description of this clause

deallocate_unused_clause

Use the `deallocate_unused_clause` to explicitly deallocate unused space at the end of the cluster and make the freed space available for other segments.

> **See Also:**
>
> *deallocate_unused_clause* for a full description of this clause and "Deallocating Unused Space: Example"

parallel_clause

Specify the `parallel_clause` to change the default degree of parallelism for queries on the cluster.

> **See Also:**
>
> *parallel_clause* in the documentation on `CREATE TABLE` for complete information on this clause

Examples

The following examples modify the clusters that were created in the `CREATE CLUSTER` "Examples".

Modifying a Cluster: Example

The next statement alters the `personnel` cluster:

```
ALTER CLUSTER personnel
   SIZE 1024 CACHE;
```

Oracle Database allocates 1024 bytes for each cluster key value and enables the cache attribute. Assuming a data block size of 2 kilobytes, future data blocks within this cluster contain 2 cluster keys in each data block, or 2 kilobytes divided by 1024 bytes.

Deallocating Unused Space: Example

The following statement deallocates unused space from the `language` cluster, keeping 30 kilobytes of unused space for future use:

```
ALTER CLUSTER language
   DEALLOCATE UNUSED KEEP 30 K;
```

ALTER DATABASE

Purpose

Use the `ALTER DATABASE` statement to modify, maintain, or recover an existing database.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for examples of performing media recovery
> - *Oracle Data Guard Concepts and Administration* for additional information on using the `ALTER DATABASE` statement to maintain standby databases
> - CREATE DATABASE for information on creating a database

Prerequisites

You must have the `ALTER DATABASE` system privilege.

To specify the *`startup_clauses`*, you must also be connected `AS SYSDBA`, `AS SYSOPER`, `AS SYSBACKUP`, or `AS SYSDG`.

To specify the *`general_recovery`* clause, you must also have the `SYSDBA` or `SYSBACKUP` system privilege.

To specify the `DEFAULT EDITION` clause, you must also have the `USE` object privilege `WITH GRANT OPTION` on the specified edition.

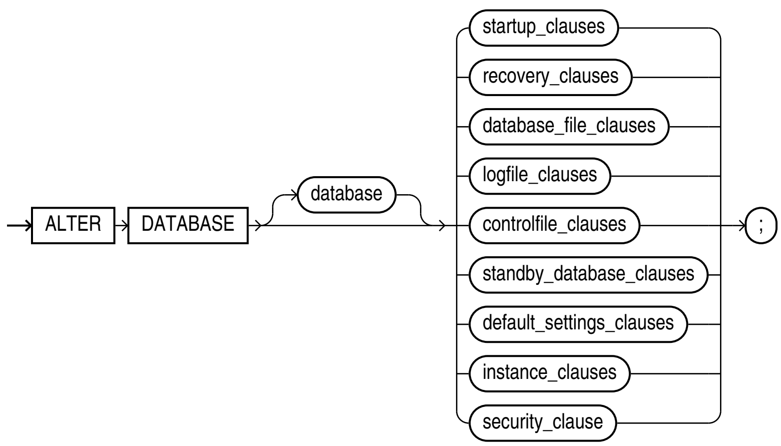
If you are connected to a multitenant container database (CDB):

- To modify the entire CDB, the current container must be the root and you must have the commonly granted `ALTER DATABASE` privilege.
- To modify a container, it must be the current container and you must have the `ALTER DATABASE` privilege, either granted commonly or granted locally in the container.

> **See Also:**
>
> "Notes on Using ALTER DATABASE in a CDB" to determine which clauses of `ALTER DATABASE` modify the entire CDB and which clauses modify only a container

Syntax

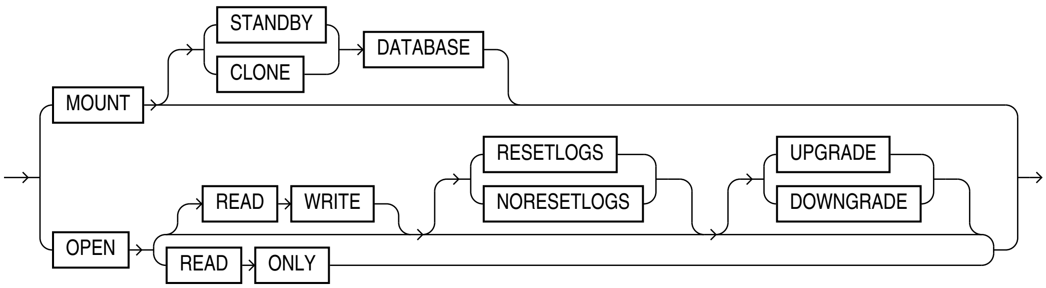
***alter_database*::=**



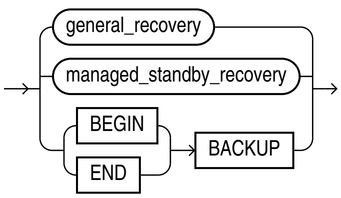
Groups of ALTER DATABASE syntax:

- ***startup_clauses*::=**
- ***recovery_clauses*::=**
- ***database_file_clauses*::=**
- ***logfile_clauses*::=**
- ***controlfile_clauses*::=**
- ***standby_database_clauses*::=**
- ***default_settings_clauses*::=**
- ***instance_clauses*::=**
- ***security_clause*::=**

***startup_clauses*::=**

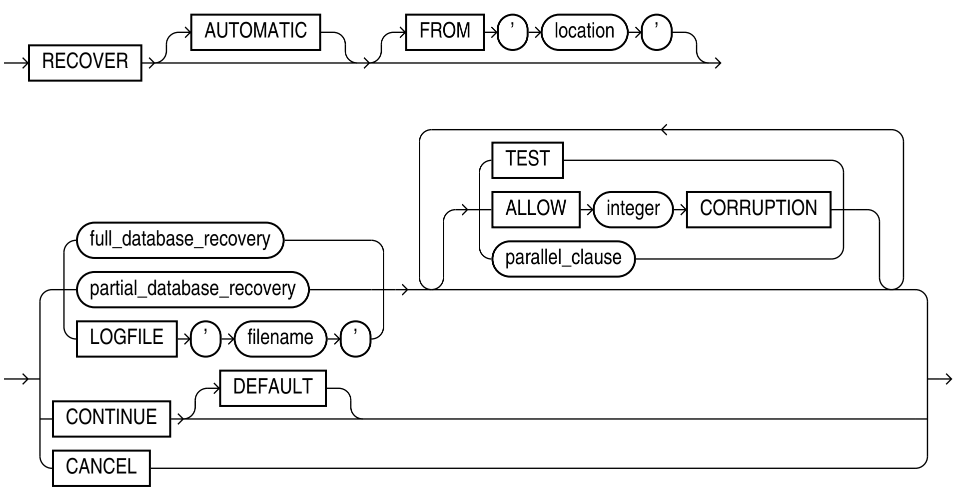


***recovery_clauses*::=**



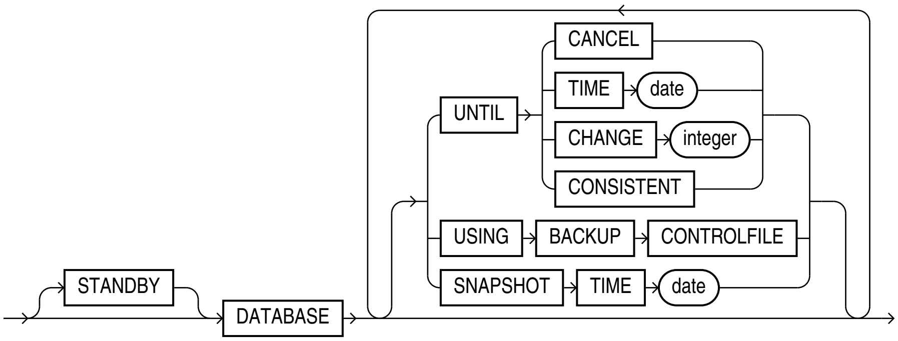
(*general_recovery*::=, *managed_standby_recovery*::=)

***general_recovery*::=**

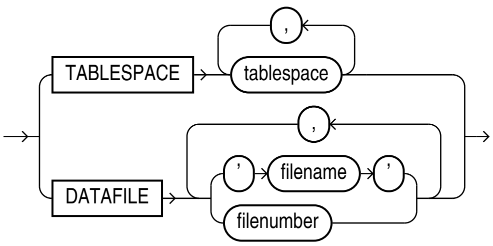


(*full_database_recovery*::=, *partial_database_recovery*::=, *parallel_clause*::=)

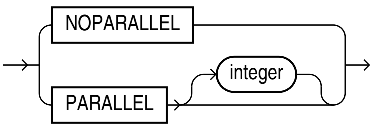
***full_database_recovery*::=**



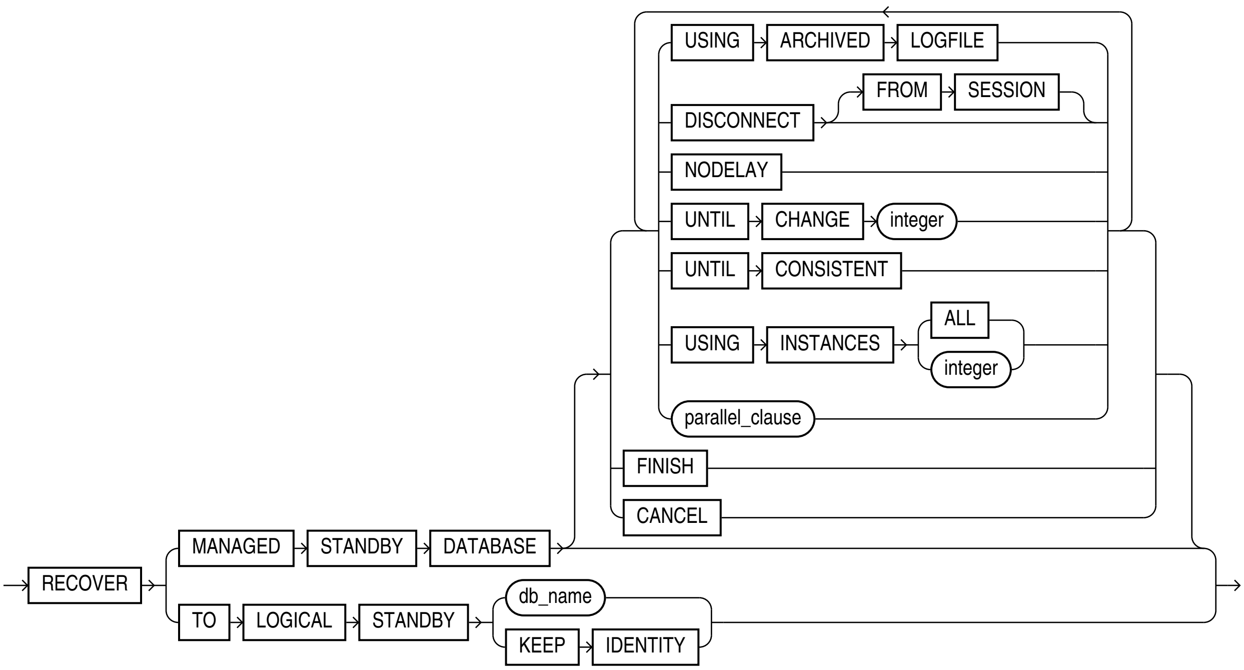
***partial_database_recovery*::=**



***parallel_clause*::=**



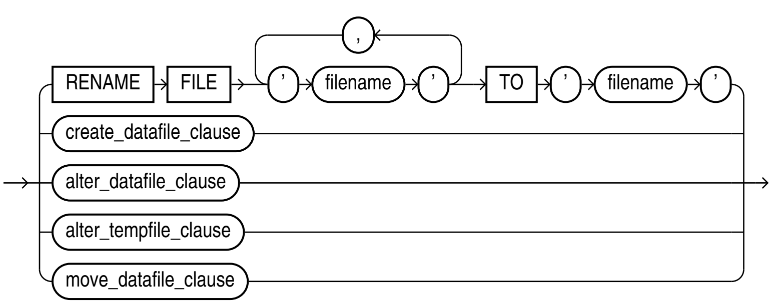
***managed_standby_recovery*::=**



(*parallel_clause::=*)

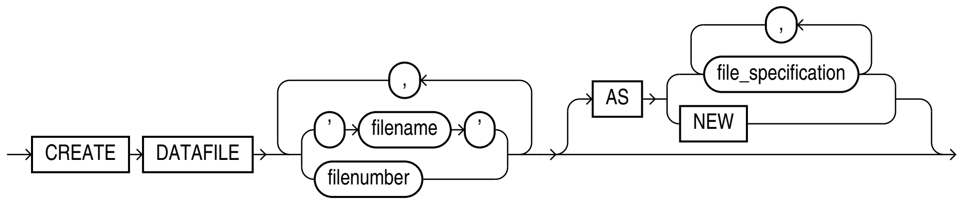
> **Note:**
>
> Several subclauses of `managed_standby_recovery` are no longer needed and have been deprecated. These clauses no longer appear in the syntax diagrams. Refer to the semantics of *managed_standby_recovery*.

***database_file_clauses*::=**



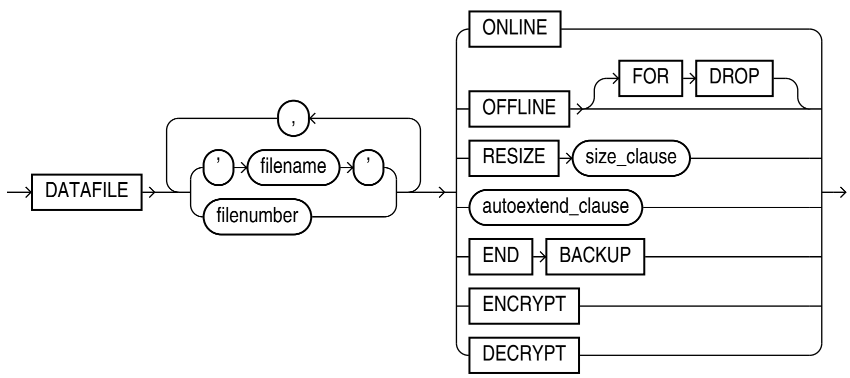
(*create_datafile_clause::=*, *alter_datafile_clause::=*, *alter_tempfile_clause::=*, *move_datafile_clause::=*)

***create_datafile_clause*::=**



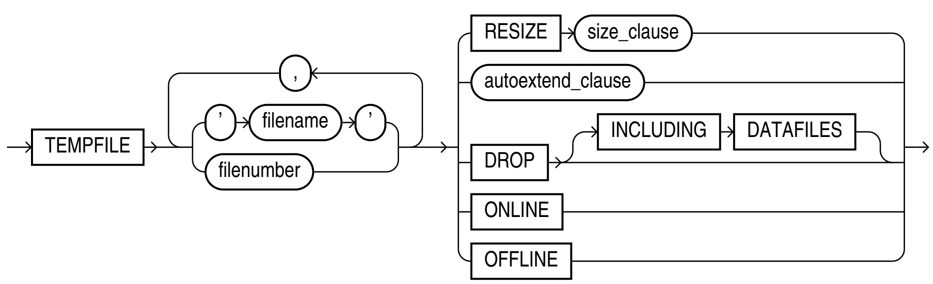
(*file_specification::=*)

***alter_datafile_clause*::=**



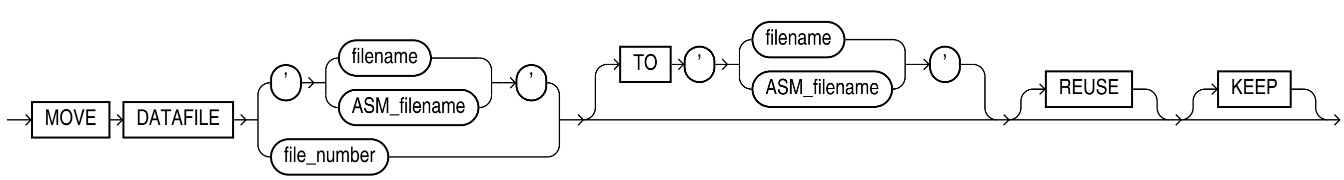
(*autoextend_clause::=*, *size_clause::=*)

***alter_tempfile_clause*::=**

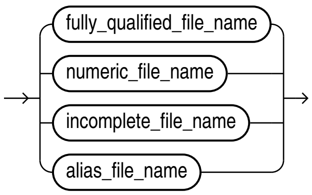


(*autoextend_clause::=*, *size_clause::=*)

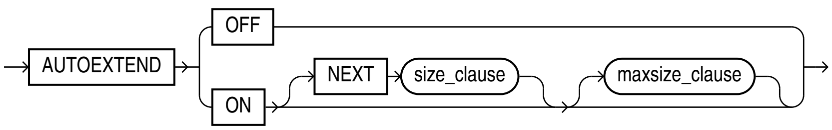
***move_datafile_clause*::=**



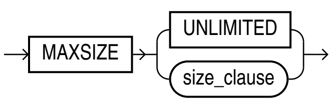
***ASM_filename*::=**



***autoextend_clause*::=**

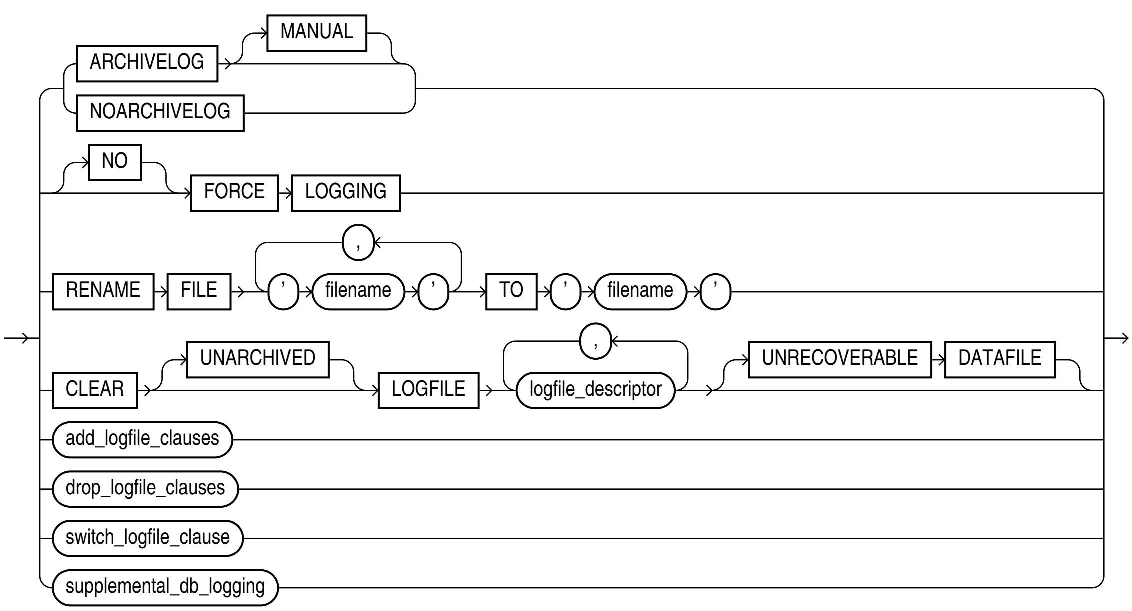


***maxsize_clause*::=**



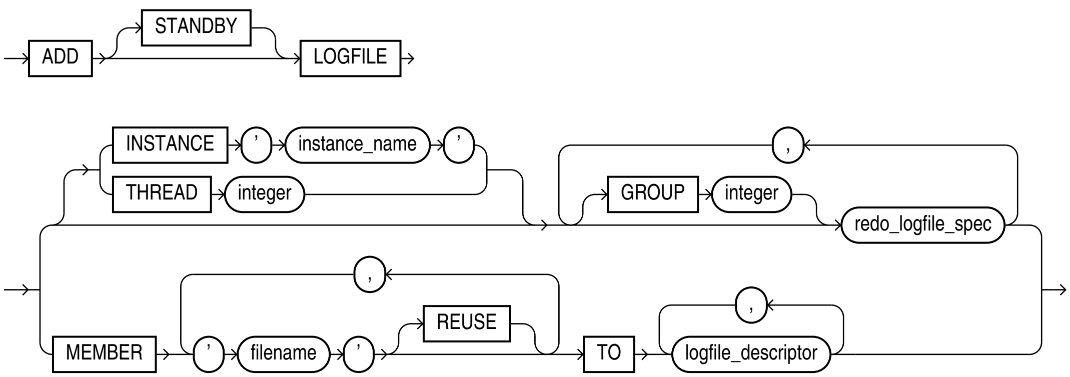
(*size_clause*::=)

***logfile_clauses*::=**



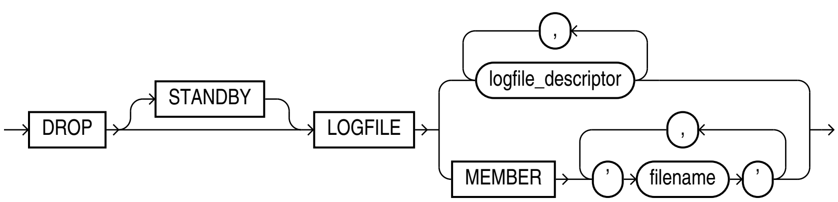
(*logfile_descriptor*::=, *add_logfile_clauses*::=, *drop_logfile_clauses*::=, *switch_logfile_clause*::=, *supplemental_db_logging*::=)

***add_logfile_clauses*::=**



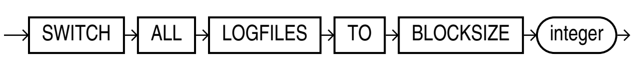
(*redo_log_file_spec*::=, *logfile_descriptor*::=)

***drop_logfile_clauses*::=**

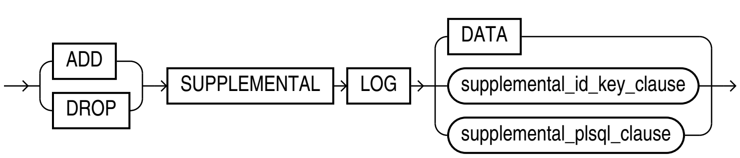


(*logfile_descriptor*::=)

***switch_logfile_clause*::=**

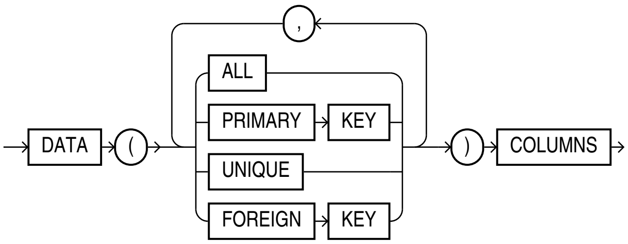


***supplemental_db_logging*::=**



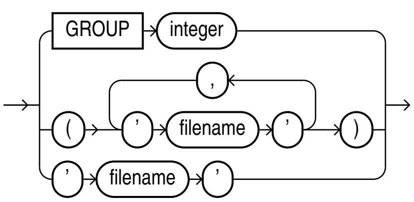
(*supplemental_id_key_clause*::=)

***supplemental_id_key_clause*::=**

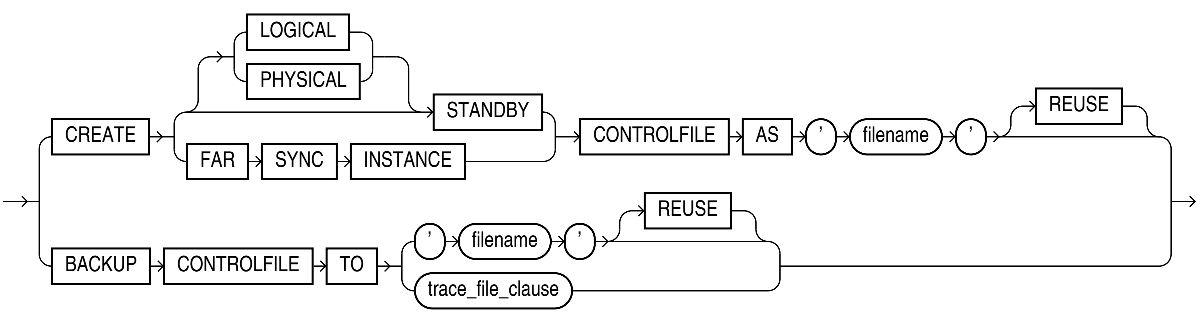


***supplemental_plsql_clause*::=**

***logfile_descriptor*::=**

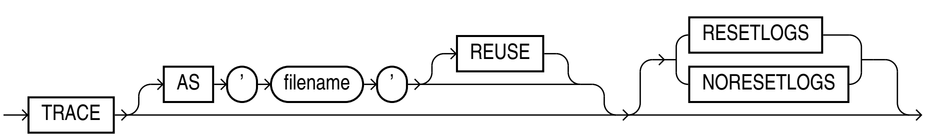


***controlfile_clauses*::=**

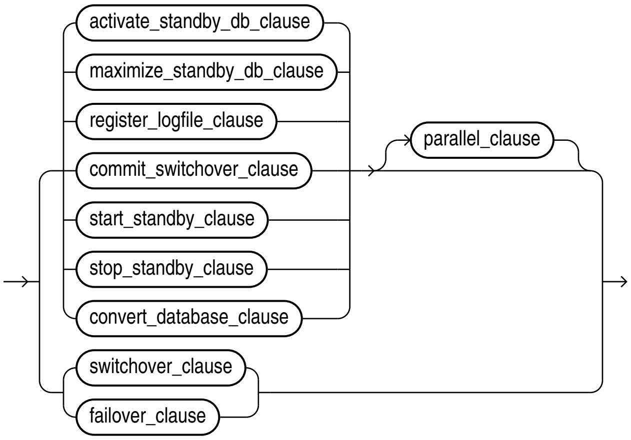


(*trace_file_clause*::=)

***trace_file_clause*::=**

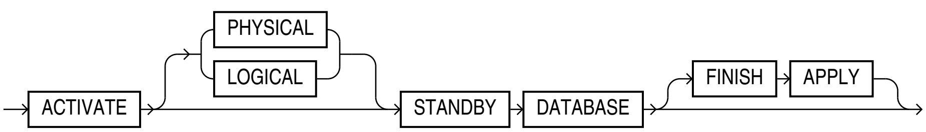


***standby_database_clauses*::=**

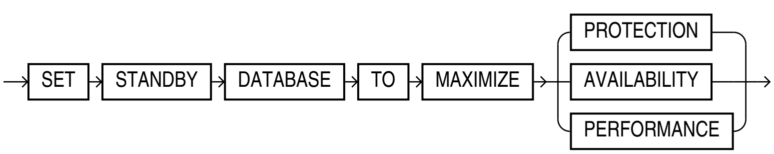


(*activate_standby_db_clause*::=, *maximize_standby_db_clause*::=, *register_logfile_clause*::=, *commit_switchover_clause*::=, *start_standby_clause*::=, *stop_standby_clause*::=, *convert_database_clause*::=, *parallel_clause*::=, *switchover_clause*::=, *failover_clause*::=)

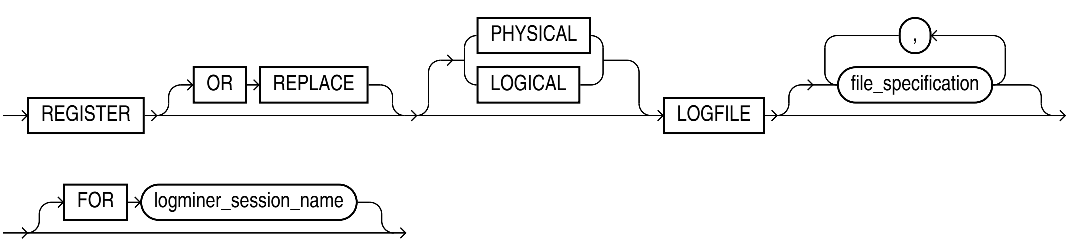
***activate_standby_db_clause*::=**



***maximize_standby_db_clause*::=**

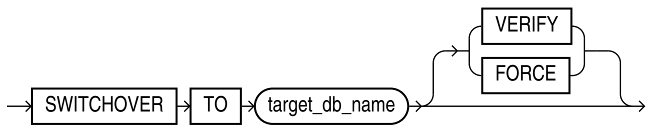


***register_logfile_clause*::=**

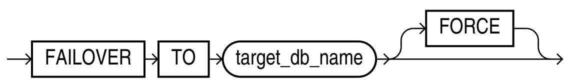


(*file_specification*::=)

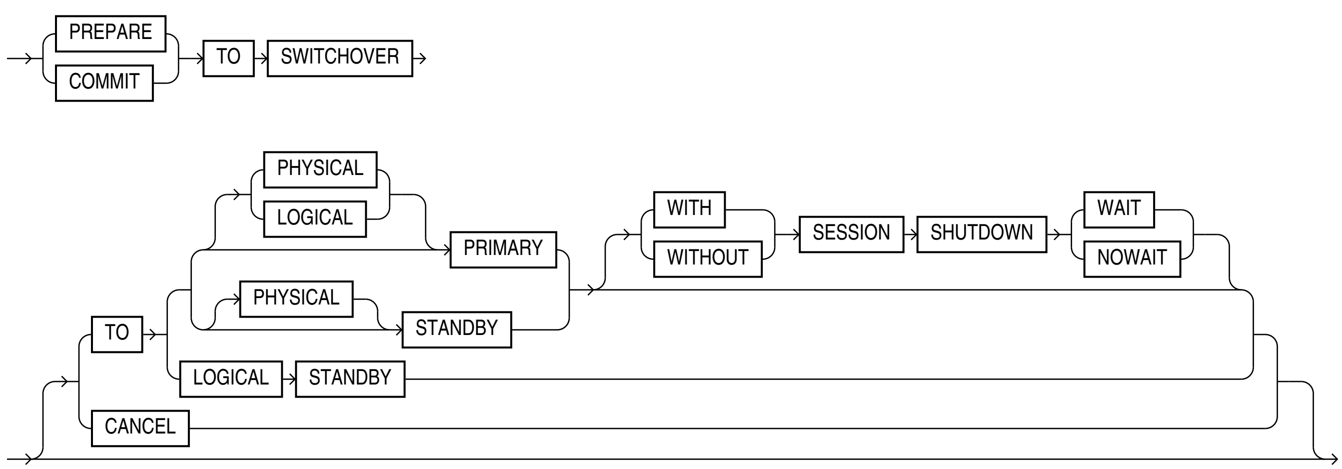
***switchover_clause*::=**



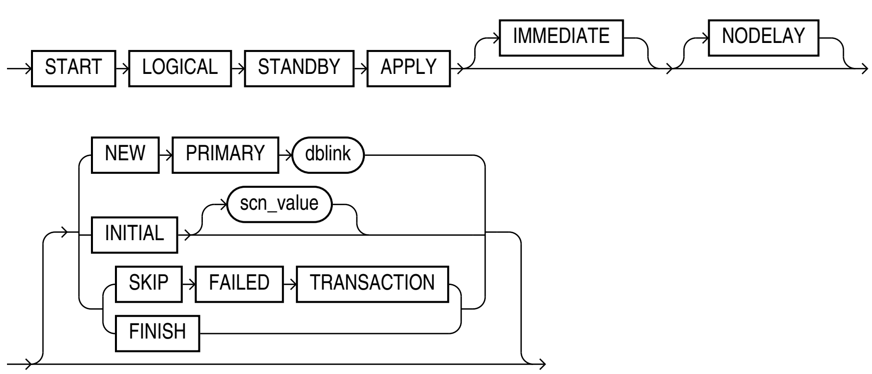
***failover_clause*::=**



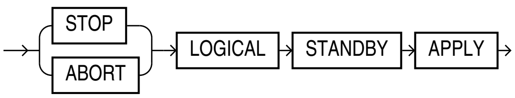
***commit_switchover_clause*::=**



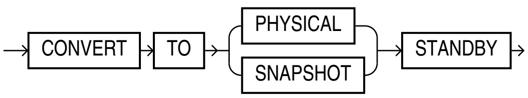
***start_standby_clause*::=**



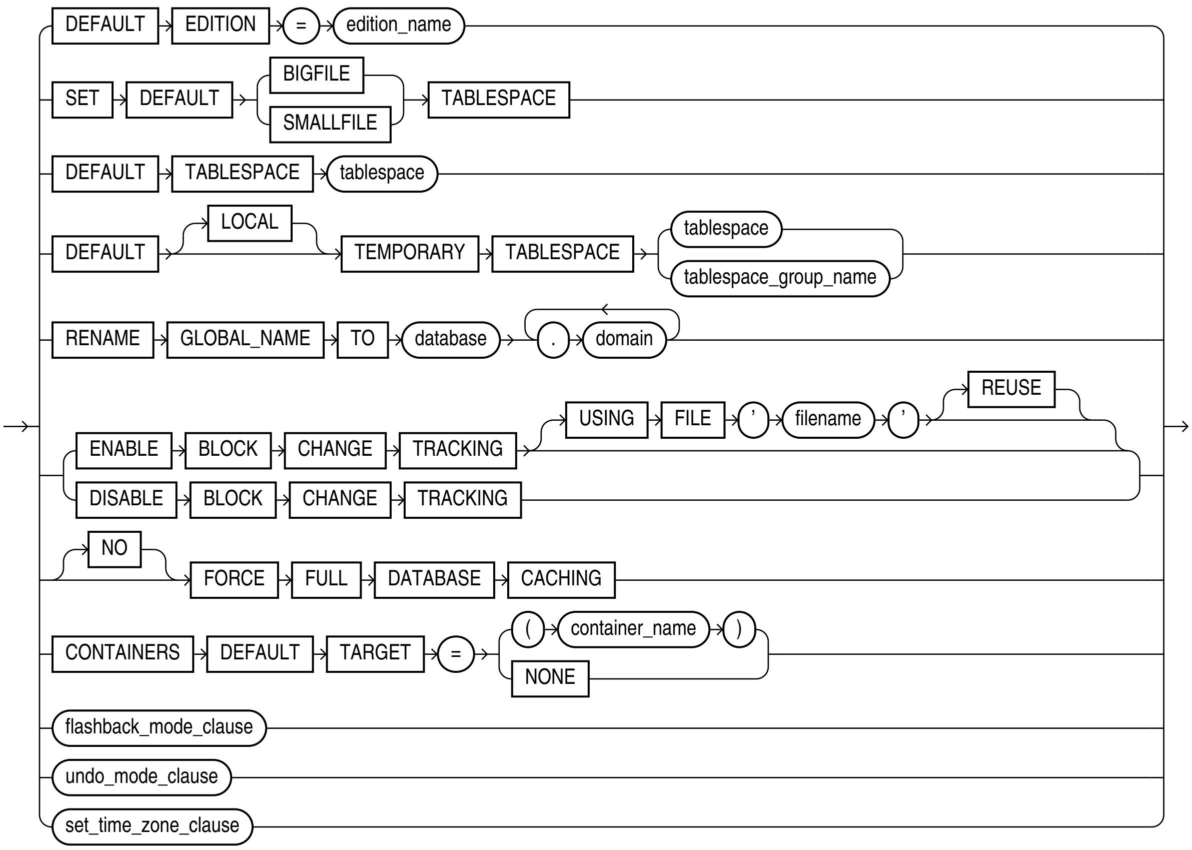
***stop_standby_clause*::=**



***convert_database_clause*::=**

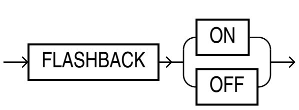


***default_settings_clauses*::=**

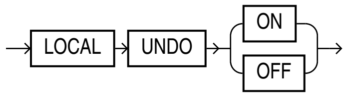


(*flashback_mode_clause*::=, *undo_mode_clause*::=, *set_time_zone_clause*::=)

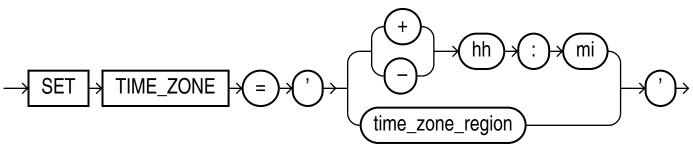
***flashback_mode_clause*::=**



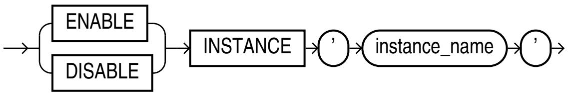
***undo_mode_clause*::=**



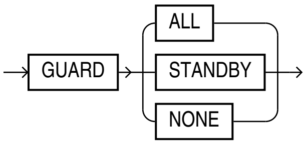
***set_time_zone_clause*::=**



***instance_clauses*::=**



***security_clause*::=**



Semantics

Notes on Using ALTER DATABASE in a CDB

When you issue the `ALTER DATABASE` statement while connected to a CDB, the behavior of the statement depends on the current container and the clause(s) you specify.

If the current container is the root, then `ALTER DATABASE` statements with the following clauses modify the entire CDB. In order to specify these clauses, you must have the commonly granted `ALTER DATABASE` privilege:

- *`startup_clauses`*
- *`recovery_clauses`*

  Note: A subset of the *`recovery_clauses`* are supported to back up and recover an individual pluggable database (PDB). In order to specify these clauses, you must have the `ALTER DATABASE` privilege, either granted commonly or granted locally in the PDB. Refer to "Notes on Using the recovery_clauses in a CDB" for more information.

- *logfile_clauses*
- *controlfile_clauses*
- *standby_database_clauses*
- *instance_clauses*
- *security_clause*
- RENAME GLOBAL_NAME TO
- ENABLE BLOCK CHANGE TRACKING
- DISABLE BLOCK CHANGE TRACKING
- *undo_mode_clause*

If the current container is the root, then ALTER DATABASE statements with the following clauses modify only the root. In order to specify these clauses, you must have the ALTER DATABASE privilege, either granted commonly or granted locally in the root:

- *database_file_clauses*
- DEFAULT EDITION
- DEFAULT TABLESPACE

If the current container is the root, then ALTER DATABASE statements with the following clauses modify the root and set default values for the PDBs. In order to specify these clauses, you must have the commonly granted ALTER DATABASE privilege:

- DEFAULT [LOCAL] TEMPORARY TABLESPACE
- *flashback_mode_clause*
- SET DEFAULT { BIGFILE | SMALLFILE } TABLESPACE
- *set_time_zone_clause*

If the current container is a PDB, then ALTER DATABASE statements modify that PDB. In this case, you can issue only ALTER DATABASE clauses that are also supported by the ALTER PLUGGABLE DATABASE statement. This functionality is provided to maintain backward compatibility for applications that have been migrated to a CDB environment. The exception is modifying PDB storage limits, for which you must use the *pdb_storage_clause* of ALTER PLUGGABLE DATABASE. Refer to the documentation on ALTER PLUGGABLE DATABASE for complete information on these clauses.

database

Specify the name of the database to be altered. If you omit *database*, then Oracle Database alters the database identified by the value of the initialization parameter DB_NAME. You can alter only the database whose control files are specified by the initialization parameter CONTROL_FILES. The database identifier is not related to the Oracle Net database specification.

startup_clauses

The *startup_clauses* let you mount and open the database so that it is accessible to users.

MOUNT Clause

Use the `MOUNT` clause to mount the database. Do not use this clause when the database is already mounted.

MOUNT STANDBY DATABASE

You can specify `MOUNT STANDBY DATABASE` to mount a physical standby database. The keywords `STANDBY DATABASE` are optional, because Oracle Database determines automatically whether the database to be mounted is a primary or standby database. As soon as this statement executes, the standby instance can receive redo data from the primary instance.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on standby databases

MOUNT CLONE DATABASE

Specify `MOUNT CLONE DATABASE` to mount the clone database.

OPEN Clause

Use the `OPEN` clause to make the database available for normal use. You must mount the database before you can open it.

If you specify only `OPEN` without any other keywords, then the default is `OPEN READ WRITE NORESETLOGS` on a primary database, logical standby database, or snapshot standby database and `OPEN READ ONLY` on a physical standby database.

OPEN READ WRITE

Specify `OPEN READ WRITE` to open the database in read/write mode, allowing users to generate redo logs. This is the default if you are opening a primary database. You cannot specify this clause for a physical standby database.

> **See Also:**
>
> "READ ONLY / READ WRITE: Example"

RESETLOGS | NORESETLOGS

This clause determines whether Oracle Database resets the current log sequence number to 1, archives any unarchived logs (including the current log), and discards any redo information that was not applied during recovery, ensuring that it will never be applied. Oracle Database uses `NORESETLOGS` automatically except in the following specific situations, which require a setting for this clause:

- You must specify `RESETLOGS`:
  - After performing incomplete media recovery or media recovery using a backup control file

  – After a previous OPEN RESETLOGS operation that did not complete
  – After a FLASHBACK DATABASE operation
- If a created control file is mounted, then you must specify RESETLOGS if the online logs are lost, or you must specify NORESETLOGS if they are not lost.

UPGRADE | DOWNGRADE

Use these OPEN clause parameters only if you are upgrading or downgrading a database. This clause instructs Oracle Database to modify system parameters dynamically as required for upgrade and downgrade, respectively. You can achieve the same result using the SQL*Plus STARTUP UPGRADE or STARTUP DOWNGRADE command.

When you use the UPGRADE or DOWNGRADE parameters for a CDB, the root container is opened in the specified mode, but all other containers are opened in READ WRITE mode.

> **See Also:**
>
> - *Oracle Database Upgrade Guide* for information on the steps required to upgrade or downgrade a database from one release to another
> - *SQL*Plus User's Guide and Reference* for information on the SQL*Plus STARTUP command

OPEN READ ONLY

Specify OPEN READ ONLY to restrict users to read-only transactions, preventing them from generating redo logs. This setting is the default when you are opening a physical standby database, so that the physical standby database is available for queries even while archive logs are being copied from the primary database site.

Restrictions on Opening a Database

The following restrictions apply to opening a database:

- You cannot open a database in READ ONLY mode if it is currently opened in READ WRITE mode by another instance.
- You cannot open a database in READ ONLY mode if it requires recovery.
- You cannot take tablespaces offline while the database is open in READ ONLY mode. However, you can take data files offline and online, and you can recover offline data files and tablespaces while the database is open in READ ONLY mode.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for additional information about opening a physical standby database

recovery_clauses

The `recovery_clauses` include post-backup operations. For all of these clauses, Oracle Database recovers the database using any incarnations of data files and log files that are known to the current control file.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for information on backing up the database and "Database Recovery: Examples"

Notes on Using the *recovery_clauses* in a CDB

When the current container is the root, you can specify all of the `recovery_clauses` to back up and recover the entire CDB.

When the current container is a PDB, you can specify the following subclauses of the `recovery_clauses` to back up and recover the PDB:

- `BEGIN BACKUP`
- `END BACKUP`
- `full_database_recovery`: You can specify only the `DATABASE` keyword
- `partial_database_recovery`
- The `LOGFILE` and `CONTINUE` clauses of `general_recovery`

You can also specify the preceding subclauses using the `pdb_recovery_clauses` of `ALTER PLUGGABLE DATABASE`. Refer to the syntax diagram *pdb_recovery_clauses* of `ALTER PLUGGABLE DATABASE`.

general_recovery

The `general_recovery` clause lets you control media recovery for the database or standby database or for specified tablespaces or files. You can use this clause when your instance has the database mounted, open or closed, and the files involved are not in use.

> **Note:**
>
> Parallelism is enabled by default during full or partial database recovery and logfile recovery. The database computes the degree of parallelism. You can disable parallelism of these operations by specifying `NOPARALLEL`, or specify a degree of parallelism with `PARALLEL` *`integer`*, as shown in the respective syntax diagrams.

Restrictions on General Database Recovery

General recovery is subject to the following restrictions:

- You can recover the entire database only when the database is closed.

- Your instance must have the database mounted in exclusive mode.
- You can recover tablespaces or data files when the database is open or closed, if the tablespaces or data files to be recovered are offline.
- You cannot perform media recovery if you are connected to Oracle Database through the shared server architecture.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for more information on RMAN media recovery and user-defined media recovery
> - *SQL*Plus User's Guide and Reference* for information on the SQL*Plus `RECOVER` command

AUTOMATIC

Specify `AUTOMATIC` if you want Oracle Database to automatically generate the name of the next archived redo log file needed to continue the recovery operation. If the `LOG_ARCHIVE_DEST_`*`n`* parameters are defined, then Oracle Database scans those that are valid and enabled for the first local destination. It uses that destination in conjunction with `LOG_ARCHIVE_FORMAT` to generate the target redo log filename. If the `LOG_ARCHIVE_DEST_`*`n`* parameters are not defined, then Oracle Database uses the value of the `LOG_ARCHIVE_DEST` parameter instead.

If the resulting file is found, then Oracle Database applies the redo contained in that file. If the file is not found, then Oracle Database prompts you for a filename, displaying the generated filename as a suggestion.

If you specify neither `AUTOMATIC` nor `LOGFILE`, then Oracle Database prompts you for a filename, displaying the generated filename as a suggestion. You can then accept the generated filename or replace it with a fully qualified filename. If you know that the archived filename differs from what Oracle Database would generate, then you can save time by using the `LOGFILE` clause.

FROM *'location'*

Specify `FROM '`*`location`*`'` to indicate the location from which the archived redo log file group is read. The value of *`location`* must be a fully specified file location following the conventions of your operating system. If you omit this parameter, then Oracle Database assumes that the archived redo log file group is in the location specified by the initialization parameter `LOG_ARCHIVE_DEST` or `LOG_ARCHIVE_DEST_1`.

full_database_recovery

The *`full_database_recovery`* clause lets you recover an entire database.

DATABASE

Specify the `DATABASE` clause to recover the entire database. This is the default. You can use this clause only when the database is closed.

STANDBY DATABASE

Specify the `STANDBY DATABASE` clause to manually recover a physical standby database using the control file and archived redo log files copied from the primary database. The standby database must be mounted but not open.

This clause recovers only online data files.

- Use the `UNTIL` clause to specify the duration of the recovery operation.
  - `CANCEL` indicates cancel-based recovery. This clause recovers the database until you issue the `ALTER DATABASE` statement with the `RECOVER CANCEL` clause.
  - `TIME` indicates time-based recovery. This parameter recovers the database to the time specified by the date. The date must be a character literal in the format `'YYYY-MM-DD:HH24:MI:SS'`.
  - `CHANGE` indicates change-based recovery. This parameter recovers the database to a transaction-consistent state immediately before the system change number specified by *`integer`*.
  - `CONSISTENT` recovers the database until all online files are brought to a consistent SCN point so that the database can be open in read only mode. This clauses requires the controlfile to be a backup controlfile.
- Specify `USING BACKUP CONTROLFILE` if you want to use a backup control file instead of the current control file.
- Specify the `SNAPSHOT TIME` clause to recover the database with a storage snapshot using Storage Snapshot Optimization. This clause can be used in cases where the database was not placed in backup mode when the storage snapshot was created.
  - *`date`* must be a character literal in the format `'YYYY-MM-DD:HH24:MI:SS'`. It must represent a time that is immediately after the snapshot was completed. If you specify the `UNTIL TIME` clause, then `SNAPSHOT TIME` *`date`* must be earlier than `UNTIL TIME` *`date`*.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information on recovery using Storage Snapshot Optimization

partial_database_recovery

The *`partial_database_recovery`* clause lets you recover individual tablespaces and data files.

TABLESPACE

Specify the `TABLESPACE` clause to recover only the specified tablespaces. You can use this clause if the database is open or closed, provided the tablespaces to be recovered are offline.

> **See Also:**
>
> "Using Parallel Recovery Processes: Example"

DATAFILE

Specify the `DATAFILE` clause to recover the specified data files. You can use this clause when the database is open or closed, provided the data files to be recovered are offline.

You can identify the data file by name or by number. If you identify it by number, then *`filenumber`* is an integer representing the number found in the `FILE#` column of the `V$DATAFILE` dynamic performance view or in the `FILE_ID` column of the `DBA_DATA_FILES` data dictionary view.

STANDBY {TABLESPACE | DATAFILE}

In earlier releases, you could specify `STANDBY TABLESPACE` or `STANDBY DATAFILE` to recover older backups of a specific tablespace or a specific data file on the standby to be consistent with the rest of the standby database. These two clauses are now desupported. Instead, to recover the standby database to a consistent point, but no further, use the statement `ALTER DATABASE RECOVER MANAGED STANDBY DATABASE UNTIL CONSISTENT`.

LOGFILE

Specify the `LOGFILE` '*`filename`*' to continue media recovery by applying the specified redo log file.

TEST

Use the `TEST` clause to conduct a trial recovery. A trial recovery is useful if a normal recovery procedure has encountered some problem. It lets you look ahead into the redo stream to detect possible additional problems. The trial recovery applies redo in a way similar to normal recovery, but it does not write changes to disk, and it rolls back its changes at the end of the trial recovery.

You can use this clause only if you have restored a backup taken since the last `RESETLOGS` operation. Otherwise, Oracle Database returns an error.

ALLOW ... CORRUPTION

The `ALLOW` *`integer`* `CORRUPTION` clause lets you specify, in the event of logfile corruption, the number of corrupt blocks that can be tolerated while allowing recovery to proceed.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for information on database recovery in general
> - *Oracle Data Guard Concepts and Administration* for information on managed recovery of standby databases

CONTINUE

Specify `CONTINUE` to continue multi-instance recovery after it has been interrupted to disable a thread.

Specify `CONTINUE DEFAULT` to continue recovery using the redo log file that Oracle Database would automatically generate if no other logfile were specified. This clause is equivalent to specifying `AUTOMATIC`, except that Oracle Database does not prompt for a filename.

CANCEL

Specify `CANCEL` to terminate cancel-based recovery.

managed_standby_recovery

Use the *`managed_standby_recovery`* clause to start and stop Redo Apply on a physical standby database. Redo Apply keeps the standby database transactionally consistent with the primary database by continuously applying redo received from the primary database.

A primary database transmits its redo data to standby sites. As the redo data is written to redo log files at the physical standby site, the log files become available for use by Redo Apply. You can use the *`managed_standby_recovery`* clause when your standby instance has the database mounted or is opened read-only.

> **Note:**
>
> Beginning with Oracle Database 12*c*, **real-time apply** is enabled by default during Redo Apply. Real-time apply recovers redo from the standby redo log files as soon as they are written, without requiring them to be archived first at the physical standby database. You can disable real-time apply with the `USING ARCHIVED LOGFILE` clause. Refer to:
>
> - *Oracle Data Guard Concepts and Administration* for more information on real-time apply
> - USING ARCHIVED LOGFILE Clause

> **Note:**
>
> Parallelism is enabled by default during Redo Apply. The database computes the degree of parallelism. You can disable parallelism of these operations by specifying `NOPARALLEL`, or specify a degree of parallelism with `PARALLEL` *`integer`*, as shown in the respective syntax diagrams.

Restrictions on Managed Standby Recovery

The same restrictions listed under *general_recovery* apply to this clause.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on the use of this clause

USING ARCHIVED LOGFILE Clause

Specify `USING ARCHIVED LOGFILE` to start Redo Apply without enabling real-time apply.

DISCONNECT

Specify `DISCONNECT` to indicate that Redo Apply should be performed in the background, leaving the current session available for other tasks. The `FROM SESSION` keywords are optional and are provided for semantic clarity.

NODELAY

The `NODELAY` clause overrides the `DELAY` attribute on the `LOG_ARCHIVE_DEST_n` parameter on the primary database. If you do not specify the `NODELAY` clause, then application of the archived redo log file is delayed according to the `DELAY` attribute of the `LOG_ARCHIVE_DEST_n` setting (if any). If the `DELAY` attribute was not specified on that parameter, then the archived redo log file is applied immediately to the standby database.

If you specify real-time apply with the `USING CURRENT LOGFILE` clause, then any `DELAY` value specified for the `LOG_ARCHIVE_DEST_n` parameter at the primary for this standby is ignored, and `NODELAY` is the default.

UNTIL CHANGE Clause

Use this clause to instruct Redo Apply to recover redo data up to, but not including, the specified system change number.

UNTIL CONSISTENT

Use this clause to recover the standby database to a consistent SCN point so that the standby database can be opened in read only mode.

USING INSTANCES

This clause is applicable only for Oracle Real Application Clusters (Oracle RAC) or Oracle RAC One Node databases and allows you to start apply processes on multiple instances of the standby that are started in the same mode (`MOUNTED` or `READ ONLY`) as the instance on which the command is executed. Specify `USING INSTANCES ALL` to perform Redo Apply on all instances in an Oracle RAC standby database started in the same mode. Specify `USING INSTANCES` *`integer`* to perform Redo Apply on the specified number of instances that are started in the same mode. For *`integer`*, specify an integer value from 1 to the number of instances in the standby database. The database chooses the instances on which to perform Redo Apply; you cannot specify particular instances. For example, if you specify 4 instances from an instance that is `MOUNTED` and only 3 instances of the standby are running in the `MOUNTED` mode, then Redo Apply will only be started on 3 instances. If you omit the `USING INSTANCES` clause, then Oracle Database performs Redo Apply only on the instance where the command was executed.

FINISH

Specify `FINISH` to complete applying all available redo data in preparation for a failover.

Use the `FINISH` clause only in the event of the failure of the primary database. This clause overrides any specified delay intervals and applies all available redo immediately. After the `FINISH` command completes, this database can no longer run in the standby database role, and it must be converted to a primary database by issuing the `ALTER DATABASE COMMIT TO SWITCHOVER TO PRIMARY` statement.

CANCEL

Specify `CANCEL` to stop Redo Apply immediately. Control is returned as soon as Redo Apply stops.

TO LOGICAL STANDBY Clause

Use this clause to convert a physical standby database into a logical standby database.

db_name

Specify a database name to identify the new logical standby database. If you are using a server parameter file (spfile) at the time you issue this statement, then the database will update the file with appropriate information about the new logical standby database. If you are not using an spfile, then the database issues a message reminding you to set the name of the `DB_NAME` parameter after shutting down the database. In addition, you must invoke the `DBMS_LOGSTDBY.BUILD` PL/SQL procedure on the primary database before using this clause on the standby database.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_LOGSTDBY.BUILD` procedure

KEEP IDENTITY

Use this clause if you want to use the rolling upgrade feature provided by a logical standby and also revert to the original configuration of a primary database and a physical standby. A logical standby database created using this clause provides only limited support for switchover and failover. Therefore, do not use this clause create a general-purpose logical standby database.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on rolling upgrade

Deprecated Managed Standby Recovery Clauses

The following clauses appeared in the syntax of earlier releases. They have been deprecated and are no longer needed. Oracle recommends that you do not use these clauses.

FINISH FORCE, FINISH WAIT, FINISH NOWAIT

These optional forms of the FINISH clause are deprecated. Their semantics are presented here for backward compatibility:

- FORCE terminates inactive redo transport sessions that would otherwise prevent FINISH processing from beginning.
- NOWAIT returns control to the foreground process before the recovery completes
- WAIT (the default) returns control to the foreground process after recovery completes

When specified, these clauses are ignored. Terminal recovery now runs in the foreground and always terminates all redo transport sessions. Therefore control is not returned to the user until recovery completes.

CANCEL IMMEDIATE, CANCEL WAIT, CANCEL NOWAIT

These optional forms of the CANCEL clause are deprecated. Their semantics are presented here for backward compatibility:

- Include the IMMEDIATE keyword to stop Redo Apply *before* completely applying the current redo log file. Session control returns when Redo Apply actually stops.
- Include the NOWAIT keyword to return session control without waiting for the CANCEL operation to complete.

When specified, these clauses are ignored. Redo Apply is now always cancelled immediately and control returns to the session only after the operation completes.

USING CURRENT LOGFILE Clause

The USING CURRENT LOGFILE clause is deprecated. It invokes real-time apply during Redo Apply. However, this is now the default behavior and this clause is no longer useful.

BACKUP Clauses

Use these clauses to move all the data files in the database into or out of online backup mode (also called hot backup mode).

> **See Also:**
>
> ALTER TABLESPACE for information on moving all data files in an individual tablespace into and out of online backup mode

BEGIN BACKUP Clause

Specify BEGIN BACKUP to move all data files in the database into online backup mode. The database must be mounted and open, and media recovery must be enabled (the database must be in ARCHIVELOG mode).

While the database is in online backup mode, you cannot shut down the instance normally, begin backup of an individual tablespace, or take any tablespace offline or make it read only.

This clause has no effect on data files that are in offline or on read-only tablespaces.

END BACKUP Clause

Specify END BACKUP to take out of online backup mode any data files in the database currently in online backup mode. The database must be mounted (either open or closed) when you perform this operation.

After a system failure, instance failure, or SHUTDOWN ABORT operation, Oracle Database does not know whether the files in online backup mode match the files at the time the system crashed. If you know the files are consistent, then you can take either individual data files or all data files out of online backup mode. Doing so avoids media recovery of the files upon startup.

- To take an individual data file out of online backup mode, use the ALTER DATABASE DATAFILE ... END BACKUP statement. See *database_file_clauses* .
- To take all data files in a tablespace out of online backup mode, use an ALTER TABLESPACE ... END BACKUP statement.

database_file_clauses

The *database_file_clauses* let you modify data files and temp files. You can use any of the following clauses when your instance has the database mounted, open or closed, and the files involved are not in use. The exception is the *move_datafile_clause*, which allows you to move a data file that is in use.

RENAME FILE Clause

Use the RENAME FILE clause to rename data files, temp files, or redo log file members. You must create each filename using the conventions for filenames on your operating system before specifying this clause.

- To use this clause for a data file or temp file, the database must be mounted. The database can also be open, but the data file or temp file being renamed must be offline. In addition, you must first rename the file on the file system to the new name.
- To use this clause for logfiles, the database must be mounted but not open.
- If you have enabled block change tracking, then you can use this clause to rename the block change tracking file. The database must be mounted but not open when you rename the block change tracking file.

This clause renames only files in the control file. It does not actually rename them on your operating system. The operating system files continue to exist, but Oracle Database no longer uses them.

See Also:

- *Oracle Database Backup and Recovery User's Guide* for information on recovery of data files and temp files
- "Renaming a Log File Member: Example" and "Manipulating Temp Files: Example"

create_datafile_clause

Use the CREATE DATAFILE clause to create a new empty data file in place of an old one. You can use this clause to re-create a data file that was lost with no backup. The *filename* or *filenumber* must identify a file that is or was once part of the database. If you identify the file by number, then *filenumber* is an integer representing the number found in the FILE# column of the V$DATAFILE dynamic performance view or in the FILE_ID column of the DBA_DATA_FILES data dictionary view.

- Specify AS NEW to create an Oracle-managed data file with a system-generated filename, the same size as the file being replaced, in the default file system location for data files.
- Specify AS *file_specification* to assign a file name (and optional size) to the new data file. Use the *datafile_tempfile_spec* form of *file_specification* (see *file_specification*) to list regular data files and temp files in an operating system file system or to list Oracle Automatic Storage Management (Oracle ASM) disk group files.

If the original file (*filename* or *filenumber*) is an existing Oracle-managed data file, then Oracle Database attempts to delete the original file after creating the new file. If the original file is an existing user-managed data file, then Oracle Database does not attempt to delete the original file.

If you omit the AS clause entirely, then Oracle Database creates the new file with the same name and size as the file specified by *filename* or *filenumber*.

During recovery, all archived redo logs written to since the original data file was created must be applied to the new, empty version of the lost data file.

Oracle Database creates the new file in the same state as the old file when it was created. You must perform media recovery on the new file to return it to the state of the old file at the time it was lost.

Restrictions on Creating New Data Files

The creation of new data files is subject to the following restrictions:

- You cannot create a new file based on the first data file of the SYSTEM tablespace.
- You cannot specify the *autoextend_clause* of *datafile_tempfile_spec* in this CREATE DATAFILE clause.

> **See Also:**
>
> - "DATAFILE Clause" of CREATE DATABASE for information on the result of this clause if you do not specify a name for the new data file
> - *file_specification* for a full description of the file specification (*datafile_tempfile_spec*) and "Creating a New Data File: Example"

alter_datafile_clause

The DATAFILE clause lets you manipulate a file that you identify by name or by number. If you identify it by number, then *filenumber* is an integer representing the number found in the FILE# column of the V$DATAFILE dynamic performance view or in the FILE_ID column of the DBA_DATA_FILES data dictionary view. The DATAFILE clauses affect your database files as follows:

ONLINE

Specify ONLINE to bring the data file online.

OFFLINE

Specify OFFLINE to take the data file offline. If the database is open, then you must perform media recovery on the data file before bringing it back online, because a checkpoint is not performed on the data file before it is taken offline.

FOR DROP

If the database is in `NOARCHIVELOG` mode, then you must specify `FOR DROP` clause to take a data file offline. However, this clause does not remove the data file from the database. To do that, you must use an operating system command or drop the tablespace in which the data file resides. Until you do so, the data file remains in the data dictionary with the status `RECOVER` or `OFFLINE`.

If the database is in `ARCHIVELOG` mode, then Oracle Database ignores the `FOR DROP` clause.

RESIZE

Specify `RESIZE` if you want Oracle Database to attempt to increase or decrease the size of the data file to the specified absolute size in bytes. There is no default, so you must specify a size.

If sufficient disk space is not available for the increased size, or if the file contains data beyond the specified decreased size, then Oracle Database returns an error.

> **See Also:**
>
> "Resizing a Data File: Example"

END BACKUP

Specify `END BACKUP` to take the data file out of online backup mode. The `END BACKUP` clause is described more fully at the top level of the syntax of `ALTER DATABASE`. See "END BACKUP Clause".

ENCRYPT | DECRYPT

Use these clauses to perform offline encryption or decryption of the data file using Transparent Data Encryption (TDE). In any given tablespace, either all data files must be encrypted or all data files must be unencrypted.

Before issuing either of these clauses, the database must be mounted. The database can also be open, but the tablespace that contains the data file being encrypted or decrypted must be offline. The TDE master key must be loaded into database memory.

- Specify `ENCRYPT` to encrypt an unencrypted data file. The data file is encrypted using the `AES128` algorithm.
- Specify `DECRYPT` to decrypt a data file. The data file must have been previously encrypted with the `ALTER DATABASE DATAFILE` ... `ENCRYPT` statement.

Restrictions on Encrypting and Decrypting Data Files

The following restrictions apply to the `ENCRYPT` and `DECRYPT` clauses:

- You cannot encrypt or decrypt a temporary data file of a temporary tablespace. Instead, you must drop the temporary tablespace and recreate it as an encrypted tablespace.
- Oracle recommends against encrypting the data files of an undo tablespace. Doing so prevents the keystore from being closed, which prevents the database from functioning. Furthermore, this practice is unnecessary because all undo

records that are associated with an encrypted tablespace are already automatically encrypted in the undo tablespace.

> **Note:**
>
> The use of the `ENCRYPT` or `DECRYPT` clause is only one step in a series of steps for performing offline encryption or decryption of a data file. Refer to *Oracle Database Advanced Security Guide* for the complete set of steps before you use either of these clauses.

alter_tempfile_clause

Use the `TEMPFILE` clause to resize your temporary data file or specify the *`autoextend_clause`*, with the same effect as for a permanent data file. The database must be open. You can identify the temp file by name or by number. If you identify it by number, then *`filenumber`* is an integer representing the number found in the `FILE#` column of the `V$TEMPFILE` dynamic performance view.

> **Note:**
>
> On some operating systems, Oracle does not allocate space for a temp file until the temp file blocks are actually accessed. This delay in space allocation results in faster creation and resizing of temp files, but it requires that sufficient disk space is available when the temp files are later used. To avoid potential problems, before you create or resize a temp file, ensure that the available disk space exceeds the size of the new temp file or the increased size of a resized temp file. The excess space should allow for anticipated increases in disk space use by unrelated operations as well. Then proceed with the creation or resizing operation.

DROP

Specify `DROP` to drop *`tempfile`* from the database. The tablespace remains.

If you specify `INCLUDING DATAFILES`, then Oracle Database also deletes the associated operating system files and writes a message to the alert log for each such deleted file. You can achieve the same result using an `ALTER TABLESPACE` ... `DROP TEMPFILE` statement. Refer to the `ALTER TABLESPACE` DROP Clause for more information.

move_datafile_clause

Use the `MOVE DATAFILE` clause to move an online data file to a new location. The database can be open and accessing the data file when you perform this operation. The database creates a copy of the data file when it is performing this operation. Ensure that there is adequate disk space for the original data file and the copy before using this clause.

You can specify the original data file using the *`file_name`*, *`ASM_filename`*, or *`file_number`*. Refer to *ASM_filename* for information on ASM file names. If you identify the file by number, then *`file_number`* is an integer representing the number found in the `FILE#` column of the `V$DATAFILE` dynamic performance view or in the `FILE_ID` column of the `DBA_DATA_FILES` data dictionary view.

Use the TO clause to specify the new *file_name* or *ASM_filename*. If you are using Oracle Managed Files, then you can omit the TO clause. In this case, Oracle Database creates a unique name for the data file and saves it in the directory specified by the DB_CREATE_FILE_DEST initialization parameter.

If you specify REUSE, then the new data file is created even if it already exists.

If you specify KEEP, then the original data file will be kept after the MOVE DATAFILE operation. You cannot specify KEEP if the original data file is an Oracle Managed File. You can specify KEEP if the new data file is an Oracle Managed File.

autoextend_clause

Use the *autoextend_clause* to enable or disable the automatic extension of a new or existing data file or temp file. Refer to *file_specification* for information about this clause.

logfile_clauses

The logfile clauses let you add, drop, or modify log files.

ARCHIVELOG

Specify ARCHIVELOG if you want the contents of a redo log file group to be archived before the group can be reused. This mode prepares for the possibility of media recovery. Use this clause only after shutting down your instance normally, or immediately with no errors, and then restarting it and mounting the database.

MANUAL

Specify MANUAL to indicate that Oracle Database should create redo log files, but the archiving of the redo log files is controlled entirely by the user. This clause is provided for backward compatibility, for example for users who archive directly to tape. If you specify MANUAL, then:

- Oracle Database does not archive redo log files when a log switch occurs. You must handle this manually.
- You cannot have specified a standby database as an archivelog destinations. As a result, the database cannot be in MAXIMUM PROTECTION or MAXIMUM AVAILABILITY standby protection mode.

If you omit this clause, then Oracle Database automatically archives the redo log files to the destination specified in the LOG_ARCHIVE_DEST_*n* initialization parameters.

NOARCHIVELOG

Specify NOARCHIVELOG if you do not want the contents of a redo log file group to be archived so that the group can be reused. This mode does not prepare for recovery after media failure. Use this clause only if your instance has the database mounted but not open.

[NO] FORCE LOGGING

Use this clause to put the database into or take the database out of FORCE LOGGING mode. The database must be mounted or open.

In FORCE LOGGING mode, Oracle Database logs all changes in the database except changes in temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any NOLOGGING or FORCE LOGGING settings you specify for individual tablespaces and any NOLOGGING settings you specify for individual database objects.

If you specify FORCE LOGGING, then Oracle Database waits for all ongoing unlogged operations to finish.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on when to use FORCE LOGGING mode

RENAME FILE Clause

This clause has the same function for logfiles that it has for data files and temp files. See "RENAME FILE Clause".

CLEAR LOGFILE Clause

Use the CLEAR LOGFILE clause to reinitialize an online redo log, optionally without archiving the redo log. CLEAR LOGFILE is similar to adding and dropping a redo log, except that the statement may be issued even if there are only two logs for the thread and may be issued for the current redo log of a closed thread.

For a standby database, if the STANDBY_FILE_MANAGEMENT initialization parameter is set to AUTO, and if any of the log files are Oracle Managed Files, Oracle Database will create as many Oracle-managed log files as are in the control file. The log file members will reside in the current default log file destination.

- You must specify UNARCHIVED if you want to reuse a redo log that was not archived.

  > **Note:**
  >
  > Specifying UNARCHIVED makes backups unusable if the redo log is needed for recovery.

- You must specify UNRECOVERABLE DATAFILE if you have taken the data file offline with the database in ARCHIVELOG mode (that is, you specified ALTER DATABASE ... DATAFILE OFFLINE without the DROP keyword), and if the unarchived log to be cleared is needed to recover the data file before bringing it back online. In this case, you must drop the data file and the entire tablespace once the CLEAR LOGFILE statement completes.

  Do not use CLEAR LOGFILE to clear a log needed for media recovery. If it is necessary to clear a log containing redo after the database checkpoint, then you must first perform incomplete media recovery. The current redo log of an open thread can be cleared. The current log of a closed thread can be cleared by switching logs in the closed thread.

  If the CLEAR LOGFILE statement is interrupted by a system or instance failure, then the database may hang. In this case, reissue the statement after the database is restarted. If the failure occurred because of I/O errors accessing one member of a log group, then that member can be dropped and other members added.

> **See Also:**
>
> "Clearing a Log File: Example"

add_logfile_clauses

Use these clauses to add redo log file groups to the database and to add new members to existing redo log file groups.

ADD LOGFILE Clause

Use the `ADD LOGFILE` clause to add one or more redo log file groups to the online redo log or standby redo log.

> **See Also:**
>
> - "LOGFILE Clause" of `CREATE DATABASE` for information on the result of this clause for Oracle Managed Files if you do not specify a name for the new log file group
> - "Adding Redo Log File Groups: Examples"
> - *Oracle Data Guard Concepts and Administration* for more information on standby redo logs

STANDBY

Use the `STANDBY` clause to add a redo log file group to the standby redo log. If you do not specify this clause, then a log file group is added to the online redo log.

INSTANCE

The `INSTANCE` clause is applicable only for Oracle Real Application Clusters (Oracle RAC) or Oracle RAC One Node databases. Specify the name of the instance for which you want to add a redo log file group. The instance name is a string of up to 80 characters. Oracle Database automatically uses the thread that is mapped to the specified instance. If no thread is mapped to the specified instance, then Oracle Database automatically acquires an available unmapped thread and assigns it to that instance. If you do not specify this clause, then Oracle Database executes the command as if you had specified the current instance. If the specified instance has no current thread mapping and there are no available unmapped threads, then Oracle Database returns an error.

THREAD

When adding a redo log file group to the standby redo log, use the `THREAD` clause to assign the log file group to a specific primary database redo thread. Query the `V$INSTANCE` view on the primary database to determine which redo threads have been opened, and specify one of these thread numbers.

You can also use the `THREAD` clause to assign a log file group to a specific redo thread when adding the log file group to the online redo log. This usage has been deprecated. The `INSTANCE` clause achieves the same purpose and is easier to use.

GROUP

The GROUP clause uniquely identifies the redo log file group among all groups in all threads and can range from 1 to the value specified for MAXLOGFILES in the CREATE DATABASE statement. You cannot add multiple redo log file groups having the same GROUP value. If you omit this parameter, then Oracle Database generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

redo_log_file_spec

Each *redo_log_file_spec* specifies a redo log file group containing one or more members (copies). If you do not specify a filename for the new log file, then Oracle Database creates Oracle Managed Files according to the rules described in the "LOGFILE Clause" of CREATE DATABASE.

> **See Also:**
>
> - *file_specification*
> - *Oracle Database Reference* for information on dynamic performance views

ADD LOGFILE MEMBER Clause

Use the ADD LOGFILE MEMBER clause to add new members to existing redo log file groups. Each new member is specified by '*filename*'. If the file already exists, then it must be the same size as the other group members and you must specify REUSE. If the file does not exist, then Oracle Database creates a file of the correct size. You cannot add a member to a group if all of the members of the group have been lost through media failure.

STANDBY

You must specify STANDBY when adding a member to a standby redo log file group. Otherwise, Oracle Database returns an error.

You can use the *logfile_descriptor* clause to specify an existing redo log file group in one of two ways:

GROUP *integer*

Specify the value of the GROUP parameter that identifies the redo log file group.

filename(s)

List all members of the redo log file group. You must fully specify each filename according to the conventions of your operating system.

> **See Also:**
>
> - "LOGFILE Clause" of `CREATE DATABASE` for information on the result of this clause for Oracle Managed Files if you do not specify a name for the new log file group
> - "Adding Redo Log File Group Members: Example"

drop_logfile_clauses

Use these clauses to drop redo log file groups or redo log file members.

DROP LOGFILE Clause

Use the `DROP LOGFILE` clause to drop all members of a redo log file group. If you use this clause to drop Oracle Managed Files, then Oracle Database also removes all log file members from disk. Specify a redo log file group as indicated for the `ADD LOGFILE MEMBER` clause.

- To drop the current log file group, you must first issue an `ALTER SYSTEM SWITCH LOGFILE` statement.
- You cannot drop a redo log file group if it needs archiving.
- You cannot drop a redo log file group if doing so would cause the redo thread to contain less than two redo log file groups.

> **See Also:**
>
> ALTER SYSTEM and "Dropping Log File Members: Example"

DROP LOGFILE MEMBER Clause

Use the `DROP LOGFILE MEMBER` clause to drop one or more redo log file members. Each *'filename'* must fully specify a member using the conventions for filenames on your operating system.

- To drop a log file in the current log, you must first issue an `ALTER SYSTEM SWITCH LOGFILE` statement. Refer to ALTER SYSTEM for more information.
- You cannot use this clause to drop all members of a redo log file group that contains valid data. To perform that operation, use the `DROP LOGFILE` clause.

> **See Also:**
>
> "Dropping Log File Members: Example"

switch_logfile_clause

This clause is useful when you are migrating the database to disks with a different block size that the block size of the current database. Use this clause to switch logfiles to a different block size for all externally enabled threads, including both open and closed threads. If you are migrating the database to use 4KB sector disks, then you must specify 4096 for *integer*. If you are unmigrating the database back to using 512B sector disks, then you must specify 512 for *integer*.

This clause is an extension of the existing `ALTER SYSTEM SWITCH LOGFILE` statement. That statement switches logs for a single thread. This clause switches logfiles for all externally enabled threads, including both open and closed threads.

Before using this clause, you must already have created at least two redo log groups with the same target block size on the migration target disk.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on migrating the database to disks with a different block size, and "Adding a Log File: Example"

supplemental_db_logging

Use these clauses to instruct Oracle Database to add or stop adding supplemental data into the log stream.

ADD SUPPLEMENTAL LOG Clause

Specify `ADD SUPPLEMENTAL LOG DATA` to enable **minimal supplemental logging**. Specify `ADD SUPPLEMENTAL LOG` *supplemental_id_key_clause* to enable column data logging in addition to minimal supplemental logging. Specify `ADD SUPPLEMENTAL LOG` *supplemental_plsql_clause* to enable supplemental logging of PL/SQL calls. Oracle Database does not enable either minimal supplemental logging or supplemental logging by default.

Minimal supplemental logging ensures that LogMiner (and any products building on LogMiner technology) will have sufficient information to support chained rows and various storage arrangements such as cluster tables.

If the redo generated on one database is to be the source of changes (to be mined and applied) at another database, as is the case with logical standby, then the affected rows need to be identified using column data (as opposed to rowids). In this case, you should specify the *supplemental_id_key_clause*.

You can query the appropriate columns in the `V$DATABASE` view to determine whether any supplemental logging has already been enabled.

You can use this clause when the database is open. However, Oracle Database will invalidate all DML cursors in the cursor cache, which will have an effect on performance until the cache is repopulated.

If you use this clause in a CDB, then the current container must be the root and the operation will be performed on the entire CDB.

For a full discussion of the *supplemental_id_clause*, refer to supplemental_id_key_clause in the documentation on `CREATE TABLE`.

> **See Also:**
>
> - *Oracle Data Guard Concepts and Administration* for information on supplemental logging on the primary database to support a logical standby database
> - *Oracle Database Utilities* for examples using the `supplemental_db_logging` clause syntax

DROP SUPPLEMENTAL LOG Clause

Use this clause to stop supplemental logging.

- Specify `DROP SUPPLEMENTAL LOG DATA` to instruct Oracle Database to stop placing minimal additional log information into the redo log stream whenever an update operation occurs. If Oracle Database is doing column data supplemental logging specified with the *`supplemental_id_key_clause`*, then you must first stop the column data supplemental logging with the `DROP SUPPLEMENTAL LOG` *`supplemental_id_key_clause`* and then specify this clause.
- Specify `DROP SUPPLEMENTAL LOG` *`supplemental_id_key_clause`* to drop some or all of the system-generated supplemental log groups. You must specify the *`supplemental_id_key_clause`* if the supplemental log groups you want to drop were added using that clause.
- Specify `DROP SUPPLEMENTAL LOG` *`supplemental_plsql_clause`* disable supplemental logging of PL/SQL calls.

If you use this clause in a CDB, then the current container must be the root and the operation will be performed on the entire CDB.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for information on supplemental logging

controlfile_clauses

The *`controlfile_clauses`* let you create or back up a control file.

CREATE CONTROLFILE Clause

The `CREATE CONTROLFILE` clause lets you create a control file.

- Specify `PHYSICAL STANDBY` to create a control file to be used to maintain a physical database. This is the default if you specify `STANDBY` and do not specify `PHYSICAL` or `LOGICAL`.
- Specify `LOGICAL STANDBY` to create a control file to be used to maintain a logical database.
- Specify `FAR SYNC INSTANCE` to create a control file to be used to maintain a Data Guard far sync instance.

If the file already exists, then you must specify `REUSE`. In an Oracle RAC environment, the control file must be on shared storage.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on creating control files

BACKUP CONTROLFILE Clause

Use the `BACKUP CONTROLFILE` clause to back up the current control file. The database must be open or mounted when you specify this clause.

TO '*filename*'

Use this clause to specify a binary backup of the control file. You must fully specify the *`filename`* using the conventions for your operating system. If the specified file already exists, then you must specify `REUSE`. In an Oracle RAC environment, *`filename`* must be on shared storage.

A binary backup contains information that is not captured if you specify `TO TRACE`, such as the archived log history, offline range for read-only and offline tablespaces, and backup sets and copies (if you use RMAN). If the `COMPATIBLE` initialization parameter is `10.2` or higher, binary control file backups include temp file entries.

TO TRACE

Specify `TO TRACE` if you want Oracle Database to write SQL statements to a trace file rather than making a physical backup of the control file. You can use SQL statements written to the trace file to start up the database, re-create the control file, and recover and open the database appropriately, based on the created control file. If you issue an `ALTER DATABASE BACKUP CONTROLFILE TO TRACE` statement while block change tracking is enabled, then the resulting trace file will contain a command to reenable block change tracking.

This statement issues an implicit `ALTER DATABASE REGISTER LOGFILE` statement, which creates incarnation records if the archived log files reside in the current archivelog destinations.

The trace file will also include `ALTER DATABASE REGISTER LOGFILE` statements for existing logfiles that reside in the current archivelog destinations. This will implicitly create database incarnation records for the branches of redo to which the logfiles apply.

You can copy the statements from the trace file into a script file, edit the statements as necessary, and use the script if all copies of the control file are lost (or to change the size of the control file).

- Specify `AS` *`filename`* if you want Oracle Database to place the trace output into a file called *`filename`* rather than into the standard trace file.
- Specify `REUSE` to allow Oracle Database to overwrite any existing file called *`filename`*.
- `RESETLOGS` indicates that the SQL statement written to the trace file for starting the database is `ALTER DATABASE OPEN RESETLOGS`. This setting is valid only if the online logs are unavailable.

- NORESETLOGS indicates that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN NORESETLOGS. This setting is valid only if all the online logs are available.

If you cannot predict the future state of the online logs, then specify neither RESETLOGS nor NORESETLOGS. In this case, Oracle Database puts both versions of the script into the trace file, and you can choose which version is appropriate when the script becomes necessary.

The trace files are stored in a subdirectory determined by the DIAGNOSTIC_DEST initialization parameter. You can find the name and location of the trace file to which the CREATE CONTROLFILE statements were written by looking in the alert log. You can also find the directory for trace files by querying the NAME and VALUE columns of the V$DIAG_INFO dynamic performance view.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on viewing the alert log

standby_database_clauses

Use these clauses to activate the standby database or to specify whether it is in protected or unprotected mode.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for descriptions of the physical and logical standby database and for information on maintaining and using standby databases

activate_standby_db_clause

Use the ACTIVATE STANDBY DATABASE clause to convert a standby database into a primary database.

> **Note:**
>
> Before using this command, refer to *Oracle Data Guard Concepts and Administration* for important usage information.

PHYSICAL

Specify PHYSICAL to activate a physical standby database. This is the default.

LOGICAL

Specify LOGICAL to activate a logical standby database. If you have more than one logical standby database, then you should first ensure that the same log data is available on all the standby systems.

FINISH APPLY

This clause applies only to logical standby databases. Use it to initiate **terminal apply**, which is the application of any remaining redo to bring the logical standby database to the same state as the primary database. When terminal apply is complete, the database completes the switchover from logical standby to primary database.

If you require immediate restoration of the database in spite of data loss, then omit this clause. The database will execute the switchover from logical standby to primary database immediately without terminal apply.

maximize_standby_db_clause

Use this clause to specify the level of protection for the data in your database environment. You specify this clause from the primary database.

> **Note:**
>
> The PROTECTED and UNPROTECTED keywords have been replaced for clarity but are still supported. PROTECTED is equivalent to TO MAXIMIZE PROTECTION. UNPROTECTED is equivalent to TO MAXIMIZE PERFORMANCE.

TO MAXIMIZE PROTECTION

This setting establishes **maximum protection mode** and offers the highest level of data protection. A transaction does not commit until all data needed to recover that transaction has been written to at least one physical standby database that is configured to use the SYNC log transport mode. If the primary database is unable to write the redo records to at least one such standby database, then the primary database is shut down. This mode guarantees zero data loss, but it has the greatest potential impact on the performance and availability of the primary database.

Restriction on Establishing Maximum Protection Mode

You can specify TO MAXIMIZE PROTECTION on an open database only if the current data protection mode is MAXIMUM AVAILABILITY and there is at least one synchronized standby database.

TO MAXIMIZE AVAILABILITY

This setting establishes **maximum availability mode** and offers the next highest level of data protection. A transaction does not commit until all data needed to recover that transaction has been written to at least one physical or logical standby database that is configured to use the SYNC log transport mode. Unlike maximum protection mode, the primary database does not shut down if it is unable to write the redo records to at least one such standby database. Instead, the protection is lowered to maximum performance mode until the fault has been corrected and the standby database has caught up with the primary database. This mode guarantees zero data loss unless the primary database fails while in maximum performance mode. Maximum availability mode provides the highest level of data protection that is possible without affecting the availability of the primary database.

TO MAXIMIZE PERFORMANCE

This setting establishes **maximum performance mode** and is the default setting. A transaction commits before the data needed to recover that transaction has been written to a standby database. Therefore, some transactions may be lost if the primary database fails and you are unable to recover the redo records from the primary database. This mode provides the highest level of data protection that is possible without affecting the performance of the primary database.

To determine the current mode of the database, query the `PROTECTION_MODE` column of the `V$DATABASE` dynamic performance view.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for full information on using these standby database settings

register_logfile_clause

Specify the `REGISTER LOGFILE` clause from the standby database to manually register log files from the failed primary. Use the *`redo_log_file_spec`* form of *`file_specification`* (see *file_specification*) to list regular redo log files in an operating system file system or to list Oracle ASM disk group redo log files.

When a log file is from an unknown incarnation, the `REGISTER LOGFILE` clause causes an incarnation record to be added to the `V$DATABASE_INCARNATION` view. If the newly registered log file belongs to an incarnation having a higher `RESETLOGS_TIME` than the current `RECOVERY_TARGET_INCARNATION#`, then the `REGISTER LOGFILE` clause also causes `RECOVERY_TARGET_INCARNATION#` to be changed to correspond to the newly added incarnation record.

OR REPLACE

Specify `OR REPLACE` to allow an existing archivelog entry in the standby database to be updated, for example, when its location or file specification changes. The system change numbers of the entries must match exactly, and the original entry must have been created by the managed standby log transmittal mechanism.

FOR *logminer_session_name*

This clause is useful in a Streams environment. It lets you register the log file with one specified LogMiner session.

switchover_clause

> **Caution:**
>
> Before using this command, refer to *Oracle Data Guard Concepts and Administration* for complete usage information.

Use this clause to perform a switchover to a physical standby database. Specify this clause from the primary database. For `target_db_name`, specify the `DB_UNIQUE_NAME` of the standby database.

VERIFY

Use this clause to verify that a physical standby database is ready for a switchover. Specify this clause from the primary database. For `target_db_name`, specify the `DB_UNIQUE_NAME` of the standby database. If the standby database is ready for a switchover, then the "Database Altered" message is returned. Otherwise, an error message that will assist you in preparing the standby database for a switchover is returned.

FORCE

Use this clause if a previous switchover command failed and created a configuration with no primary database. Specify this clause from the physical standby database that you want to convert to the primary database. For `target_db_name`, specify the `DB_UNIQUE_NAME` of the database that you want to convert to the primary database.

failover_clause

> **Caution:**
>
> Before using this command, refer to *Oracle Data Guard Concepts and Administration* for complete usage information.

Use this clause to perform a failover to a physical standby database. Specify this clause from the standby database. For `target_db_name`, specify the `DB_UNIQUE_NAME` of the standby database.

FORCE

This clause has meaning only when the failover target is serviced by a Data Guard far sync instance. Use this clause when a previous failover command failed and the reason for the failure cannot be resolved. It instructs the failover to ignore any failures encountered when interacting with the Data Guard far sync instance and proceed with the failover, if at all possible.

commit_switchover_clause

Use this clause to perform database role transitions in a Data Guard configuration.

> **Caution:**
>
> Before using this command, refer to *Oracle Data Guard Concepts and Administration* for complete usage information.

PREPARE TO SWITCHOVER

This clause prepares a primary database to become a logical standby database or a logical standby database to become a primary database.

- Specify `PREPARE TO SWITCHOVER TO LOGICAL STANDBY` on a primary database.
- Specify `PREPARE TO SWITCHOVER TO PRIMARY DATABASE` on a logical standby database.

COMMIT TO SWITCHOVER

This clause switches a primary database to a standby database role or switches a standby database to the primary database role.

- Specify `COMMIT TO SWITCHOVER TO PHYSICAL STANDBY` or `COMMIT TO SWITCHOVER TO LOGICAL STANDBY` on a primary database.
- Specify `COMMIT TO SWITCHOVER TO PRIMARY DATABASE` on a standby database.

PHYSICAL

This clause is always optional. Use of this clause with the `COMMIT TO SWITCHOVER TO PRIMARY` clause has been deprecated.

LOGICAL

This clause is specified with the `PREPARE TO SWITCHOVER` or `COMMIT TO SWITCHOVER` clauses when switching a primary database to the logical standby database role. Use of this clause with the `COMMIT TO SWITCHOVER TO PRIMARY` clause has been deprecated.

WITH SESSION SHUTDOWN

This clause causes all database sessions to be closed and uncommitted transactions to be rolled back before performing a database role transition.

WITHOUT SESSION SHUTDOWN

This clause prevents a requested role transition from occurring if there are any database sessions. This is the default.

WAIT

Specify this clause to wait for a role transition to complete before returning control to the user.

NOWAIT

Specify this clause to return control to the user without waiting for a role transition to complete. This is the default.

CANCEL

Specify this clause to reverse the effect of a previously specified `PREPARE TO SWITCHOVER` statement.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for full information on switchover between primary and standby databases

start_standby_clause

Specify the `START LOGICAL STANDBY APPLY` clause to begin applying redo logs to a logical standby database. This clause enables primary key, unique index, and unique constraint supplemental logging as well as PL/SQL call logging.

- Specify `IMMEDIATE` to apply redo data from the current standby redo log file.
- Specify `NODELAY` if you want Oracle Database to ignore a delay for this apply. This is useful if the primary database is no longer present, which would otherwise require a PL/SQL call to be made.
- Specify `INITIAL` the first time you apply the logs to the standby database.
- The `NEW PRIMARY` clause is needed in two situations:
  - On a failover to a logical standby, specify this clause on a logical standby not participating in the failover operation, and on the old primary database after it has been reinstated as a logical standby database.
  - During a rolling upgrade using a logical standby database (which uses an unprepared switchover operation), specify this clause after the original primary database has been upgraded to the new database software.
- Specify `SKIP FAILED` [`TRANSACTION`] to skip the last transaction in the events table and restart the apply.
- Specify `FINISH` to force the standby redo logfile information into archived logs. If the primary database becomes disabled, then you can then apply the data in the redo log files.

stop_standby_clause

Use this clause to stop the log apply services. This clause applies only to logical standby databases, not to physical standby databases. Use the `STOP` clause to stop the apply in an orderly fashion.

convert_database_clause

Use this clause to convert a database from one form to another.

- Specify `CONVERT TO PHYSICAL STANDBY` to convert a primary database, a logical standby database, or a snapshot standby database into a physical standby database.

  Perform these steps before specifying this clause:
  - On an Oracle Real Application Clusters (Oracle RAC) database, shut down all but one instance.
  - Ensure that the database is mounted, but not open.

  The database is dismounted after conversion and must be restarted.
- Specify `CONVERT TO SNAPSHOT STANDBY` to convert a physical standby database into a snapshot standby database.

  Ensure that redo apply is stopped before specifying this clause.

> **Note:**
>
> A snapshot standby database must be opened at least once in read/write mode before it can be converted into a physical standby database.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information about standby databases

default_settings_clauses

Use these clauses to modify the default settings of the database.

DEFAULT EDITION Clause

Use this clause to designate the specified edition as the default edition for the database. The specified edition must already have been created and must be `USABLE`. The change takes place immediately and is visible to all nodes in an Oracle RAC environment. New database sessions automatically start out in the specified edition. The new setting persists across database shutdown and startup.

When you designate an edition as the database default edition, all users can use the edition, as though the `USE` object privilege were granted on the specified edition to the role `PUBLIC`.

You can determine the current default edition of the database with the following query:

```
SELECT PROPERTY_VALUE FROM DATABASE_PROPERTIES
  WHERE PROPERTY_NAME = 'DEFAULT_EDITION';
```

> **See Also:**
>
> CREATE EDITION for more information on editions and *Oracle Database PL/SQL Language Reference* for information on how editions are designated as `USABLE`

CHARACTER SET, NATIONAL CHARACTER SET

You can no longer change the database character set or the national character set using the `ALTER DATABASE` statement. Refer to *Oracle Database Globalization Support Guide* for information on database character set migration.

SET DEFAULT TABLESPACE Clause

Use this clause to specify or change the default type of subsequently created tablespaces. Specify `BIGFILE` or `SMALLFILE` to indicate whether the tablespaces should be bigfile or smallfile tablespaces.

- A **bigfile tablespace** contains only one data file or temp file, which can contain up to approximately 4 billion (2^{32}) blocks. The maximum size of the single data file or temp file is 128 terabytes (TB) for a tablespace with 32K blocks and 32TB for a tablespace with 8K blocks.

- A **smallfile tablespace** is a traditional Oracle tablespace, which can contain 1022 data files or temp files, each of which can contain up to approximately 4 million (2^{22}) blocks.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about bigfile tablespaces
> - "Setting the Default Type of Tablespaces: Example"

DEFAULT TABLESPACE Clause

Specify this clause to establish or change the default permanent tablespace of the database. The tablespace you specify must already have been created. After this operation completes, Oracle Database automatically reassigns to the new default tablespace all non-`SYSTEM` users. All objects subsequently created by those users will by default be stored in the new default tablespace. If you are replacing a previously specified default tablespace, then you can move the previously created objects from the old to the new default tablespace, and then drop the old default tablespace if you want to.

DEFAULT [LOCAL] TEMPORARY TABLESPACE Clause

Specify this clause to change the default shared temporary tablespace of the database to a new tablespace or tablespace group, or to change the default local temporary tablespace to a new tablespace.

- Specify *`tablespace`* to indicate the new default temporary tablespace for the database. After this operation completes, Oracle Database automatically reassigns to the new default temporary tablespace all users who had been assigned to the old default temporary tablespace. You can then drop the old default temporary tablespace if you want to. Specify `DEFAULT TEMPORARY TABLESPACE` to change the default shared temporary tablespace. Specify `DEFAULT LOCAL TEMPORARY TABLESPACE` to change the default local temporary tablespace.
- Specify *`tablespace_group_name`* to indicate that all tablespaces in the tablespace group specified by *`tablespace_group_name`* are now default shared temporary tablespaces for the database. After this operation completes, users who have not been explicitly assigned a default temporary tablespace can create temporary segments in any of the tablespaces that are part of *`tablespace_group_name`*. You cannot drop an old default temporary tablespace if it is part of the default temporary tablespace group. Local temporary tablespaces cannot be part of a tablespace group.

To learn the name of the current default temporary tablespace or default temporary tablespace group, query the `TEMPORARY_TABLESPACE` column of the `ALL_`, `DBA_`, or `USER_USERS` data dictionary views.

Restrictions on Default Temporary Tablespaces

Default temporary tablespaces are subject to the following restrictions:

- The tablespace you assign or reassign as the default temporary tablespace must have a standard block size.

- If the `SYSTEM` tablespace is locally managed, then the tablespace you specify as the default temporary tablespace must also be locally managed.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on tablespace groups
> - "Changing the Default Temporary Tablespace: Examples"

instance_clauses

In an Oracle Real Application Clusters environment, specify `ENABLE INSTANCE` to enable the thread that is mapped to the specified database instance. The thread must have at least two redo log file groups, and the database must be open.

Specify `DISABLE INSTANCE` to disable the thread that is mapped to the specified database instance. The name of the instance is a string of up to 80 characters. If no thread is currently mapped to the specified instance, then Oracle Database returns an error. The database must be open, but you cannot disable a thread if an instance using it has the database mounted.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information on enabling and disabling instances

RENAME GLOBAL_NAME Clause

Specify `RENAME GLOBAL_NAME` to change the global name of the database. The database must be open. The *`database`* is the new database name and can be as long as eight bytes. The optional *`domain`* specifies where the database is effectively located in the network hierarchy. If you specify a domain name, then the components of the domain name must be legal identifiers. See "Database Object Naming Rules " for information on valid identifiers.

> **Note:**
>
> Renaming your database does not change global references to your database from existing database links, synonyms, and stored procedures and functions on remote databases. Changing such references is the responsibility of the administrator of the remote databases.

> **See Also:**
>
> "Changing the Global Database Name: Example"

BLOCK CHANGE TRACKING Clauses

The **block change tracking** feature causes Oracle Database to keep track of the physical locations of all database updates on both the primary database and any physical standby database. You must enable block change tracking on each database for which you want tracking to be performed. The tracking information is maintained in a separate file called the block change tracking file. If you are using Oracle Managed Files, then Oracle Database automatically creates the block change tracking file in the location specified by `DB_CREATE_FILE_DEST`. If you are not using Oracle Managed Files, then you must specify the change tracking filename. Oracle Database uses change tracking data for some internal tasks, such as increasing the performance of incremental backups. You can enable or disable block change tracking with the database either open or mounted, in either archivelog or `NOARCHIVELOG` mode.

ENABLE BLOCK CHANGE TRACKING

This clause enables block change tracking and causes Oracle Database to create a block change tracking file.

- Specify `USING FILE` '*filename*' if you want to name the block change tracking file instead of letting Oracle Database generate a name for it. You must specify this clause if you are not using Oracle Managed Files.
- Specify `REUSE` to allow Oracle Database to overwrite an existing block change tracking file of the same name.

DISABLE BLOCK CHANGE TRACKING

Specify this clause if you want Oracle Database to stop tracking changes and delete the existing block change tracking file.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for information on setting up block change tracking and "Enabling and Disabling Block Change Tracking: Examples"

[NO] FORCE FULL DATABASE CACHING

Use this clause to enable or disable the force full database caching mode. In contrast to the default mode, which is automatic, the force full database caching mode considers the entire database, including `NOCACHE` LOBs, as eligible for caching in the buffer cache.

The database must be mounted but not open. In an Oracle RAC environment, the database must be mounted but not open in the current instance and unmounted in all other instances.

- Specify `FORCE FULL DATABASE CACHING` to enable the force full database caching mode.
- Specify `NO FORCE FULL DATABASE CACHING` to disable the force full database caching mode. This is the default mode.

You can determine whether the force full database caching mode is enabled by querying the `FORCE_FULL_DB_CACHING` column of the `V$DATABASE` dynamic performance view.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on the force full database caching mode
> - *Oracle Database Administrator's Guide* to learn how to enable the force full database caching mode
> - *Oracle Database Reference* for more information on the `V$DATABASE` dynamic performance view

CONTAINERS DEFAULT TARGET

Use this clause to specify the default container for DML statements in a CDB. You must be connect to the CDB root.

- For *`container_name`*, specify the name of the default container. The default container can be any container in the CDB, including the CDB root, a PDB, an application root, or an application PDB. You can specify only one default container.
- If you specify `NONE`, then the default container is the CDB root. This is the default.

When a DML statement is issued in a CDB root without specifying containers in the `WHERE` clause, the DML statement affects the default container for the CDB.

flashback_mode_clause

Use this clause to put the database in or take the database out of `FLASHBACK` mode. You can specify this clause only if the database is in `ARCHIVELOG` mode and you have already prepared a fast recovery area for the database. You can specify this clause when the database is mounted or open. This clause cannot be specified on a physical standby database if redo apply is active.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for information on preparing the fast recovery area for Flashback operations

FLASHBACK ON

Use this clause to put the database in `FLASHBACK` mode. When the database is in `FLASHBACK` mode, Oracle Database automatically creates and manages Flashback Database logs in the fast recovery area. Users with `SYSDBA` system privilege can then issue a `FLASHBACK DATABASE` statement.

FLASHBACK OFF

Use this clause to take the database out of `FLASHBACK` mode. Oracle Database stops logging Flashback data and deletes all existing Flashback Database logs. Any attempt to issue a `FLASHBACK DATABASE` will fail with an error.

undo_mode_clause

This clause is valid only when you are connected to a CDB. It lets you change the undo mode for the CDB. The CDB must be in `OPEN UPGRADE` mode.

- Specify `LOCAL UNDO ON` to change the CDB to use local undo mode.
- Specify `LOCAL UNDO OFF` to change the CDB to use shared undo mode.

> **See Also:**
>
> - `CREATE DATABASE` *undo_mode_clause* for the full semantics of this clause
> - *Oracle Database Administrator's Guide* for the complete steps for configuring a CDB to use local undo mode or shared undo mode

set_time_zone_clause

This clause has the same semantics in `CREATE DATABASE` and `ALTER DATABASE` statements. When used in with `ALTER DATABASE`, this clause resets the time zone of the database. To determine the time zone of the database, query the built-in function DBTIMEZONE . After setting or changing the time zone with this clause, you must restart the database for the new time zone to take effect.

Oracle Database normalizes all new `TIMESTAMP WITH LOCAL TIME ZONE` data to the time zone of the database when the data is stored on disk.Oracle Database does not automatically update existing data in the database to the new time zone. Therefore, you cannot reset the database time zone if there is any `TIMESTAMP WITH LOCAL TIME ZONE` data in the database. You must first delete or export the `TIMESTAMP WITH LOCAL TIME ZONE` data and then reset the database time zone. For this reason, Oracle does not encourage you to change the time zone of a database that contains data.

For a full description of this clause, refer to *set_time_zone_clause* in the documentation on `CREATE DATABASE`.

security_clause

Use the *`security_clause`* (`GUARD`) to protect data in the database from being changed. You can override this setting for a current session using the `ALTER SESSION DISABLE GUARD` statement. Refer to ALTER SESSION for more information.

ALL

Specify `ALL` to prevent all users other than `SYS` from making any changes to the database.

STANDBY

Specify `STANDBY` to prevent all users other than `SYS` from making changes to any database object being maintained by logical standby. This setting is useful if you want report operations to be able to modify data as long as it is not being replicated by logical standby.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for information on logical standby

NONE

Specify `NONE` if you want normal security for all data in the database.

> **Note:**
>
> Oracle strongly recommends that you not use this setting on a logical standby database.

Examples

READ ONLY / READ WRITE: Example

The following statement opens the database in read-only mode:

```
ALTER DATABASE OPEN READ ONLY;
```

The following statement opens the database in read/write mode and clears the online redo logs:

```
ALTER DATABASE OPEN READ WRITE RESETLOGS;
```

Using Parallel Recovery Processes: Example

The following statement performs tablespace recovery using parallel recovery processes:

```
ALTER DATABASE
   RECOVER TABLESPACE tbs_03
   PARALLEL;
```

Adding Redo Log File Groups: Examples

The following statement adds a redo log file group with two members and identifies it with a `GROUP` parameter value of 3:

```
ALTER DATABASE
  ADD LOGFILE GROUP 3
    ('diska:log3.log' ,
     'diskb:log3.log') SIZE 50K;
```

The following statement adds a redo log file group containing two members to thread 5 (in a Real Application Clusters environment) and assigns it a `GROUP` parameter value of 4:

```
ALTER DATABASE
    ADD LOGFILE THREAD 5 GROUP 4
        ('diska:log4.log',
         'diskb:log4:log');
```

Adding Redo Log File Group Members: Example

The following statement adds a member to the redo log file group added in the previous example:

```
ALTER DATABASE
   ADD LOGFILE MEMBER 'diskc:log3.log'
   TO GROUP 3;
```

Dropping Log File Members: Example

The following statement drops one redo log file member added in the previous example:

```
ALTER DATABASE
    DROP LOGFILE MEMBER 'diskb:log3.log';
```

The following statement drops all members of the redo log file group 3:

```
ALTER DATABASE DROP LOGFILE GROUP 3;
```

Renaming a Log File Member: Example

The following statement renames a redo log file member:

```
ALTER DATABASE
    RENAME FILE 'diskc:log3.log' TO 'diskb:log3.log';
```

The preceding statement only changes the member of the redo log group from one file to another. The statement does not actually change the name of the file `diskc:log3.log` to `diskb:log3.log`. Before issuing this statement, you must change the name of the file through your operating system.

Setting the Default Type of Tablespaces: Example

The following statement specifies that subsequently created tablespaces be created as bigfile tablespaces by default:

```
ALTER DATABASE
    SET DEFAULT BIGFILE TABLESPACE;
```

Changing the Default Temporary Tablespace: Examples

The following statement makes the `tbs_05` tablespace (created in "Creating a Temporary Tablespace: Example") the default temporary tablespace of the database. This statement either establishes a default temporary tablespace if none was specified at create time, or replaces an existing default temporary tablespace with `tbs_05`:

```
ALTER DATABASE
   DEFAULT TEMPORARY TABLESPACE tbs_05;
```

Alternatively, a group of tablespaces can be defined as the default temporary tablespace by using a tablespace group. The following statement makes the tablespaces in the tablespace group `tbs_group_01` (created in "Adding a Temporary Tablespace to a Tablespace Group: Example") the default temporary tablespaces of the database:

```
ALTER DATABASE
   DEFAULT TEMPORARY TABLESPACE tbs_grp_01;
```

Creating a New Data File: Example

The following statement creates a new data file `tbs_f04.dbf` based on the file `tbs_f03.dbf`. Before creating the new data file, you must take the existing data file (or the tablespace in which it resides) offline.

```
ALTER DATABASE
    CREATE DATAFILE 'tbs_f03.dbf'
                 AS 'tbs_f04.dbf';
```

Manipulating Temp Files: Example

The following takes offline the temp file temp02.dbf created in Adding and Dropping Data Files and Temp Files: Examples and then renames the temp file:

```
ALTER DATABASE TEMPFILE 'temp02.dbf' OFFLINE;

ALTER DATABASE RENAME FILE 'temp02.dbf' TO 'temp03.dbf';
```

The statement renaming the temp file requires that you first create the file `temp03.dbf` on the operating system.

Changing the Global Database Name: Example

The following statement changes the global name of the database and includes both the database name and domain:

```
ALTER DATABASE
    RENAME GLOBAL_NAME TO demo.world.example.com;
```

Enabling and Disabling Block Change Tracking: Examples

The following statement enables block change tracking and causes Oracle Database to create a block change tracking file named `tracking_file` and overwrite the file if it already exists:

```
ALTER DATABASE
  ENABLE BLOCK CHANGE TRACKING
    USING FILE 'tracking_file' REUSE;
```

The following statement disables block change tracking and deletes the existing block change tracking file:

```
ALTER DATABASE
  DISABLE BLOCK CHANGE TRACKING;
```

Resizing a Data File: Example

The following statement attempts to change the size of data file `diskb:tbs_f5.dbf`:

```
ALTER DATABASE
    DATAFILE 'diskb:tbs_f5.dbf' RESIZE 10 M;
```

Clearing a Log File: Example

The following statement clears a log file:

```
ALTER DATABASE
    CLEAR LOGFILE 'diskc:log3.log';
```

Database Recovery: Examples

The following statement performs complete recovery of the entire database, letting Oracle Database generate the name of the next archived redo log file needed:

```
ALTER DATABASE
  RECOVER AUTOMATIC DATABASE;
```

The following statement explicitly names a redo log file for Oracle Database to apply:

```
ALTER DATABASE
    RECOVER LOGFILE 'diskc:log3.log';
```

The following statement performs time-based recovery of the database:

```
ALTER DATABASE
    RECOVER AUTOMATIC UNTIL TIME '2001-10-27:14:00:00';
```

Oracle Database recovers the database until 2:00 p.m. on October 27, 2001.

For an example of recovering a tablespace, see "Using Parallel Recovery Processes: Example".

ALTER DATABASE LINK

Purpose

Use the `ALTER DATABASE LINK` statement to modify a fixed-user database link when the password of the connection or authentication user changes.

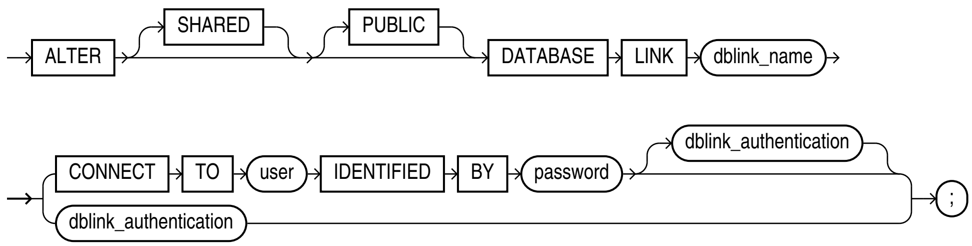
> **Note:**
>
> - You cannot use this statement to change the connection or authentication user associated with the database link. To change *`user`*, you must re-create the database link.
> - You cannot use this statement to change the password of a connection or authentication user. You must use the ALTER USER statement for this purpose, and then alter the database link with the `ALTER DATABASE LINK` statement.
> - This statement is valid only for fixed-user database links, not for connected-user or current user database links. See CREATE DATABASE LINK for more information on these two types of database links.

Prerequisites

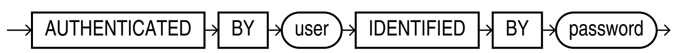
To alter a private database link, you must have the `ALTER DATABASE LINK` system privilege. To alter a public database link, you must have the `ALTER PUBLIC DATABASE LINK` system privilege.

Syntax

***alter_database_link*::=**



dblink_authentication



Semantics

The `ALTER DATABASE LINK` statement is intended only to update fixed-user database links with the current passwords of connection and authentication users. Therefore, any clauses valid in a `CREATE DATABASE LINK` statement that do not appear in the syntax diagram above are not valid in an `ALTER DATABASE LINK` statement. The semantics of all of the clauses permitted in this statement are the same as the semantics for those clauses in `CREATE DATABASE LINK`. Refer to CREATE DATABASE LINK for this information.

Examples

The following statements show the valid variations of the `ALTER DATABASE LINK` statement:

```
ALTER DATABASE LINK private_link
  CONNECT TO hr IDENTIFIED BY hr_new_password;

ALTER PUBLIC DATABASE LINK public_link
  CONNECT TO scott IDENTIFIED BY scott_new_password;

ALTER SHARED PUBLIC DATABASE LINK shared_pub_link
  CONNECT TO scott IDENTIFIED BY scott_new_password
  AUTHENTICATED BY hr IDENTIFIED BY hr_new_password;

ALTER SHARED DATABASE LINK shared_pub_link
  CONNECT TO scott IDENTIFIED BY scott_new_password;
```

ALTER DIMENSION

Purpose

Use the `ALTER DIMENSION` statement to change the hierarchical relationships or dimension attributes of a dimension.

> **See Also:**
>
> CREATE DIMENSION and DROP DIMENSION

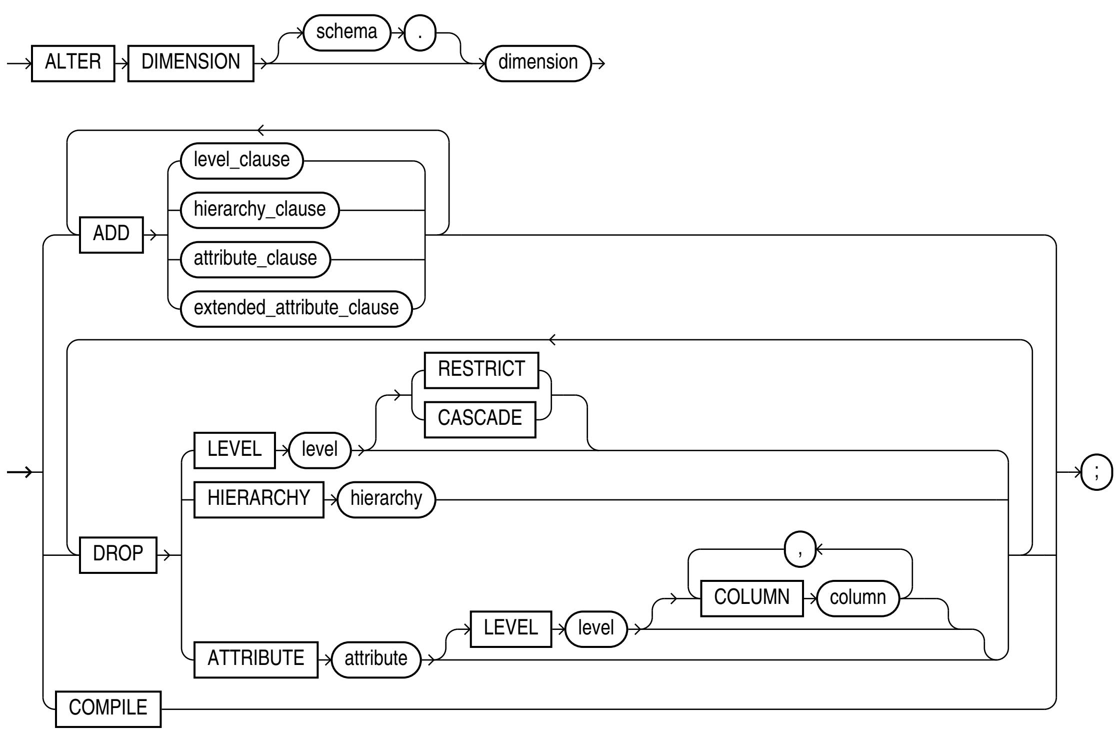
Prerequisites

The dimension must be in your schema or you must have the `ALTER ANY DIMENSION` system privilege to use this statement.

A dimension is always altered under the rights of the owner.

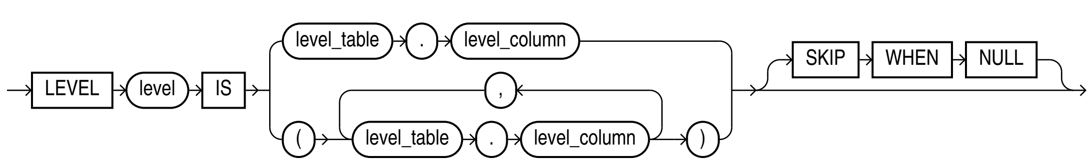
Syntax

***alter_dimension*::=**

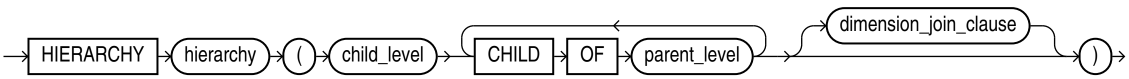


(*level_clause*::=, *hierarchy_clause*::=, *attribute_clause*::=, *extended_attribute_clause*::=)

***level_clause*::=**

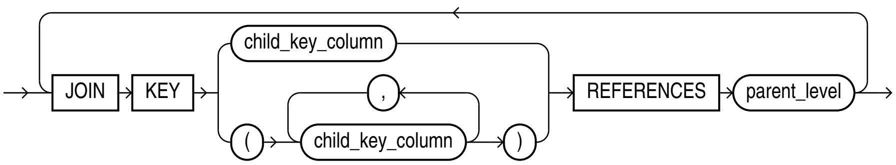


***hierarchy_clause*::=**

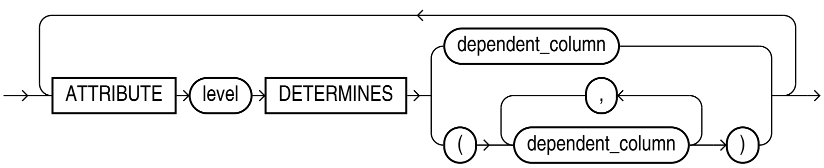


(*dimension_join_clause*::=)

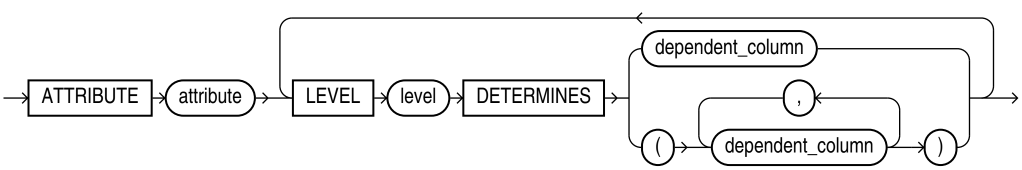
***dimension_join_clause*::=**



***attribute_clause*::=**



***extended_attribute_clause*::=**



Semantics

The following keywords, parameters, and clauses have meaning unique to `ALTER DIMENSION`. Keywords, parameters, and clauses that do not appear here have the same functionality that they have in the `CREATE DIMENSION` statement. Refer to CREATE DIMENSION for more information.

schema

Specify the schema of the dimension you want to modify. If you do not specify *`schema`*, then Oracle Database assumes the dimension is in your own schema.

dimension

Specify the name of the dimension. This dimension must already exist.

ADD

The `ADD` clauses let you add a level, hierarchy, or attribute to the dimension. Adding one of these elements does not invalidate any existing materialized view.

Oracle Database processes `ADD LEVEL` clauses prior to any other `ADD` clauses.

DROP

The `DROP` clauses let you drop a level, hierarchy, or attribute from the dimension. Any level, hierarchy, or attribute you specify must already exist.

Within one attribute, you can drop one or more level-to-column relationships associated with one level.

Restriction on DROP

If any attributes or hierarchies reference a level, then you cannot drop the level until you either drop all the referencing attributes and hierarchies or specify `CASCADE`.

CASCADE

Specify `CASCADE` if you want Oracle Database to drop any attributes or hierarchies that reference the level, along with the level itself.

RESTRICT

Specify `RESTRICT` if you want to prevent Oracle Database from dropping a level that is referenced by any attributes or hierarchies. This is the default.

COMPILE

Specify `COMPILE` to explicitly recompile an invalidated dimension. Oracle Database automatically compiles a dimension when you issue an `ADD` clause or `DROP` clause. However, if you alter an object referenced by the dimension (for example, if you drop and then re-create a table referenced in the dimension), Oracle Database invalidates, and you must recompile it explicitly.

Examples

Modifying a Dimension: Examples

The following examples modify the `customers_dim` dimension in the sample schema `sh`:

```
ALTER DIMENSION customers_dim
   DROP ATTRIBUTE country;

ALTER DIMENSION customers_dim
   ADD LEVEL zone IS customers.cust_postal_code
   ADD ATTRIBUTE zone DETERMINES (cust_city);
```

ALTER DISKGROUP

> **Note:**
>
> This SQL statement is valid only if you are using Oracle ASM and you have started an Oracle ASM instance. You must issue this statement from within the Oracle ASM instance, not from a normal database instance. For information on starting an Oracle ASM instance, refer to *Oracle Automatic Storage Management Administrator's Guide*.

Purpose

The `ALTER DISKGROUP` statement lets you perform a number of operations on a disk group or on the disks in a disk group.

> **See Also:**
>
> - CREATE DISKGROUP for information on creating disk groups
> - *Oracle Automatic Storage Management Administrator's Guide* for information on Oracle ASM and using disk groups to simplify database administration

Prerequisites

You must have an Oracle ASM instance started from which you issue this statement. The disk group to be modified must be mounted.

You can issue all `ALTER DISKGROUP` clauses if you have the `SYSASM` system privilege. You can issue specific clauses as follows:

- The `SYSOPER` privilege permits the following subset of the `ALTER DISKGROUP` operations: *`diskgroup_availability`*, *`rebalance_diskgroup_clause`*, *`check_diskgroup_clause`* (without the `REPAIR` option).
- If you are connected as `SYSDBA`, you have limited privileges to use this statement. The following operations are always granted to users connected as `SYSDBA`:
  - `ALTER DISKGROUP` ... `ADD DIRECTORY`
  - `ALTER DISKGROUP` ... `ADD`/`ALTER`/`DROP TEMPLATE` (nonsystem templates only)
  - `ALTER DISKGROUP` ... `ADD USERGROUP`
  - `SELECT`
  - `SHOW PARAMETER`

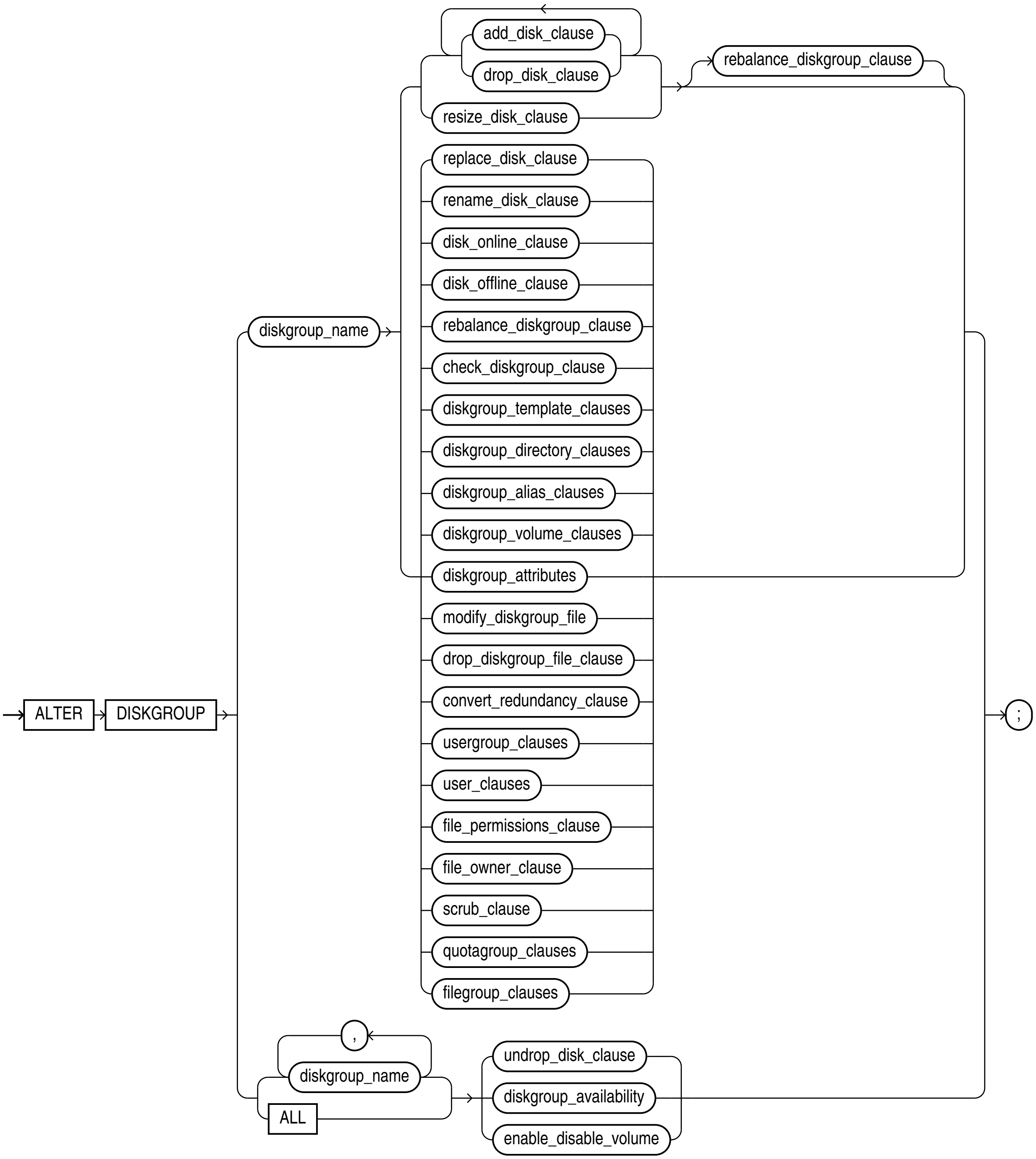
Table 10-1 shows additional privileges granted to users connected as `SYSDBA` under the conditions shown:

Table 10-1 Conditional Diskgroup Privileges for SYSDBA

ALTER DISKGROUP Operation	Condition
DROP FILE	User must have read-write permission on the file.
ADD ALIAS	User must have read-write permission on the related file.
RENAME ALIAS	User must have read-write permission on the related file.
DROP ALIAS	User must have read-write permission on the related file.
RENAME DIRECTORY	Directory must contain only aliases and no files. User must have DROP ALIAS permissions on all aliases under the directory.
DROP DIRECTORY	Directory must contain only aliases and no files. User must have DROP ALIAS permissions on all aliases under the directory.
DROP USERGROUP	User must be the owner of the user group.
MODIFY FILE	User must be the owner of the file.
MODIFY USERGROUP ADD MEMBER	User must be the owner of the user group.
MODIFY USERGROUP DROP MEMBER	User must be the owner of the user group.
SET PERMISSION	User must be the owner of the file.
SET OWNER GROUP	User must be the owner of the file and a member of the user group.

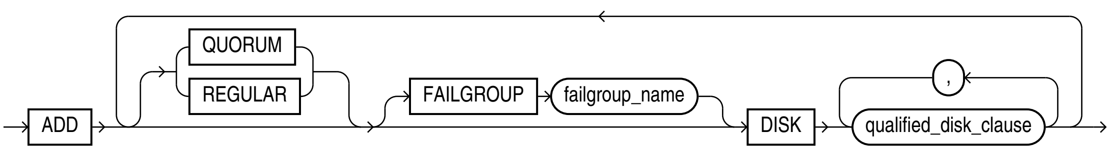
Syntax

***alter_diskgroup*::=**



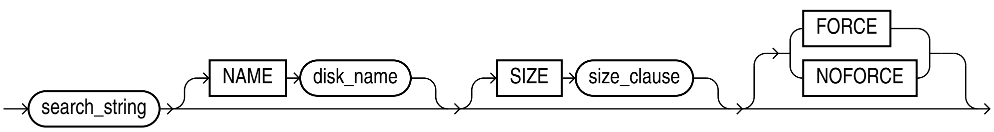
(*add_disk_clause*::=, *drop_disk_clause*::=, *resize_disk_clause*::=, *replace_disk_clause*::=, *rename_disk_clause*::=, *disk_online_clause*::=, *disk_offline_clause*::=, *rebalance_diskgroup_clause*::=, *check_diskgroup_clause*::=, *diskgroup_template_clauses*::=, *diskgroup_directory_clauses*::=, *diskgroup_alias_clauses*::=, *diskgroup_volume_clauses*::=, *diskgroup_attributes*::=, *modify_diskgroup_file*::=, *drop_diskgroup_file_clause*::=, *convert_redundancy_clause*::=, *usergroup_clauses*::=, *user_clauses*::=, *file_permissions_clause*::=, *file_owner_clause*::=, *scrub_clause*::=, *quotagroup_clauses*::=, *filegroup_clauses*::=, *undrop_disk_clause*::=, *diskgroup_availability*::=, *enable_disable_volume*::=)

***add_disk_clause*::=**



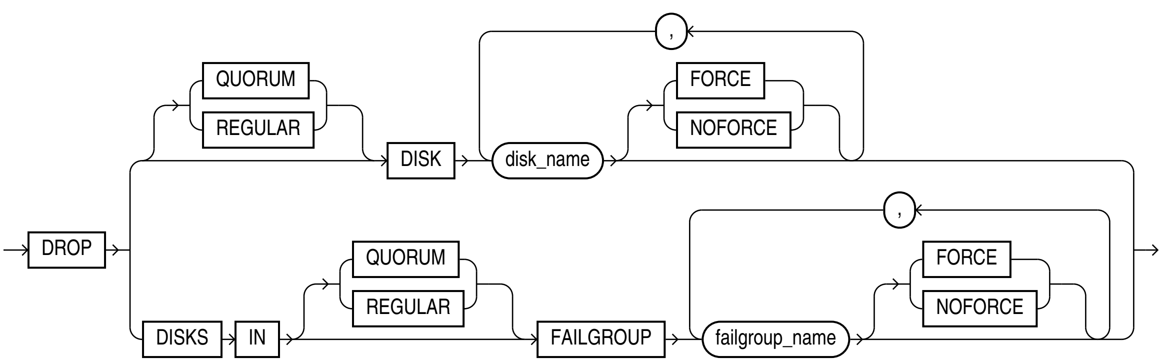
(*qualified_disk_clause*::=)

***qualified_disk_clause*::=**

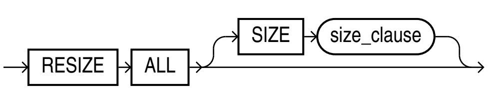


(*size_clause*::=)

***drop_disk_clause*::=**

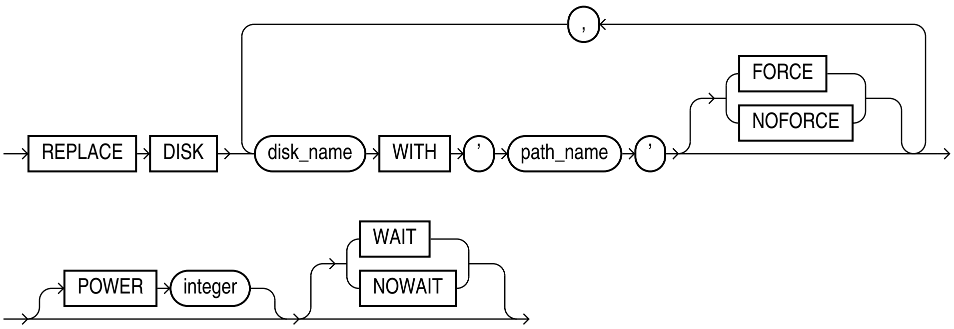


***resize_disk_clause*::=**

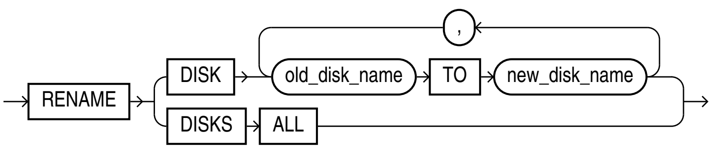


(*size_clause*::=)

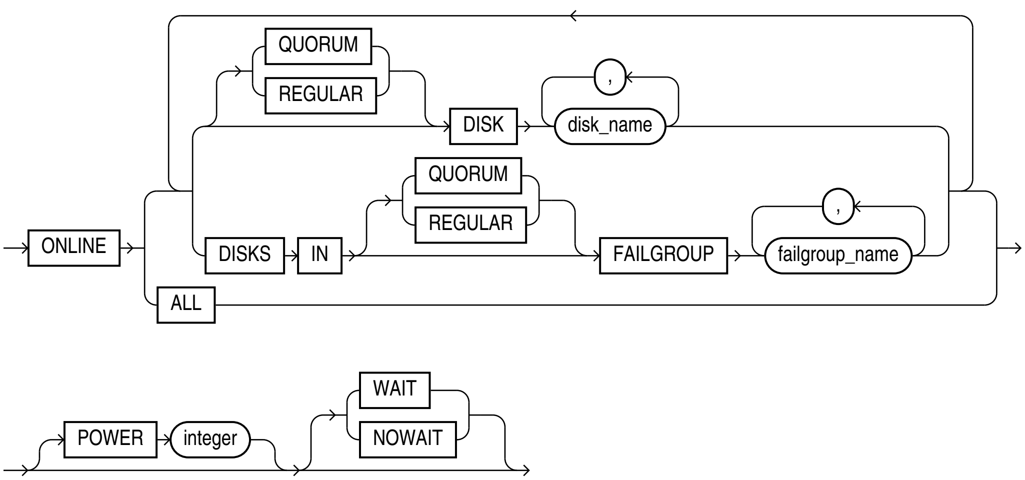
***replace_disk_clause*::=**



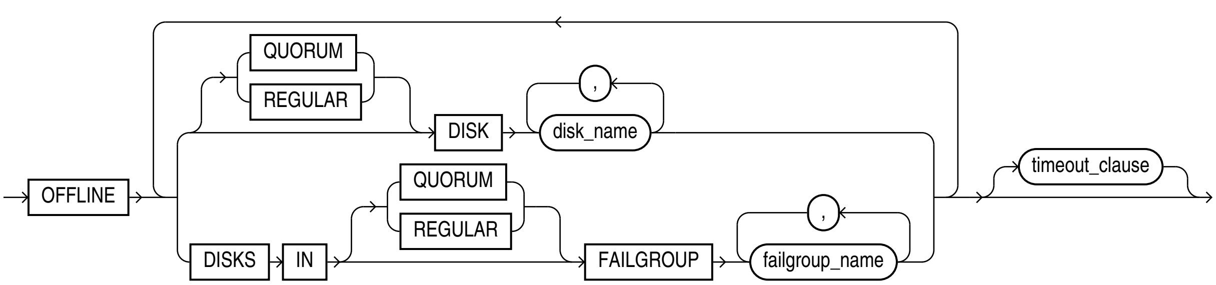
***rename_disk_clause*::=**



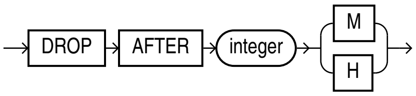
***disk_online_clause*::=**



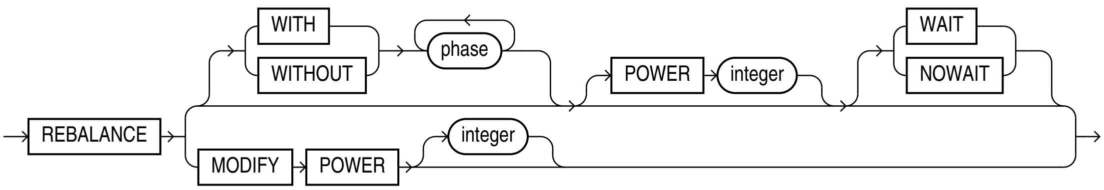
***disk_offline_clause*::=**



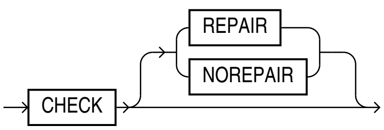
***timeout_clause*::=**



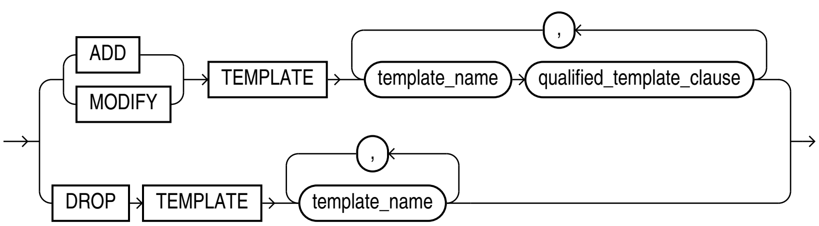
***rebalance_diskgroup_clause*::=**



***check_diskgroup_clause*::=**

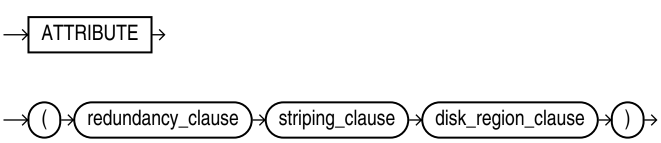


***diskgroup_template_clauses*::=**

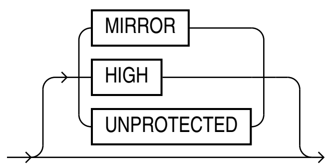


(*qualified_template_clause*::=)

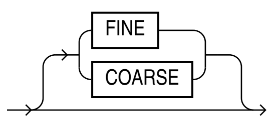
***qualified_template_clause*::=**



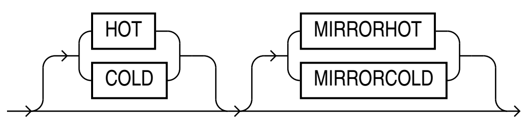
***redundancy_clause*::=**



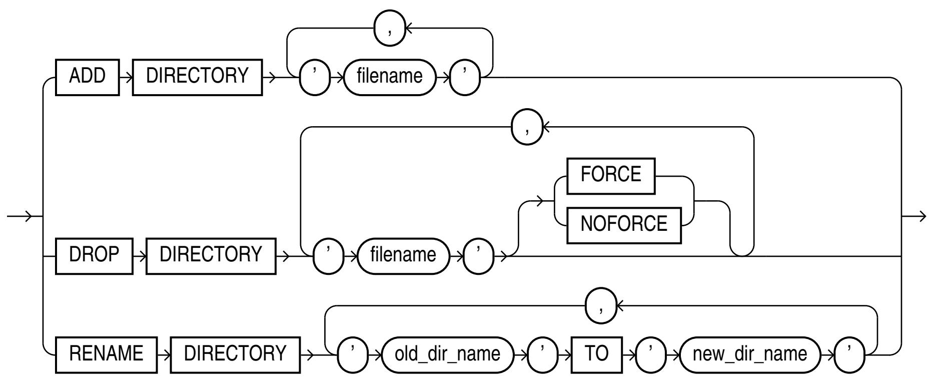
***striping_clause*::=**



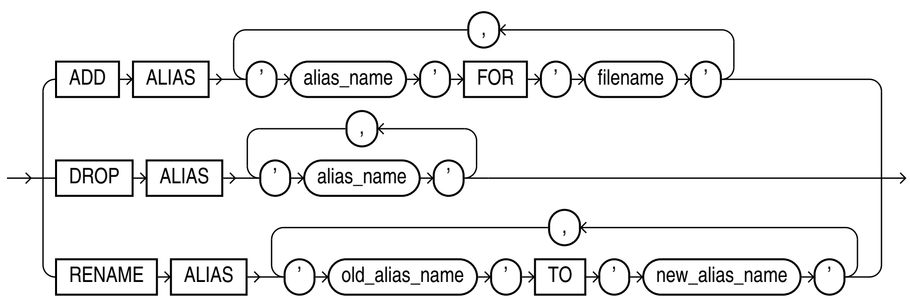
***disk_region_clause*::=**



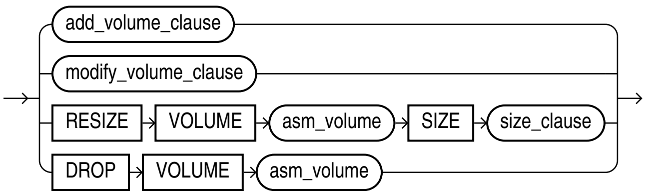
***diskgroup_directory_clauses*::=**



***diskgroup_alias_clauses*::=**

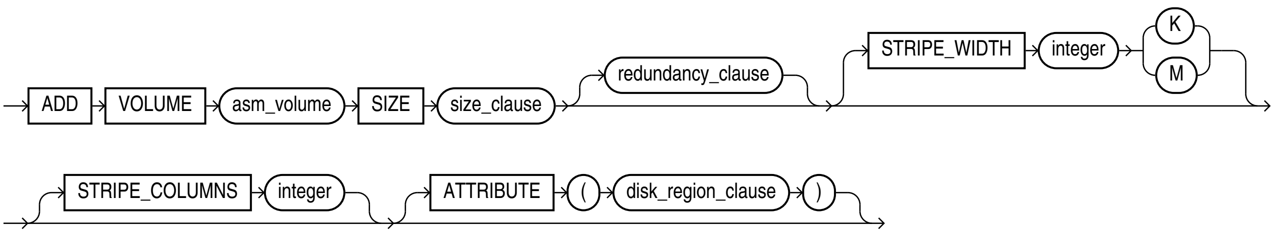


***diskgroup_volume_clauses*::=**



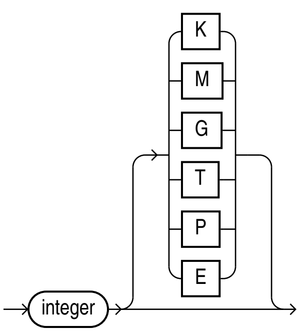
(*add_volume_clause*::=, *modify_volume_clause*::=

***add_volume_clause*::=**

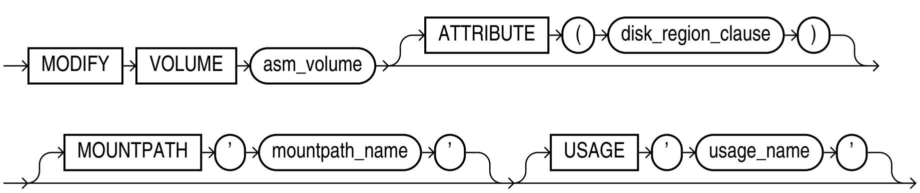


(*size_clause*::=, *redundancy_clause*::=, *disk_region_clause*::=)

***size_clause*::=**

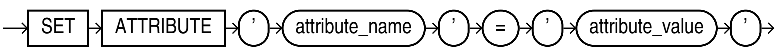


***modify_volume_clause*::=**

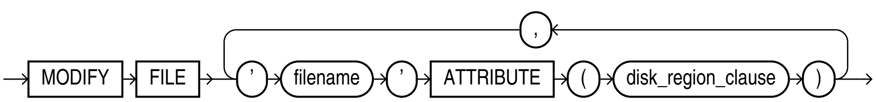


(*disk_region_clause*::=)

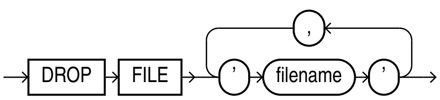
***diskgroup_attributes*::=**



***modify_diskgroup_file*::=**

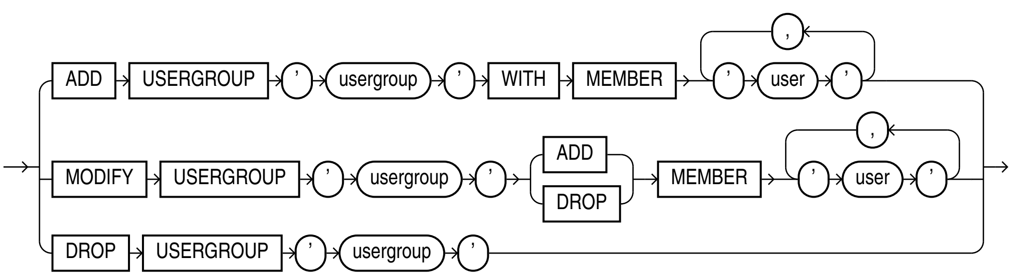


***drop_diskgroup_file_clause*::=**

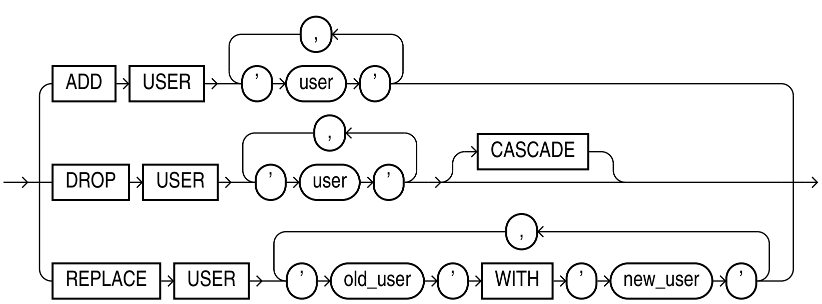


***convert_redundancy_clause*::=**

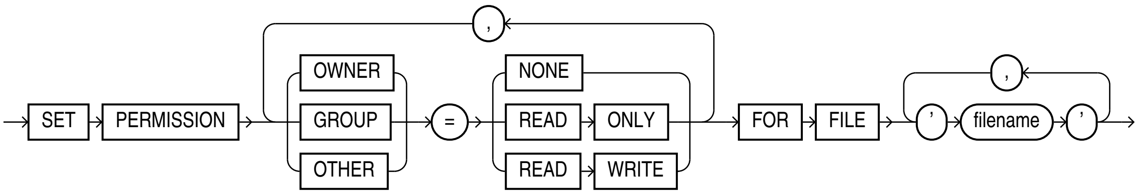
***usergroup_clauses*::=**



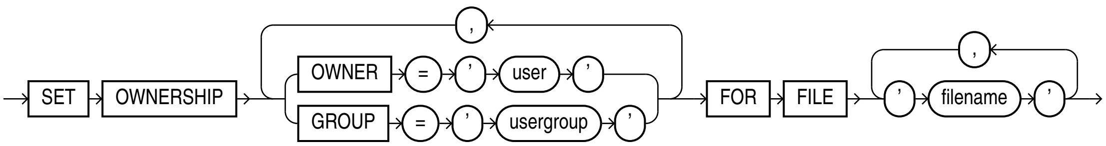
***user_clauses*::=**



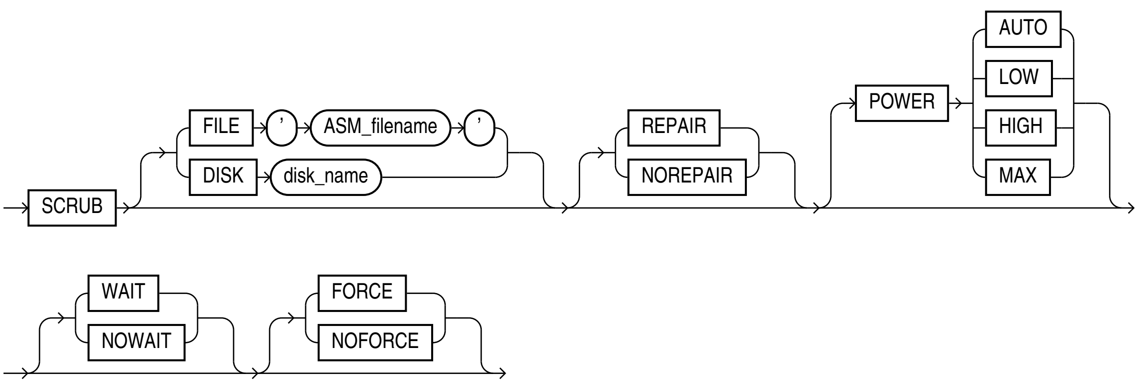
***file_permissions_clause*::=**



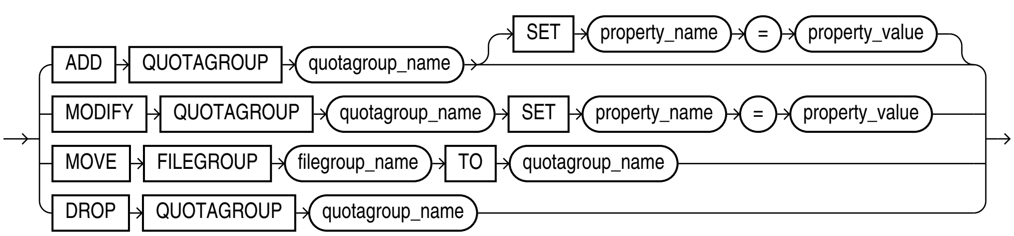
***file_owner_clause*::=**



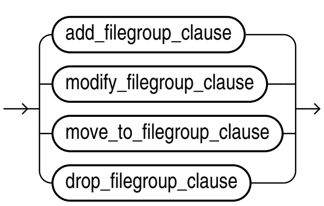
***scrub_clause*::=**



***quotagroup_clauses*::=**

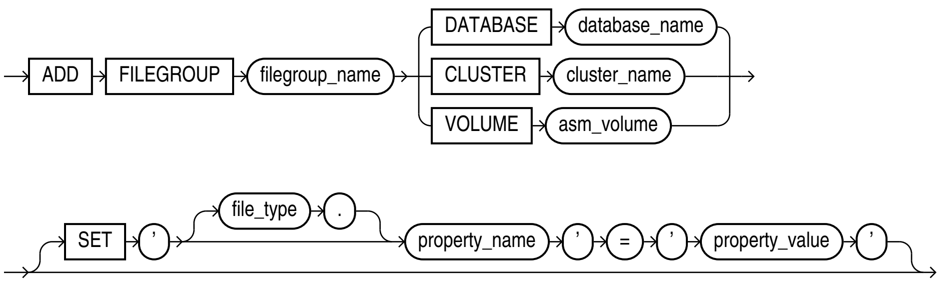


***filegroup_clauses*::=**

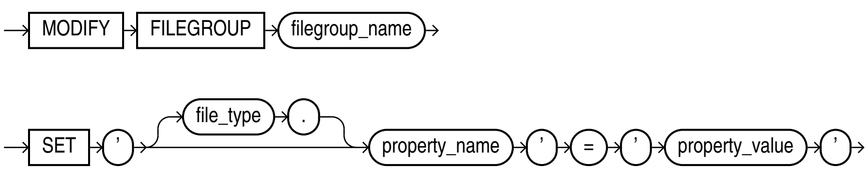


(add_filegroup_clause::=, modify_filegroup_clause::=, move_to_filegroup_clause::=, drop_filegroup_clause::=)

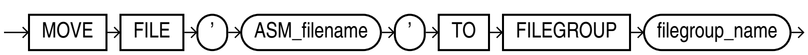
***add_filegroup_clause*::=**



***modify_filegroup_clause*::=**

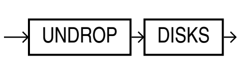


***move_to_filegroup_clause*::=**

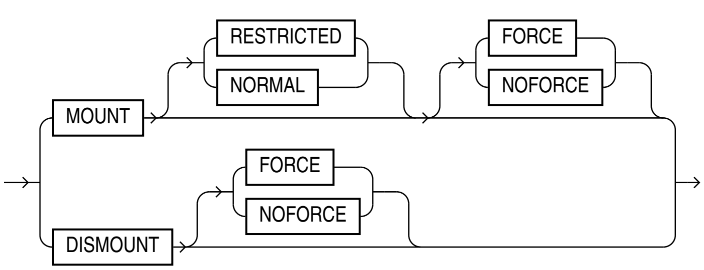


***drop_filegroup_clause*::=**

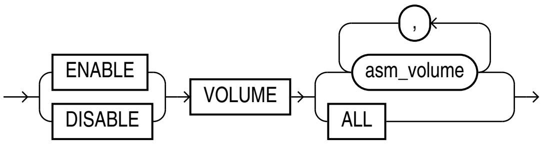
***undrop_disk_clause*::=**



***diskgroup_availability*::=**



***enable_disable_volume*::=**



Semantics

diskgroup_name

Specify the name of the disk group you want to modify. To determine the names of existing disk groups, query the `V$ASM_DISKGROUP` dynamic performance view.

add_disk_clause

Use this clause to add one or more disks to the disk group and specify attributes for the newly added disk. Oracle ASM automatically rebalances the disk group as part of this operation.

You cannot use this clause to change the failure group of a disk. Instead you must drop the disk from the disk group and then add the disk back into the disk group as part of the new failure group.

To determine the names of the disks already in this disk group, query the `V$ASM_DISK` dynamic performance view.

QUORUM | REGULAR

The semantics of these keyword are the same as the semantics in a `CREATE DISKGROUP` statement. See QUORUM | REGULAR for more information on these keywords.

You cannot change this qualifier for an existing disk or disk group. Therefore, you cannot specify in this clause a keyword different from the keyword that was specified when the disk group was created.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about the use of these keywords

FAILGROUP Clause

Use this clause to assign the newly added disk to a failure group. If you omit this clause and you are adding the disk to a normal or high redundancy disk group, then Oracle Database automatically adds the newly added disk to its own failure group. The implicit name of the failure group is the same as the operating system independent disk name (see "NAME Clause").

You cannot specify this clause if you are creating an external redundancy disk group.

qualified_disk_clause

This clause has the same semantics in CREATE DISKGROUP and ALTER DISKGROUP statements. For complete information on this clause, refer to *qualified_disk_clause* in the documentation on CREATE DISKGROUP.

drop_disk_clause

Use this clause to drop one or more disks from the disk group.

DROP DISK

The DROP DISK clause lets you drop one or more disks from the disk group and automatically rebalance the disk group. When you drop a disk, Oracle ASM relocates all the data from the disk and clears the disk header so that it no longer is part of the disk group. The disk header is not cleared if you specify the FORCE keyword.

Specify *disk_name* as shown in the NAME column of the V$ASM_DISK dynamic performance view.

If a disk to be dropped is a quorum disk or belongs to a quorum failure group, then you must specify QUORUM in order to drop the disk. See QUORUM | REGULAR.

DROP DISKS IN FAILGROUP

The DROP DISKS IN FAILGROUP clause lets you drop all the disks in the specified failure group. The behavior is otherwise the same as that for the DROP DISK clause.

If the specified failure group is a quorum failure group, then you must specify the QUORUM keyword in order to drop the disks. See QUORUM | REGULAR.

FORCE | NOFORCE

These keywords let you specify when the disk is considered to be no longer part of the disk group. The default and recommended setting is NOFORCE.

- When you specify NOFORCE, Oracle ASM reallocates all of the extents of the disk to other disks and then expels the disk from the disk group and rebalances the disk group.

> **Note:**
>
> DROP DISK ... NOFORCE returns control to the user before the disk can be safely reused or removed from the system. To ensure that the drop disk operation has completed, query the V$ASM_DISK view to verify that HEADER_STATUS has the value FORMER. Do not attempt to remove or reuse a disk if STATE has the value DROPPING. Query the V$ASM_OPERATION view for approximate information on how long it will take to complete the rebalance resulting from dropping the disk.If you also specify REBALANCE ... WAIT (see *rebalance_diskgroup_clause*), then the statement will not return until the rebalance operation is complete and the disk has been cleared. However, you should always verify that the HEADER_STATUS column of V$ASM_DISK is FORMER, because of the unlikely event the rebalance operations fails.

- When you specify `FORCE`, Oracle Database expels the disk from the disk group immediately. It then reconstructs the data from the redundant copies on other disks, reallocates the data to other disks, and rebalances the disk group.

  The `FORCE` clause can be useful, for example, if Oracle ASM can no longer read the disk to be dropped. However, it is more time consuming than a `NOFORCE` drop, and it can leave portions of a file with reduced protection. You cannot specify `FORCE` for an external redundancy disk group at all, because in the absence of redundant data on the disk, Oracle ASM must read the data from the disk before it can be dropped.

The rebalance operation invoked when a disk is dropped is time consuming, whether or not you specify `FORCE` or `NOFORCE`. You can monitor the progress by querying the `V$ASM_OPERATION` dynamic performance view. Refer to *rebalance_diskgroup_clause* for more information on rebalance operations.

resize_disk_clause

Use this clause to specify a new size for every disk in a disk group. This clause lets you override the size returned by the operating system or the size you specified previously for the disks.

SIZE

Specify the new size in kilobytes, megabytes, gigabytes, or terabytes. You cannot specify a size greater than the capacity of the disk. If you specify a size smaller than the disk capacity, then you limit the amount of disk space Oracle ASM will use. If you omit this clause, then Oracle ASM attempts programatically to determine the size of the disks.

replace_disk_clause

Use this clause to replace one or more disks in the disk group. This clause allows you to replace disks with a single operation, which is more efficient than dropping and adding each disk.

For *`disk_name`*, specify the name of the disk you want to replace. This name is assigned to the replacement disk. You can view disk names by querying the `NAME` column of the `V$ASM_DISK` dynamic performance view.

For *`path_name`*, specify the full path name for the replacement disk.

FORCE

Specify `FORCE` if you want Oracle ASM to add the replacement disk to the disk group even if the replacement disk is already a member of a disk group.

> **Note:**
>
> Using `FORCE` in this way may destroy existing disk groups.

NOFORCE

Specify `NOFORCE` if you want Oracle ASM to return an error if the replacement disk is already a member of a disk group. `NOFORCE` is the default.

POWER

The POWER clause has the same semantics here as for a manual rebalancing of a disk group, except that the power value cannot be set to 0. See POWER.

WAIT | NOWAIT

The WAIT and NOWAIT keywords have the same semantics here as for a manual rebalancing of a disk group. See WAIT | NOWAIT.

rename_disk_clause

Use this clause to rename one or more disks in the disk group. The disk group must be in the MOUNT RESTRICTED state and all disks in the disk group must be online.

RENAME DISK

Specify this clause to rename one or more disks. For each disk, specify the *old_disk_name* and *new_disk_name*. If *new_disk_name* already exists, then this operation fails.

RENAME DISKS ALL

Specify this clause to rename all disks in the disk group to a name of the form *diskgroupname_####*, where *####* is the disk number. Disk names that are already in the *diskgroupname_####* format are not changed.

disk_online_clause

Use this clause to bring one or more disks online and rebalance the disk group.

ONLINE DISK

The ONLINE DISK clause lets you bring one or more specified disks online and rebalance the disk group.

Specify *disk_name* as shown in the NAME column of the V$ASM_DISK dynamic performance view.

The QUORUM and REGULAR keywords have the same semantics here as they have when adding a disk to a disk group. See QUORUM | REGULAR.

ONLINE DISKS IN FAILGROUP

The ONLINE DISKS IN FAILGROUP clause lets you bring all disks in the specified failure group online and rebalance the disk group.

If the specified failure group is a quorum failure group, then you must specify the QUORUM keyword in order to bring the disks online. See QUORUM | REGULAR.

ALL

The ALL clause lets you bring all disks in the disk group online and rebalance the disk group.

POWER

The POWER clause has the same semantics here as for a manual rebalancing of a disk group. See POWER

WAIT | NOWAIT

The `WAIT` and `NOWAIT` keywords have the same semantics here as for a manual rebalancing of a disk group. See WAIT | NOWAIT.

disk_offline_clause

Use the *`disk_offline_clause`* to take one or more disks offline. This clause fails if the redundancy level of the disk group would be violated by taking the specified disks offline.

OFFLINE DISK

The `OFFLINE DISK` clause lets you take one or more specified disks offline.

Specify *`disk_name`* as shown in the `NAME` column of the `V$ASM_DISK` dynamic performance view.

The `QUORUM` and `REGULAR` keywords have the same semantics here as they have when adding a disk to a disk group. See QUORUM | REGULAR.

OFFLINE DISKS IN FAILGROUP

The `OFFLINE DISKS IN FAILGROUP` clause lets you take all disks in the specified failure group offline.

If the specified failure group is a quorum failure group, then you must specify the `QUORUM` keyword in order to take the disks offline. See QUORUM | REGULAR.

timeout_clause

By default, Oracle ASM drops a disk shortly after it is taken offline. You can delay this operation by specifying the *`timeout_clause`*, which gives you the opportunity to repair the disk and bring it back online. You can specify the timeout value in units of minute or hour. If you omit the unit, then the default is hour.

You can change the timeout period by specifying this clause multiple times. Each time you specify it, Oracle ASM measures the time from the most recent previous *`disk_offline_clause`* while the disk group is mounted. To learn how much time remains before Oracle ASM will drop an offline disk, query the `REPAIR_TIMER` column of `V$ASM_DISK`.

This clause overrides any previous setting of the `disk_repair_time` attribute. Refer to Table 13-2 for more information about disk group attributes.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about taking Oracle ASM disks online and offline

rebalance_diskgroup_clause

Use this clause to manually rebalance the disk group. During a rebalance operation, Oracle ASM redistributes data files evenly across all drives. This clause is rarely necessary, because Oracle ASM allocates files evenly and automatically rebalances disk groups when the storage configuration changes. However, it is useful if you want to perform a controlled rebalance operation. It allows you to include or exclude certain phases of a rebalance operation, pause and restart a rebalance operation, and adjust the power of a rebalance operation.

WITH | WITHOUT

A rebalance operation consists of the following phases: RESTORE (includes the RESYNC, RESILVER, or REBUILD phases), BALANCE, PREPARE, and COMPACT.

You can use the WITH or WITHOUT clause to instruct Oracle ASM to include or exclude specific phases of a rebalance operation. For example, if you have time constraints, you can include only the RESTORE phase. Or, if you are using flash storage disk groups or disk groups with flash cache, you can exclude the COMPACT phase, which is not beneficial for such disk groups.

- Use the WITH clause to include only the specified phases of a rebalance operation. You can specify any of phases RESTORE, BALANCE, PREPARE, and COMPACT. It is acceptable, but not necessary, to specify RESTORE, because the RESTORE phase always occurs.
- Use the WITHOUT clause to exclude the specified phases of a rebalance operation. You can specify any of the phases BALANCE, PREPARE, and COMPACT. You cannot specify RESTORE, because the RESTORE phase must always occur.

The order in which you specify multiple phases in the WITH or WITHOUT clause does not matter. Oracle ASM will perform the phases of the rebalance operation in the proper order. You cannot specify the RESYNC, RESILVER, or REBUILD phases; they are part of the RESTORE phase.

If you omit the WITH and WITHOUT clauses, then Oracle ASM performs all phases of the rebalance operation.

You can monitor the progress of the rebalance operation by querying the V$ASM_OPERATION dynamic performance view.

See *Oracle Automatic Storage Management Administrator's Guide* for more information on the phases of a rebalance operation.

POWER

This clause lets you specify the power, or speed, of the rebalance operation. It also lets you stop the rebalance operation.

For *integer*, specify a value from 0 to 1024:

- A value of 1 through 1024 specifies the power at which Oracle ASM is to perform the rebalance operation, with 1 representing the lowest possible power and 1024 representing the highest possible power.
- A value of 0 stops an active rebalance operation. No further rebalancing will occur until the start of another manual or automatic rebalance operation on the disk group, and at that time the rebalance operation will start from the beginning. If you would like to have the option of later resuming the rebalance operation from where it left off, then instead stop the rebalance operation by specifying MODIFY POWER 0. See the clause MODIFY POWER for more information.

If you omit the POWER clause, then the default power is determined as follows:

- For flex disk groups, Oracle ASM rebalances each file group according the value of its POWER_LIMIT property. If the POWER_LIMIT property is not set for a file group, then Oracle ASM uses the value of the ASM_POWER_LIMIT initialization parameter for the file group.

- For all other types of disk groups, if you omit the POWER clause, then Oracle ASM rebalances the disk group according to the value of the ASM_POWER_LIMIT initialization parameter.

WAIT | NOWAIT

Use this clause to specify when, in the course of the rebalance operation, control should be returned to the user.

- Specify WAIT if you want control returned to the user after the rebalance operation has finished. You can explicitly terminate a rebalance operation running in WAIT mode, although doing so does not undo any completed disk add, drop, or resize operations in the same statement.
- Specify NOWAIT if you want control returned to the user immediately after the statement is issued. This is the default.

MODIFY POWER

Use this clause to pause, resume, or change the power of an active rebalance operation.

You can specify *integer* as follows:

- Specify 0 to pause the rebalance operation. When you pause a rebalance operation in this manner, you can subsequently resume the operation from the phase where it left off by issuing an ALTER DISKGROUP ... MODIFY POWER ... statement. If you subsequently start a manual rebalance operation on the disk group using the clause POWER, or an automatic rebalance operation for the disk group occurs, then the rebalance operation will start at the beginning.
- Specify 1 through 1024 to specify the power of the rebalance operation, with 1 representing the lowest possible power and 1024 representing the highest possible power. If a rebalance operation is running, then Oracle ASM changes the power without interrupting the operation. If a rebalance operation was previously paused with the MODIFY POWER 0 clause, then the rebalance operation resumes at the specified power.
- Omit *integer* to specify the default power. If a rebalance operation is running, then Oracle ASM changes the power to the default power without interrupting the operation. If a rebalance operation was previously paused with the MODIFY POWER 0 clause, then the rebalance operation resumes at the default power. Refer to the clause POWER for information on how the default power is determined.

See Also:

- *Oracle Database Reference* for more information on the ASM_POWER_LIMIT initialization parameter and the V$ASM_OPERATION dynamic performance view
- Rebalancing a Disk Group: Example

check_diskgroup_clause

The *check_diskgroup_clause* lets you verify the internal consistency of Oracle ASM disk group metadata. The disk group must be mounted. Oracle ASM displays summary errors and writes the details of the detected errors in the alert log.

The CHECK keyword performs the following operations:

- Checks the consistency of the disk.
- Cross checks all the file extent maps and allocation tables for consistency.
- Checks that the alias metadata directory and file directory are linked correctly.
- Checks that the alias directory tree is linked correctly.
- Checks that Oracle ASM metadata directories do not have unreachable allocated blocks.

REPAIR | NOREPAIR

This clause lets you instruct Oracle ASM whether or not to attempt to repair any errors found during the consistency check. The default is `NOREPAIR`. The `NOREPAIR` setting is useful if you want to be alerted to any inconsistencies but do not want Oracle ASM to take any automatic action to resolve them.

Deprecated Clauses

In earlier releases, you could specify `CHECK` for `ALL`, `DISK`, `DISKS IN FAILGROUP`, or `FILE`. Those clauses have been deprecated as they are no longer needed. If you specify them, then their behavior is the same as in earlier releases and a message is added to the alert log. However, Oracle recommends that you do not introduce these clauses into your new code, as they are scheduled for desupport. The deprecated clauses are these:

- `ALL` checks all disks and files in the disk group.
- `DISK` checks one or more specified disks in the disk group.
- `DISKS IN FAILGROUP` checks all disks in a specified failure group.
- `FILE` checks one or more specified files in the disk group. You must use one of the reference forms of the filename. Refer to *ASM_filename* for information on the reference forms of Oracle ASM filenames.

diskgroup_template_clauses

A template is a named collection of attributes. When you create a disk group, Oracle ASM associates a set of initial system default templates with that disk group. The attributes defined by the template are applied to all files in the disk group. Table 10-2 lists the system default templates and the attributes they apply to the various file types. The *`diskgroup_template_clauses`* described following the table let you change the template attributes and create new templates.

You cannot use this clause to change the attributes of a disk group file after it has been created. Instead, you must use Recovery Manager (RMAN) to copy the file into a new file with the new attributes.

Table 10-2 Oracle Automatic Storage Management System Default File Group Templates

Template Name	File Type	Mirroring Level in External Redundancy Disk Groups	Mirroring Level in Normal Redundancy Disk Groups	Mirroring Level in High Redundancy Disk Groups	Striped	Region
`CONTROLFILE`	Control files	Unprotected	3-way mirror	3-way mirror	`FINE`	`COLD`

Table 10-2 (Cont.) Oracle Automatic Storage Management System Default File Group Templates

Template Name	File Type	Mirroring Level in External Redundancy Disk Groups	Mirroring Level in Normal Redundancy Disk Groups	Mirroring Level in High Redundancy Disk Groups	Striped	Region
DATAFILE	Data Files and copies	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
ONLINELOG	Online logs	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
ARCHIVELOG	Archive logs	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
TEMPFILE	Temp files	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
BACKUPSET	Data File backup pieces, data file incremental backup pieces, and archive log backup pieces	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
PARAMETERFILE	SPFILE	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
DATAGUARDCONFIG	Disaster recovery configurations (used in standby databases)	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
FLASHBACK	Flashback logs	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
CHANGETRACKING	Block change tracking data (used during incremental backups)	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
DUMPSET	Data Pump dumpset	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
XTRANSPORT	Cross-platform converted data file	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
AUTOBACKUP	Automatic backup files	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
ASMPARAMETERFILE	SPFILE	Unprotected	2-way mirror	3-way mirror	COARSE	COLD
OCRFILE	Oracle Cluster Registry file	Unprotected	2-way mirror	3-way mirror	COARSE	COLD

ADD TEMPLATE

Use this clause to add one or more named templates to a disk group. To determine the names of existing templates, query the `V$ASM_TEMPLATE` dynamic performance view.

MODIFY TEMPLATE

Use this clause to modify the attributes of a system default or user-defined disk group template. Only the specified attributes are altered. Unspecified properties retain their current values.

> **Note:**
>
> In earlier releases, the keywords ALTER TEMPLATE were used instead of MODIFY TEMPLATE. The ALTER keyword is still supported for backward compatibility, but is replaced with MODIFY for consistency with other Oracle SQL.

template_name

Specify the name of the template to be added or modified. The maximum length of a template name is 30 characters. The name must satisfy the requirements listed in "Database Object Naming Rules ".

redundancy_clause

Specify the redundancy level of the newly added or modified template:

- MIRROR: Files to which this template are applied are protected by mirroring their data blocks. In normal redundancy disk groups, each primary extent has one mirror extent (2-way mirroring). For high redundancy disk groups, each primary extent has two mirror extents (3-way mirroring). You cannot specify MIRROR for templates in external redundancy disk groups.
- HIGH: Files to which this template are applied are protected by mirroring their data blocks. Each primary extent has two mirror extents (3-way mirroring) for both normal redundancy and high redundancy disk groups. You cannot specify HIGH for templates in external redundancy disk groups.
- UNPROTECTED: Files to which this template are applied are not protected by Automated Storage Management from media failures. Disks taken offline, either through system action or by user command, can cause loss of unprotected files. UNPROTECTED is the only valid setting for external redundancy disk groups. UNPROTECTED may not be specified for templates in high redundancy disk groups. Oracle discourages the use of unprotected files in high and normal redundancy disk groups.

If you omit this clause, then the value defaults to MIRROR for a normal redundancy disk group, HIGH for a high redundancy disk group, and UNPROTECTED for an external redundancy disk group.

striping_clause

Specify how the files to which this template are applied will be striped:

- FINE: Files to which this template are applied are striped every 128KB. This striping mode is not valid for an Oracle ASM spfile.
- COARSE: Files to which this template are applied are striped every 1MB. This is the default value.

disk_region_clause

This clause lets you determine the Intelligent Data Placement attribute of the disk group file. Specify the region of the disk in which you want Oracle ASM to allocate extents for the file:

- `HOT`: Extents are allocated in the region of the disk furthest away from the spindle. These outer tracks on the disk are longer than inner tracks, and so have more sectors and increased throughput.
- `COLD`: Extents are allocated in the region of the disk closest to the spindle.
- `MIRRORHOT` and `MIRRORCOLD`: Specify the region desired for the mirrored datablocks of the file.

If no space is available in the desired disk region, then Oracle ASM allocates extents in the other region but initiates a rebalance to adjust the size of the region.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information on Intelligent Data Placement

DROP TEMPLATE

Use this clause to drop one or more templates from the disk group. You can use this clause to drop only user-defined templates, not system default templates.

diskgroup_directory_clauses

Before you can create alias names for Oracle ASM filenames (see *diskgroup_alias_clauses*), you must specify the full directory structure in which the alias name will reside. The *`diskgroup_directory_clauses`* let you create and manipulate such a directory structure.

ADD DIRECTORY

Use this clause to create a new directory path for hierarchically named aliases. Use a slash (/) to separate components of the directory. Each directory component can be up to 48 bytes in length and must not contain the slash character. You cannot use a space for the first or last character of any component. The total length of the directory path cannot exceed 256 bytes minus the length of any alias name you intend to create in this directory (see *diskgroup_alias_clauses*).

DROP DIRECTORY

Use this clause to drop a directory for hierarchically named aliases. Oracle ASM will not drop the directory if it contains any alias definitions unless you also specify `FORCE`. This clause is not valid for dropping directories created as part of a system alias. Such directories are labeled with the value `Y` in the `SYSTEM_CREATED` column of the `V$ASM_ALIAS` dynamic performance view.

RENAME DIRECTORY

Use this clause to change the name of a directory for hierarchically named aliases. This clause is not valid for renaming directories created as part of a system alias. Such directories are labeled with the value `Y` in the `SYSTEM_CREATED` column of the `V$ASM_ALIAS` dynamic performance view.

diskgroup_alias_clauses

When an Oracle ASM file is created, either implicitly or by user specification, Oracle ASM assigns to the file a fully qualified name ending in a dotted pair of numbers (see *file_specification*). The *diskgroup_alias_clauses* let you create more user-friendly alias names for the Oracle ASM filenames. You cannot specify an alias name that ends in a dotted pair of numbers, as this format is indistinguishable from an Oracle ASM filename.

Before specifying this clause, you must first create the directory structure appropriate for your naming conventions (see *diskgroup_directory_clauses*). The total length of the alias name, including the directory prefix, is limited to 256 bytes. Alias names are case insensitive but case retentive.

ADD ALIAS

Use this clause to create an alias name for an Oracle ASM filename. The *alias_name* consists of the full directory path and the alias itself. To determine the names of existing Oracle ASM aliases, query the V$ASM_ALIAS dynamic performance view. Refer to *ASM_filename* for information on Oracle ASM filenames.

DROP ALIAS

Use this clause to remove an alias name from the disk group directory. Each alias name consists of the full directory path and the alias itself. The underlying file to which the alias refers remains unchanged.

RENAME ALIAS

Use this clause to change the name of an existing alias. The *alias_name* consists of the full directory path and the alias itself.

Restriction on Dropping and Renaming Aliases

You cannot drop or rename a system-generated alias. To determine whether an alias was system generated, query the SYSTEM_CREATED column of the V$ASM_ALIAS dynamic performance view.

diskgroup_volume_clauses

Use these clauses to manipulate logical Oracle ASM Dynamic Volume Manager (Oracle ADVM) volumes corresponding to physical volume devices. To use these clauses, Oracle ASM must be started and the disk group being modified must be mounted.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about disk group volumes, including examples

add_volume_clause

Use this clause to add a volume to the disk group.

For *asm_volume*, specify the name of the volume. The name can contain only alphanumeric characters and the first character must be alphabetic. The maximum length of the name is platform dependent. Refer to *Oracle Automatic Storage Management Administrator's Guide* for more information.

For *size_clause*, specify the size of the Oracle ADVM volume. The Oracle ASM instance determines whether sufficient space exists to create the volume. If sufficient space does not exist, then the Oracle ASM instance returns an error. If sufficient space does exist, then all nodes in the cluster with an Oracle ASM instance running and the disk group mounted are notified of the addition. Oracle ASM creates and enables on those nodes a volume device that can be used to create and mount file systems.

The following optional settings are also available:

- In the *redundancy_clause*, specify the redundancy level of the Oracle ADVM volume. You can specify this clause only when creating a volume in a normal redundancy disk group. You can specify the following volume redundancy levels:
  - MIRROR: 2-way mirroring of the volume. This is the default.
  - HIGH: 3-way mirroring of the volume.
  - UNPROTECTED: No mirroring of the volume.

  You cannot specify the *redundancy_clause* when creating a volume in a high redundancy disk group or an external redundancy disk group. If you do so, then an error will result. In high redundancy disk groups, Oracle Database automatically sets the volume redundancy to HIGH (3-way mirroring). In external redundancy disk groups, Oracle Database automatically sets the volume redundancy to UNPROTECTED (no mirroring).
- In the STRIPE_WIDTH clause, specify a stripe width for the Oracle ADVM volume. The valid range is from 4KB to 1MB, at intervals of the power of 2. The default value is 128K.
- In the STRIPE_COLUMNS clause, specify the number of stripes in a stripe set of the Oracle ADVM volume. The valid range is 1 to 8. The default is 4. If STRIPE_COLUMNS is set to 1, then striping becomes disabled. In this case, the stripe width is the extent size of the volume. This volume extent size is 64 times the allocation unit (AU) size of the disk group.
- In the *disk_region_clause* clause, specify the Intelligent Data Placement attribute of both the primary and nonprimary mirror of the disk group volume. The default for both is COLD. See disk_region_clause for details on this clause.

modify_volume_clause

Use this clause to modify the characteristics of an existing Oracle ADVM volume. You must specify at least one of the following clauses:

- In the *disk_region_clause* clause, specify the Intelligent Data Placement attribute of both the primary and nonprimary mirror of the disk group volume. The default for the primary mirror is COLD. The default for mirror and high redundancy is HOT. See disk_region_clause for details on this clause.
- In the MOUNTPATH clause, specify the mountpath name associated with the volume. The *mountpath_name* can be up to 1024 characters.
- In the USAGE clause, specify the usage name associated with the volume. The *usage_name* can be up to 30 characters.

RESIZE VOLUME Clause

Use this clause to change the size of an existing Oracle ADVM volume. In an Oracle ASM cluster, the new size is propagated to all nodes. If an Oracle Automatic Storage Management File System (ACFS) exists on the volume, then you must use the acfsutil size command instead of the ALTER DISKGROUP statement.

See Also:

Oracle Automatic Storage Management Administrator's Guide for more information about the `acfsutil size` command

DROP VOLUME Clause

Use this clause to remove the Oracle ASM file that is the storage container for an existing Oracle ADVM volume. In an Oracle ASM cluster, all nodes with an Oracle ASM instance running and with this disk group open are notified of the drop operation, which results in removal of the volume device. If the volume file is open, then this clause returns an error.

diskgroup_attributes

Use this clause to specify attributes for the disk group. Table 13-2 lists the attributes you can set with this clause. Refer to the `CREATE DISKGROUP` "ATTRIBUTE Clause " for information on the behavior of this clause.

modify_diskgroup_file

Use this clause to modify the Intelligent Data Placement attributes of an existing disk group file. When you modify the Intelligent Data Placement for a file, this action will apply to new extensions of the file, but existing file contents are not affected until a rebalance operation. To apply the new Intelligent Data Placement policy for existing file contents, you can manually initiate a rebalance. A rebalance operation uses the last specified policy for the file extents.

See Also:

- disk_region_clause for more information on this clause
- *Oracle Automatic Storage Management Administrator's Guide* for more information on manually rebalancing disk groups

drop_diskgroup_file_clause

Use this clause to drop a file from the disk group. Oracle ASM also drops all aliases associated with the file being dropped. You must use one of the reference forms of the filename. Most Oracle ASM files do not need to be manually deleted because, as Oracle Managed Files, they are removed automatically when they are no longer needed. Refer to *ASM_filename* for information on the reference forms of Oracle ASM filenames.

You cannot drop a disk group file it if is the spfile that was used to start up the current instance or any instance in the Oracle ASM cluster.

convert_redundancy_clause

You can use this clause to convert a `NORMAL REDUNDANCY` or `HIGH REDUNDANCY` disk group to a `FLEX REDUNDANCY` disk group. The disk group must have at least three failure groups before you start the conversion, and must be in the MOUNT RESTRICTED state for the duration of the conversion.

usergroup_clauses

Use these clauses to add a user group to the disk group, remove a user group from the disk group, or to add a member to or drop a member from an existing user group.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for detailed information about user groups and members, including examples

ADD USERGROUP

Use this clause to add a user group to the disk group. You must have SYSASM or SYSDBA privilege to create a user group. The maximum length of a user group name is 63 bytes. If you specify the user name, then it must be in the OS password file and its length cannot exceed 32 characters.

MODIFY USERGROUP

Use these clauses to add a member to or drop a member from an existing user group. You must be an Oracle ASM administrator (with SYSASM privilege) or the creator (with SYSDBA privilege) of the user group to use these clauses. The user name must be an existing user in the OS password file.

DROP USERGROUP

Use this clause to drop an existing user group from the disk group. You must be an Oracle ASM administrator (with SYSASM privilege) or the creator (with SYSDBA privilege) of the user group to use this clause. Dropping a user group may leave a disk group file without a valid user group. In this case, you can update the disk group file manually to add a new, valid group using the *file_permissions_clause*.

user_clauses

Use these clauses to add a user to, drop a user from, or replace a user in a disk group.

> **Note:**
>
> When administering users with SQL*Plus, the users must be existing operating system users and their user names must have corresponding operating system user IDs. However, only users in the same cluster as the Oracle ASM instance can be validated.

ADD USER

Use this clause to add one or more operating system (OS) users to an Oracle ASM disk group and give those users access privileges on the disk group. A user name must be an existing user in the OS password file and its length cannot exceed 32 characters. If a specified user already exists in the disk group, as shown by V$ASM_USER, then the command records an error and continues to add other users, if any have been specified. This command is seldom needed, because the OS user running the database instance is added to a disk group automatically when the instance accesses the disk group. However, this clause is useful when adding users that are not associated with a particular database instance.

DROP USER

Use this clause to drop one or more users from the disk group. If a specified user is not in the disk group, then this clause records an error and continues to drop other users, if any are specified. If the user owns any files, then you must specify the `CASCADE` keyword, which drops the user and all the user's files. If any files owned by the user are open, then `DROP USER CASCADE` fails with an error.

To delete a user without deleting the files owned by that user, change the owner of each of these files to another user and then issue an `ALTER DISKGROUP` ... `DROP USER` statement on the user. Alternatively, you can issue an `ALTER DISKGROUP` ... `REPLACE USER` statement to replace the user you want to drop with a user that currently does not exist in the disk group. This operation has the side effect of making the new user the owner of files that were previously owned by the dropped user.

REPLACE USER

Use this clause to replace *`old_user`* with *`new_user`* in the disk group. All files that are currently owned by *`old_user`* will become owned by *`new_user`*, and *`old_user`* will be dropped from the disk group. *`old_user`* must exist in the disk group and *`new_user`* must not exist in the disk group.

file_permissions_clause

Use this clause to change the permission settings of a disk group file. The three classes of permissions are owner, user group, and other. You must be the file owner or the Oracle ASM administrator to use this clause.

If you change the permission settings of an open file, then the operation currently running on the file will complete using the old permission settings. The new permission settings will take effect when re-authentication is required.

file_owner_clause

Use this clause to set the owner or user group for a specified file. You must be the Oracle ASM administrator to change the owner of the file. You must be the owner of the file or the Oracle ASM administrator to change the user group of a file. In addition, to change the associated user group of a file, the specified user group must already exist in the disk group, and the owner of the file must be a member of that user group.

If you use this clause on an open file, then the following conditions apply:

- If you change the owner or user group of an open file, then the operation currently running on the file will complete using the old owner or user group. The new owner or user group will take effect when re-authentication is required.
- If you change the owner of an open file, then the new owner of the file cannot be dropped from the disk group until the instance has been restarted. In an Oracle ASM cluster, the new owner of the file cannot be dropped until all instances in the cluster have been restarted.
- If you change the owner of an open file, then the old owner cannot be dropped while the file is still open, even after the ownership of the file has changed.

scrub_clause

Use this clause to scrub a disk group. The scrub operation checks for logical data corruptions and repairs the corruptions automatically in normal and high redundancy disks groups.

- Use the `FILE` clause to scrub the specified Oracle ASM file in the disk group. You must use one of the reference forms of the *`ASM_filename`*. Refer to *ASM_filename* for information on the reference forms of Oracle ASM filenames.
- Use the `DISK` clause to scrub the specified disk in the disk group.
- If you do not specify `FILE` or `DISK`, then all files and disks in the disk group are scrubbed.

REPAIR | NOREPAIR

Specify `REPAIR` to attempt to repair any errors found during the logical data corruption check. Specify `NOREPAIR` to be alerted of any corruptions; Oracle ASM will not take any action to resolve them. The default is `NOREPAIR`.

POWER

Use the `POWER` clause to specify the power level of the scrub operation. Valid values are `AUTO`, `LOW`, `HIGH`, and `MAX`. If you omit this clause, then the power level defaults to `AUTO` and the power adjusts to the optimum level for the system.

WAIT | NOWAIT

Specify `WAIT` to allow the scrub operation to complete before returning control to the user. Specify `NOWAIT` to add the operation to the scrubbing queue and return control to the user immediately. The default is `NOWAIT`.

FORCE | NOFORCE

Specify `FORCE` to process the command even if the system I/O load is high or scrubbing has been disabled at the system level. Specify `NOFORCE` to process the command normally. The default is `NOFORCE`.

You can monitor the progress of the scrub operation by querying the `V$ASM_ OPERATION` dynamic performance view.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information on scrubbing disk groups and "Scrubbing a Disk Group: Example"

quotagroup_clauses

Use these clauses to add a quota group to the disk group, modify a quota group, move a file group into a quota group, or drop a quota group.

A quota group is a collection of file groups. A file group is a container for all files of a database within one disk group. A quota group has a specified quota limit, which is the maximum amount of storage space that its file groups can collectively use. Therefore, a quota group enables you to define the quota limit for a group of databases within a disk group. The sum of the quota limits for all quota groups in a disk group can exceed the storage capacity of the disk group.

Each disk group contains a default quota group named GENERIC. If you create a file group and do not specify its quota group, then the file group belongs to the GENERIC quota group. Oracle ASM automatically creates the GENERIC quota group when you create a disk group with the compatible.asm attribute set to 12.2 or higher, or when you set compatible.asm to 12.2 or higher for an existing disk group. Initially, the quota limit for GENERIC is UNLIMITED. You can subsequently modify this quota limit with the MODIFY QUOTAGROUP clause.

ADD QUOTAGROUP

Use this clause to create a quota group and add it to the disk group. For *quotagroup_name*, specify the name of the new quota group.

The SET clause allows you to set the quota limit for the quota group.

- For *property_name*, specify QUOTA.
- For *property_value*, specify one of the following clauses:
  - Specify *size_clause* to set a number of bytes for the quota limit. The minimum value you can specify is 1 byte. You can specify a value that is greater than the storage size of the disk group. In this case, storage use is limited by the current size of the disk group. However, if you subsequently increase the storage space for the disk group to a size that exceeds the quota limit, then the quota limit will be enforced. Refer to *size_clause* for the syntax and semantics of this clause. Note that specifying 0 bytes is equivalent to specifying UNLIMITED.
  - Specify UNLIMITED if you do not want to set a quota limit. In this case, storage use is limited by the storage size of the disk group.

If you omit the SET clause, then the default is SET QUOTA=UNLIMITED.

MODIFY QUOTAGROUP

Use this clause to modify the quota limit for a quota group. For *quotagroup_name*, specify the name of the quota group you want to modify. You can modify the quota limit for any quota group, including the GENERIC quota group. The SET clause has the same semantics here as for the ADD QUOTAGROUP clause. The quota limit can be set below the amount of space currently used by the quota group. This action prevents any additional space from being allocated for files described by file groups associated with this quota group.

MOVE FILEGROUP

Use this clause to move a file group from one quota group to another. For *filegroup_name*, specify the file group you want to move. For *quotagroup_name*, specify the name of the destination quota group. If the move operation causes the amount of used storage space in the destination quota group to exceed the quota limit, then the operation succeeds, but no new storage allocations can take place in the file groups within the quota group. This capability enables you to stop any files described by a specific file group from allocating additional space.

DROP QUOTAGROUP

Use this clause to drop a quota group from the disk group. For *quotagroup_name*, specify the quota group you want to drop. The quota group must not contain any file groups. You cannot drop the quota group GENERIC.

> **See Also:**
>
> *Automatic Storage Management Administrator's Guide* for more information on quota groups

filegroup_clauses

The *filegroup_clauses* are valid only for flex disk groups. Use these clauses to create a file group, modify a file group, move a file into a file group, or drop a file group. A file group is a container for all files of a database within one disk group. A file group must belong to a quota group.

Each disk group has a default file group with FILEGROUP_NUMBER = 0.

add_filegroup_clause

Use this clause to create a file group.

For *filegroup_name*, specify the name of the new file group. The maximum length of a file group name is 127 characters. The name must satisfy the requirements listed in "Database Object Naming Rules ", with the following addition: File group names are not case sensitive, even if you specify them with quotation marks. They are always stored internally as uppercase. File group names must be unique within a disk group.

- Use the DATABASE clause to specify the database (non-CDB, CDB, or PDB) with which the file group is associated.
- Use the CLUSTER clause to specify the cluster with which the file group is associated.
- Use the VOLUME clause to specify the volume with which the file group is associated.

You cannot associate more than one file group in the same disk group with the same database, cluster, or volume. If the database, cluster, or volume does not exist at the time of file group creation, then the file group will be automatically associated with it when it is subsequently created. Database, cluster, and volume names must satisfy the requirements listed in "Database Object Naming Rules ".

The SET clause allows you to set properties for the file group. If you do not specify the SET clause for a property, then the default value is assigned. You can specify the *file_type* for any property for which a file type applies. If you do not specify *file_type* for such a property, then the property applies to all file types. For complete information on file group properties and their default values, see *Oracle Automatic Storage Management Administrator's Guide*.

modify_filegroup_clause

Use this clause to modify file group properties. For *filegroup_name*, specify the name of the file group you want to modify. You can modify properties for any file group, including the default file group. Any that you do not specify with this clause remain unchanged. The SET clause has the same semantics here as for the *add_filegroup_clause*.

move_to_filegroup_clause

Use this clause to move a file to a file group. If the file is currently associated with a different file group, then it is disassociated from that file group. The target file group must have enough space available to contain the file. You must be the owner of the file and the target file group.

drop_filegroup_clause

Use this clause to drop a file group from a quota group. For *filegroup_name*, specify the name of the file group you want to drop. The file group must be empty, or you must specify the CASCADE keyword. If you specify CASCADE, then the files in the file group are dropped.

> **See Also:**
>
> *Automatic Storage Management Administrator's Guide* for more information on file groups

undrop_disk_clause

Use this clause to cancel the drop of disks from the disk group. You can cancel the pending drop of all the disks in one or more disk groups (by specifying *diskgroup_name*) or of all the disks in all disk groups (by specifying ALL).

This clause is not relevant for disks that have already been completely dropped from the disk group or for disk groups that have been completely dropped. This clause results in a long-running operation. You can see the status of the operation by querying the V$ASM_OPERATION dynamic performance view.

> **See Also:**
>
> V$ASM_OPERATION for more information on the details of long-running Oracle ASM operations

diskgroup_availability

Use this clause to make one or more disk groups available or unavailable to the database instances running on the same node as the Oracle ASM instance. This clause does not affect the status of the disk group on other nodes in a cluster.

MOUNT

Specify MOUNT to mount the disk groups in the local Oracle ASM instance. Specify ALL MOUNT to mount all disk groups specified in the ASM_DISKGROUPS initialization parameter. File operations can only be performed when a disk group is mounted. If Oracle ASM is running in a cluster or a standalone server managed by Oracle Grid Infrastructure for Independent Servers (Oracle Restart), then the MOUNT clause automatically brings the corresponding resource online.

> **Note:**
>
> Starting with Oracle Database 12*c*, Oracle Grid Infrastructure for Independent Servers (Oracle Restart) is deprecated and is subject to desupport in future releases. Oracle continues to provide Oracle Automatic Storage Management (Oracle ASM) as part of the Oracle Grid Infrastructure installation for Standalone and Cluster deployments.

RESTRICTED | NORMAL

Use these clauses to determine the manner in which the disk groups are mounted.

- In the `RESTRICTED` mode, the disk group is mounted in single-instance exclusive mode. No other Oracle ASM instance in the same cluster can mount that disk group. In this mode the disk group is not usable by any Oracle ASM client.
- In the `NORMAL` mode, the disk group is mounted in shared mode, so that other Oracle ASM instances and clients can access the disk group. This is the default.

FORCE | NOFORCE

Use these clauses to determine the circumstances under which the disk groups are mounted.

- In the `FORCE` mode, Oracle ASM attempts to mount the disk group even if it cannot discover all of the devices that belong to the disk group. This setting is useful if some of the disks in a normal or high redundancy disk group became unavailable while the disk group was dismounted. When `MOUNT FORCE` succeeds, Oracle ASM takes the missing disks offline.

  If Oracle ASM discovers *all* of the disks in the disk group, then `MOUNT FORCE` fails. Therefore, use the `MOUNT FORCE` setting only if some disks are unavailable. Otherwise, use `NOFORCE`.

  In normal- and high-redundancy disk groups, disks from one failure group can be unavailable and `MOUNT FORCE` will succeed. Also in high-redundancy disk groups, two disks in two different failure groups can be unavailable and `MOUNT FORCE` will succeed. Any other combination of unavailable disks causes the operation to fail, because Oracle ASM cannot guarantee that a valid copy of all user data or metadata exists on the available disks.
- In the `NOFORCE` mode, Oracle ASM does not attempt to mount the disk group unless it can discover all the member disks. This is the default.

> **See Also:**
>
> `ASM_DISKGROUPS` for more information about adding disk group names to the initialization parameter file

DISMOUNT

Specify `DISMOUNT` to dismount the specified disk groups. Oracle ASM returns an error if any file in the disk group is open unless you also specify `FORCE`. Specify `ALL DISMOUNT` to dismount all currently mounted disk groups. File operations can only be performed when a disk group is mounted. If Oracle ASM is running in a cluster or a standalone server managed by Oracle Grid Infrastructure for Independent Servers (Oracle Restart), then the `DISMOUNT` clause automatically takes the corresponding resource offline.

> **Note:**
>
> Starting with Oracle Database 12*c*, Oracle Grid Infrastructure for Independent Servers (Oracle Restart) is deprecated and is subject to desupport in future releases. Oracle continues to provide Oracle Automatic Storage Management (Oracle ASM) as part of the Oracle Grid Infrastructure installation for Standalone and Cluster deployments.

FORCE

Specify `FORCE` if you want Oracle ASM to dismount the disk groups even if some files in the disk group are open.

enable_disable_volume

Use this clause to enable or disable one or more volumes in the disk group.

- For each volume you enable, Oracle ASM creates a volume device file on the local node that can be used to create or mount the file system.
- For each volume you disable, Oracle ASM deletes the device file on the local node. If the volume file is open on the local node, then the `DISABLE` clause returns an error.

Use the `ALL` keyword to enable or disable all volumes in the disk group. If you specify `ALTER DISKGROUP ALL ...`, then you must use the `ALL` keyword in this clause as well.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about disk group volumes

Examples

The following examples require a disk group called `dgroup_01`. They assume that `ASM_DISKSTRING` is set to `/devices/disks/*`. In addition, they assume the Oracle user has read/write permission to `/devices/disks/d100`. Refer to "Creating a Diskgroup: Example" to create `dgroup_01`.

Adding a Disk to a Disk Group: Example

To add a disk, `d100`, to a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  ADD DISK '/devices/disks/d100';
```

Dropping a Disk from a Disk Group: Example

To drop a disk, `dgroup_01_0000`, from a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  DROP DISK dgroup_01_0000;
```

Undropping a Disk from a Disk Group: Example

To cancel the drop of disks from a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  UNDROP DISKS;
```

Resizing a Disk Group: Example

To resize every disk in a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  RESIZE ALL
  SIZE 36G;
```

Rebalancing a Disk Group: Example

To manually rebalance a disk group, `dgroup_01`, and permit Oracle ASM to execute the rebalance as fast as possible, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  REBALANCE POWER 11 WAIT;
```

The `WAIT` keyword causes the database to wait for the disk group to be rebalanced before returning control to the user.

Verifying the Internal Consistency of Disk Group Metadata: Example

To verify the internal consistency of Oracle ASM disk group metadata and instruct Oracle ASM to repair any errors found, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  CHECK ALL
  REPAIR;
```

Adding a Named Template to a Disk Group: Example

To add a named template, `template_01` to a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  ADD TEMPLATE template_01
    ATTRIBUTES (UNPROTECTED COARSE);
```

Changing the Attributes of a Disk Group Template: Example

To modify the attributes of a system default or user-defined disk group template, `template_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  MODIFY TEMPLATE template_01
    ATTRIBUTES (FINE);
```

Dropping a User-Defined Template from a Disk Group: Example

To drop a user-defined template, `template_01`, from a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  DROP TEMPLATE template_01;
```

Creating a Directory Path for Hierarchically Named Aliases: Example

To specify the directory structure in which alias names will reside, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  ADD DIRECTORY '+dgroup_01/alias_dir';
```

Creating an Alias Name for an Oracle ASM Filename: Example

To create a user alias by specifying the numeric Oracle ASM filename, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  ADD ALIAS '+dgroup_01/alias_dir/datafile.dbf'
    FOR '+dgroup_01.261.1';
```

Scrubbing a Disk Group: Example

To scrub a disk group, `dgroup_01`, issue the following statement. This statement attempts to repair any errors found during the logical data corruption check and allows the scrub operation to complete before returning control to the user.

```
ALTER DISKGROUP dgroup_01
  SCRUB REPAIR WAIT;
```

Dismounting a Disk Group: Example

To dismount a disk group, `dgroup_01`, issue the following statement. This statement dismounts the disk group even if one or more files are active:

```
ALTER DISKGROUP dgroup_01
  DISMOUNT FORCE;
```

Mounting a Disk Group: Example

To mount a disk group, `dgroup_01`, issue the following statement:

```
ALTER DISKGROUP dgroup_01
  MOUNT;
```

ALTER FLASHBACK ARCHIVE

Purpose

Use the `ALTER FLASHBACK ARCHIVE` statement for these operations:

- Designate a flashback data archive as the default flashback data archive for the system
- Add a tablespace for use by the flashback data archive
- Change the quota of a tablespace used by the flashback data archive
- Remove a tablespace from use by the flashback data archive
- Change the retention period of the flashback data archive
- Purge the flashback data archive of old data that is no longer needed

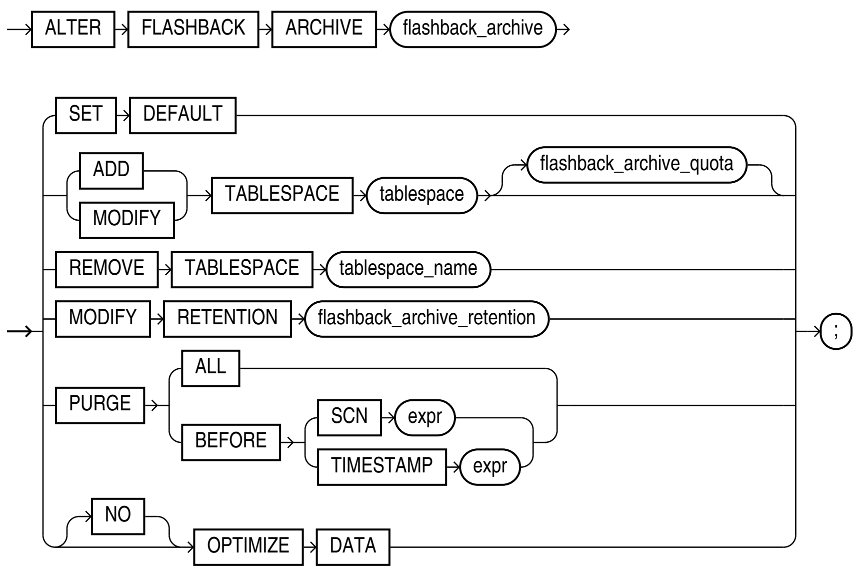
> **See Also:**
>
> *Oracle Database Development Guide* and CREATE FLASHBACK ARCHIVE for more information on using flashback data archives

Prerequisites

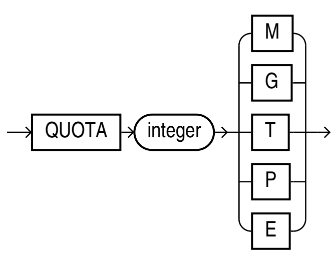
You must have the `FLASHBACK ARCHIVE ADMINISTER` system privilege to alter a flashback data archive in any way. You must also have appropriate privileges on the affected tablespaces to add, modify, or remove a flashback data archive tablespace.

Syntax

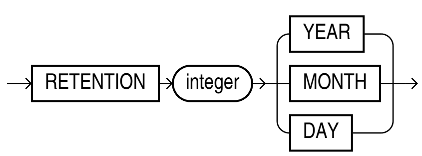
alter_flashback_archive::=



flashback_archive_quota::=



flashback_archive_retention::=



Semantics

flashback_archive

Specify the name of an existing flashback data archive.

SET DEFAULT

You must be logged in as `SYSDBA` to specify this clause. Use this clause to designate this flashback data archive as the default flashback data archive for the system. When a `CREATE TABLE` or `ALTER TABLE` statement specifies the *`flashback_archive_clause`* without specifying a flashback data archive name, the database uses the default flashback data archive to store data from that table.

This statement overrides any previous designation of a different flashback data archive as the default.

> **See Also:**
>
> The `CREATE TABLE` *flashback_archive_clause* for more information

ADD TABLESPACE

Use this clause to add a tablespace to the flashback data archive. You can use the *`flashback_archive_quota`* clause to specify the amount of space that can be used by the flashback data archive in the new tablespace. If you omit that clause, then the flashback data archive has unlimited space in the newly added tablespace.

MODIFY TABLESPACE

Use this clause to change the tablespace quota of a tablespace already used by the flashback data archive.

REMOVE TABLESPACE

Use this clause to remove a tablespace from use by the flashback data archive. You cannot remove the last remaining tablespace used by the flashback data archive.

If the tablespace to be removed contains any data within the retention period of the flashback archive, then that data will be dropped as well. Therefore, you should move your data to another tablespace before removing the tablespace with this clause.

MODIFY RETENTION

Use this clause to change the retention period of the flashback data archive.

PURGE

Use this clause to purge data from the flashback data archive.

- Specify `PURGE ALL` to remove all data from the flashback data archive. This historical information can be retrieved using a flashback query only if the SCN or timestamp specified in the flashback query is within the undo retention duration.

- Specify PURGE BEFORE SCN to remove all data from the flashback data archive before the specified system change number.
- Specify PURGE BEFORE TIMESTAMP to remove all data from the flashback data archive before the specified timestamp.

[NO] OPTIMIZE DATA

This clause has the same semantics as the [NO] OPTIMIZE DATA clause of CREATE FLASHBACK ARCHIVE.

> **See Also:**
>
> CREATE FLASHBACK ARCHIVE for information on creating flashback data archives and for some simple examples of using flashback data archives

ALTER FUNCTION

Purpose

Functions are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the ALTER FUNCTION statement to recompile an invalid standalone stored function. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

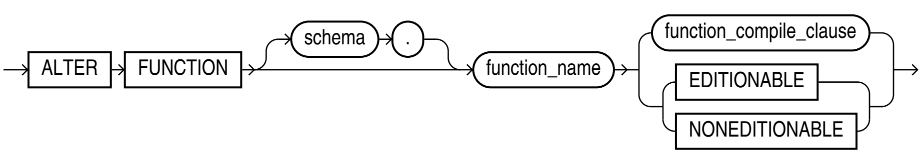
This statement does not change the declaration or definition of an existing function. To redeclare or redefine a function, use the CREATE FUNCTION statement with the OR REPLACE clause. See CREATE FUNCTION .

Prerequisites

The function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

***alter_function*::=**



(*function_compile_clause*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

schema

Specify the schema containing the function. If you omit `schema`, then Oracle Database assumes the function is in your own schema.

function_name

Specify the name of the function to be recompiled.

function_compile_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling functions.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the function becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `FUNCTION` in `schema`. The default is `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

ALTER HIERARCHY

Purpose

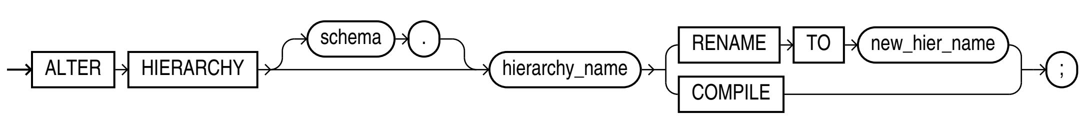
Use the `ALTER HIERARCHY` statement to rename or compile a hierarchy. For other alterations, use `CREATE OR REPLACE HIERARCHY`.

Prerequisites

To alter a hierarchy in your own schema, you must have the `ALTER HIERARCHY` system privilege. To alter a hierarchy in another user's schema, you must have the `ALTER ANY HIERARCHY` system privilege or have been granted `ALTER` directly on the hierarchy.

Syntax

***alter_hierarchy*::=**



Semantics

schema

Specify the schema in which the hierarchy exists. If you do not specify a schema, then Oracle Database looks for the hierarchy in your own schema.

hierarchy_name

Specify the name of the hierarchy.

RENAME TO

Specify `RENAME TO` to change the name of the hierarchy.

COMPILE

Specify `COMPILE` to compile the hierarchy.

new_hier_name

Specify a new name for the hierarchy.

Example

The following statement changes the name of a hierarchy:

```
ALTER HIERARCHY product_hier RENAME TO myproduct_hier;
```

ALTER INDEX

Purpose

Use the `ALTER INDEX` statement to change or rebuild an existing index.

> **See Also:**
>
> CREATE INDEX for information on creating an index

Prerequisites

The index must be in your own schema or you must have the `ALTER ANY INDEX` system privilege.

To execute the `MONITORING USAGE` clause, the index must be in your own schema.

To modify a domain index, you must have `EXECUTE` object privilege on the indextype of the index.

Object privileges are granted on the parent index, not on individual index partitions or subpartitions.

You must have tablespace quota to modify, rebuild, or split an index partition or to modify or rebuild an index subpartition.

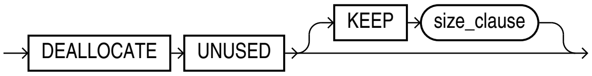
Syntax

***alter_index*::=**

(*deallocate_unused_clause*::=, *allocate_extent_clause*::=, *shrink_clause*::=, *parallel_clause*::=, *physical_attributes_clause*::=, *logging_clause*::=, *partial_index_clause*::=, *rebuild_clause*::=, *alter_index_partitioning*::=)

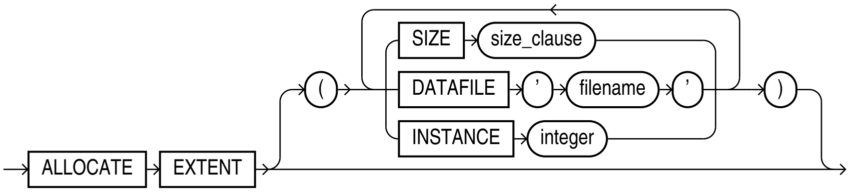
(The `ODCI_parameters` are documented in Oracle Database Data Cartridge Developer's Guide.)

***deallocate_unused_clause*::=**



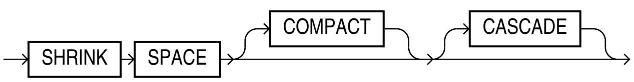
(*size_clause*::=)

***allocate_extent_clause*::=**

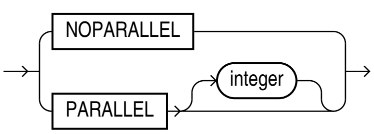


(*size_clause*::=)

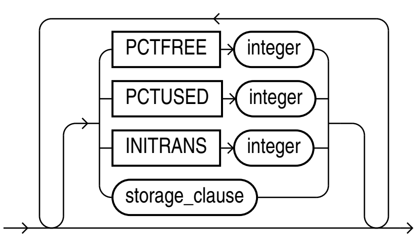
***shrink_clause*::=**



***parallel_clause*::=**

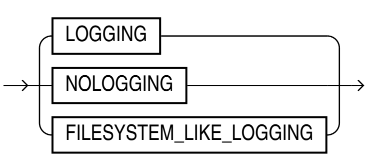


***physical_attributes_clause*::=**

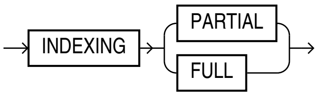


(*storage_clause*::=)

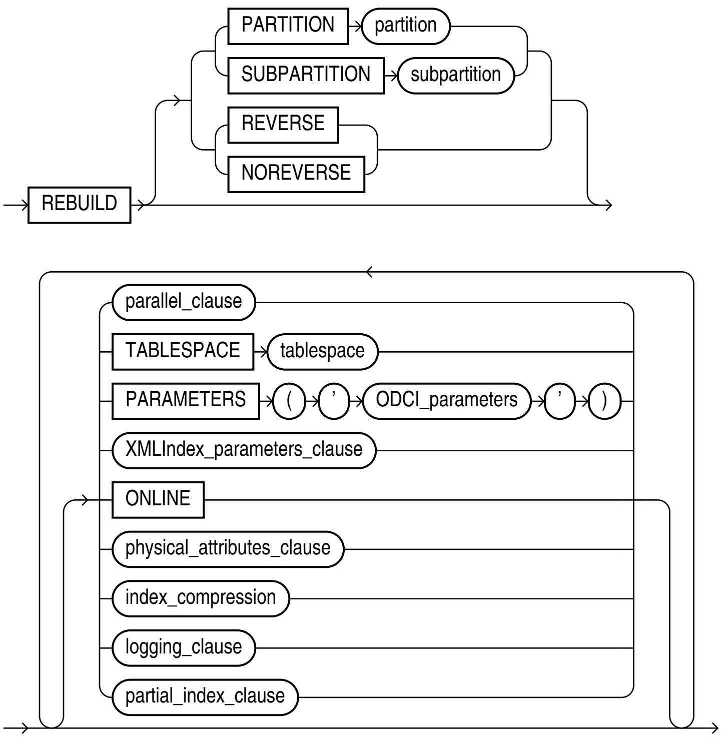
***logging_clause*::=**



***partial_index_clause*::=**



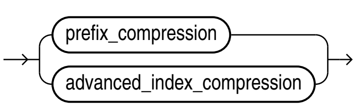
***rebuild_clause*::=**



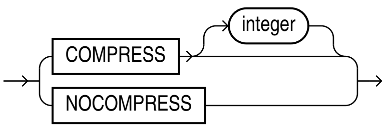
(*parallel_clause*::=, *physical_attributes_clause*::=, *index_compression*::=, *logging_clause*::=, *partial_index_clause*::=)

(The `ODCI_parameters` are documented in Oracle Database Data Cartridge Developer's Guide. The `XMLIndex_parameters_clause` is documented in *Oracle XML DB Developer's Guide*.

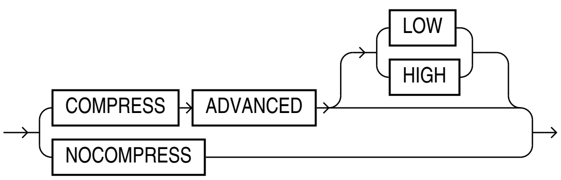
***index_compression*::=**



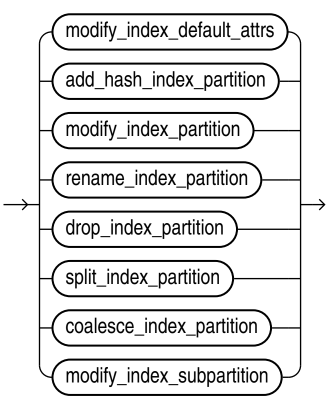
***prefix_compression*::=**



***advanced_index_compression*::=**

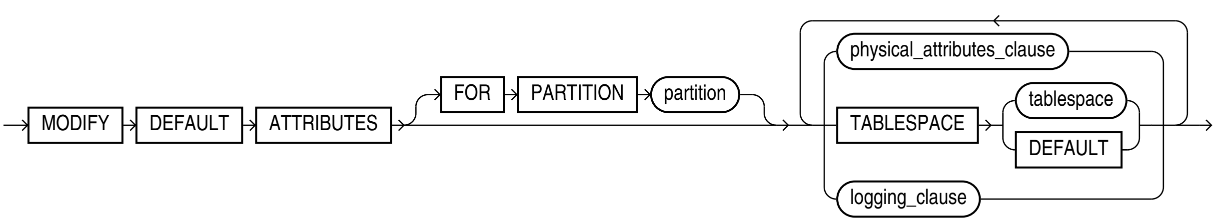


***alter_index_partitioning*::=**



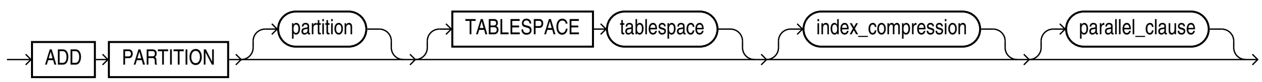
(*modify_index_default_attrs*::=, *add_hash_index_partition*::=, *modify_index_partition*::=, *rename_index_partition*::=, *drop_index_partition*::=, *split_index_partition*::=, *coalesce_index_partition*::=, *modify_index_subpartition*::=)

***modify_index_default_attrs*::=**



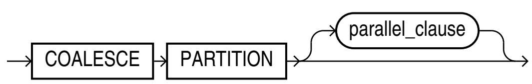
(*physical_attributes_clause*::=, *logging_clause*::=)

***add_hash_index_partition*::=**



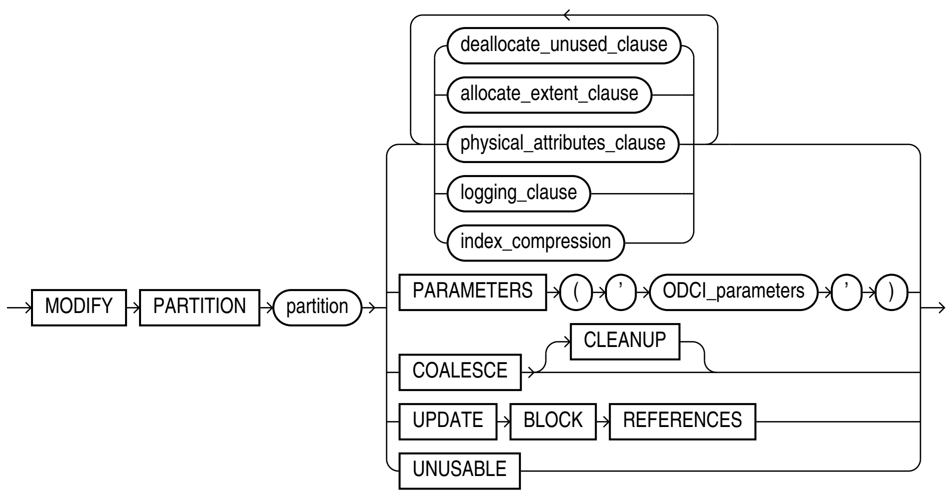
(*index_compression*::=, *parallel_clause*::=)

***coalesce_index_partition*::=**



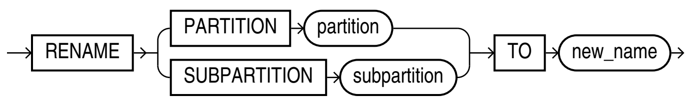
(*parallel_clause*::=)

***modify_index_partition*::=**



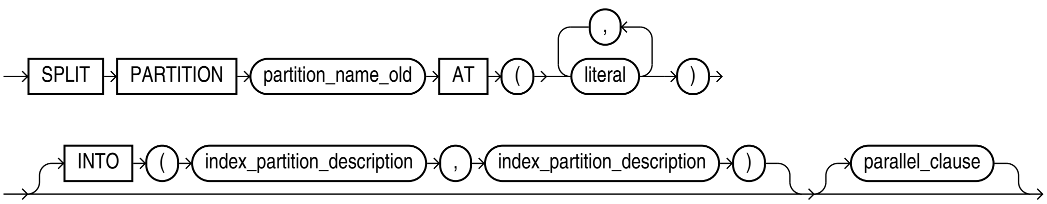
(*deallocate_unused_clause*::=, *allocate_extent_clause*::=, *physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=)

***rename_index_partition*::=**



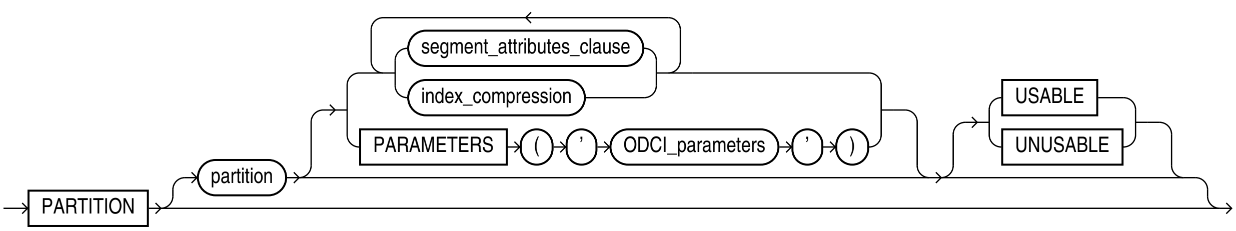
***drop_index_partition*::=**

***split_index_partition*::=**



(*parallel_clause::=*)

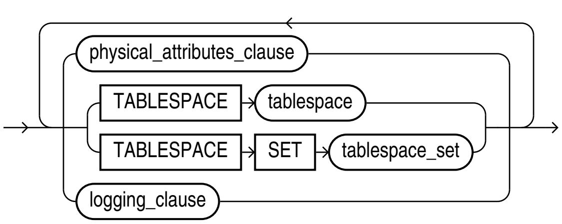
***index_partition_description*::=**



(*segment_attributes_clause::=*, *index_compression::=*)

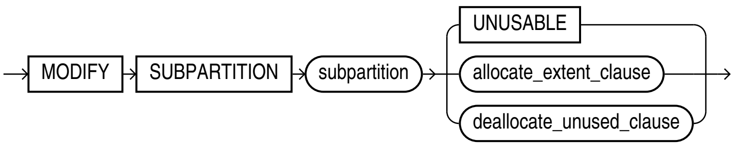
> **Note:**
>
> The `USABLE` and `UNUSABLE` keywords are not supported when *`index_partition_description`* is specified for the *`split_index_partition`* clause.

***segment_attributes_clause*::=**



(*physical_attributes_clause*::=, TABLESPACE SET: not supported with ALTER INDEX, *logging_clause*::=)

***modify_index_subpartition*::=**



(*allocate_extent_clause*::=, *deallocate_unused_clause*::=)

Semantics

schema

Specify the schema containing the index. If you omit *schema*, then Oracle Database assumes the index is in your own schema.

index

Specify the name of the index to be altered.

Restrictions on Modifying Indexes

The modification of indexes is subject to the following restrictions:

- If *index* is a domain index, then you can specify only the PARAMETERS clause, the RENAME clause, the *rebuild_clause* (with or without the PARAMETERS clause), the *parallel_clause*, or the UNUSABLE clause. No other clauses are valid.
- You cannot alter or rename a domain index that is marked LOADING or FAILED. If an index is marked FAILED, then the only clause you can specify is REBUILD.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information on the LOADING and FAILED states of domain indexes

deallocate_unused_clause

Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of the index and make the freed space available for other segments in the tablespace.

If *index* is range-partitioned or hash-partitioned, then Oracle Database deallocates unused space from each index partition. If *index* is a local index on a composite-partitioned table, then Oracle Database deallocates unused space from each index subpartition.

Restrictions on Deallocating Space

Deallocation of space is subject to the following restrictions:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify this clause and also specify the *rebuild_clause*.

Refer to *deallocate_unused_clause* for a full description of this clause.

KEEP *integer*

The KEEP clause lets you specify the number of bytes above the high water mark that the index will have after deallocation. If the number of remaining extents is less than MINEXTENTS, then MINEXTENTS is set to the current number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is set to the value of the current initial extent. If you omit KEEP, then all unused space is freed.

Refer to ALTER TABLE for a complete description of this clause.

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocate a new extent for the index. For a local index on a hash-partitioned table, Oracle Database allocates a new extent for each partition of the index.

Restriction on Allocating Extents

You cannot specify this clause for an index on a temporary table or for a range-partitioned or composite-partitioned index.

Refer to *allocate_extent_clause* for a full description of this clause.

shrink_clause

Use this clause to compact the index segments. Specifying ALTER INDEX ... SHRINK SPACE COMPACT is equivalent to specifying ALTER INDEX ... COALESCE.

For complete information on this clause, refer to *shrink_clause* in the documentation on CREATE TABLE.

Restriction on Shrinking Index Segments

You cannot specify this clause for a bitmap join index or for a function-based index.

parallel_clause

Use the PARALLEL clause to change the default degree of parallelism for queries and DML on the index.

Restriction on Parallelizing Indexes

You cannot specify this clause for an index on a temporary table.

For complete information on this clause, refer to *parallel_clause* in the documentation on CREATE TABLE.

> **See Also:**
>
> "Enabling Parallel Queries: Example"

physical_attributes_clause

Use the `physical_attributes_clause` to change the values of parameters for a nonpartitioned index, all partitions and subpartitions of a partitioned index, a specified partition, or all subpartitions of a specified partition.

> **See Also:**
>
> - the physical attributes parameters in CREATE TABLE
> - "Modifying Real Index Attributes: Example" and "Changing MAXEXTENTS: Example"

Restrictions on Index Physical Attributes

Index physical attributes are subject to the following restrictions:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify the `PCTUSED` parameter at all when altering an index.
- You can specify the `PCTFREE` parameter only as part of the `rebuild_clause`, the `modify_index_default_attrs` clause, or the `split_index_partition` clause.

storage_clause

Use the `storage_clause` to change the storage parameters for a nonpartitioned index, index partition, or all partitions of a partitioned index, or default values of these parameters for a partitioned index. Refer to storage_clause for complete information on this clause.

logging_clause

Use the `logging_clause` to change the logging attribute of the index. If you also specify the `REBUILD` clause, then this new setting affects the rebuild operation. If you specify a different value for logging in the `REBUILD` clause, then Oracle Database uses the last logging value specified as the logging attribute of the index and of the rebuild operation.

An index segment can have logging attributes different from those of the base table and different from those of other index segments for the same base table.

Restriction on Index Logging

You cannot specify this clause for an index on a temporary table.

> **See Also:**
>
> - *logging_clause* for a full description of this clause
> - *Oracle Database VLDB and Partitioning Guide* for more information about parallel DML

partial_index_clause

Use the *partial_index_clause* to change the index to a full index or a partial index. Specify INDEXING FULL to change the index to a full index. Specify INDEXING PARTIAL to change the index to a partial index. This clause is valid only for indexes on partitioned tables. Refer to the *partial_index_clause* of CREATE INDEX for the full semantics of this clause.

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with LOGGING and NOLOGGING, respectively. Although RECOVERABLE and UNRECOVERABLE are supported for backward compatibility, Oracle strongly recommends that you use the LOGGING and NOLOGGING keywords.

RECOVERABLE is not a valid keyword for creating partitioned tables or LOB storage characteristics. UNRECOVERABLE is not a valid keyword for creating partitioned or index-organized tables. Also, it can be specified only with the AS subquery clause of CREATE INDEX.

rebuild_clause

Use the *rebuild_clause* to re-create an existing index or one of its partitions or subpartitions. If index is marked UNUSABLE, then a successful rebuild will mark it USABLE. For a function-based index, this clause also enables the index. If the function on which the index is based does not exist, then the rebuild statement will fail.

> **Note:**
>
> When you rebuild the secondary index of an index-organized table, Oracle Database preserves the primary key columns contained in the logical rowid when the index was created. Therefore, if the index was created with the COMPATIBLE initialization parameter set to less than 10.0.0, the rebuilt index will contain the index key and any of the primary key columns of the table that are not also in the index key. If the index was created with the COMPATIBLE initialization parameter set to 10.0.0 or greater, then the rebuilt index will contain the index key and all the primary key columns of the table, including those also in the index key.

Restrictions on Rebuilding Indexes

The rebuilding of indexes is subject to the following restrictions:

- You cannot rebuild an index on a temporary table.
- You cannot rebuild a bitmap index that is marked INVALID. Instead, you must drop and then re-create it.
- You cannot rebuild an entire partitioned index. You must rebuild each partition or subpartition, as described for the PARTITION clause.
- You cannot specify the *deallocate_unused_clause* in the same statement as the *rebuild_clause*.
- You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or for a partition (ALTER INDEX ... MODIFY PARTITION). You can specify PCTFREE in all other forms of the ALTER INDEX statement.

- For a domain index:
  - You can specify only the `PARAMETERS` clause (either for the index or for a partition of the index) or the *`parallel_clause`*. No other rebuild clauses are valid.
  - You can rebuild an index only if the index is not marked `IN_PROGRESS`.
  - You can rebuild an index partition only if the index is not marked `IN_PROGRESS` or `FAILED` and the partition is not marked `IN_PROGRESS`.
- You cannot rebuild a local index, but you can rebuild a partition of a local index (`ALTER INDEX` ... `REBUILD PARTITION`).
- For a local index on a hash partition or subpartition, the only parameter you can specify is `TABLESPACE`.
- You cannot rebuild an online index that is used to enforce a deferrable unique constraint.

PARTITION Clause

Use the `PARTITION` clause to rebuild one partition of an index. You can also use this clause to move an index partition to another tablespace or to change a create-time physical attribute.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

Restriction on Rebuilding Partitions

You cannot specify this clause for a local index on a composite-partitioned table. Instead, use the `REBUILD SUBPARTITION` clause.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information about partition maintenance operations and "Rebuilding Unusable Index Partitions: Example"

SUBPARTITION Clause

Use the `SUBPARTITION` clause to rebuild one subpartition of an index. You can also use this clause to move an index subpartition to another tablespace. If you do not specify `TABLESPACE`, then the subpartition is rebuilt in the same tablespace.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

Restriction on Modifying Index Subpartitions

The only parameters you can specify for a subpartition are `TABLESPACE`, `ONLINE`, and the *`parallel_clause`*.

REVERSE | NOREVERSE

Indicate whether the bytes of the index block are stored in reverse order:

- `REVERSE` stores the bytes of the index block in reverse order and excludes the rowid when the index is rebuilt.
- `NOREVERSE` stores the bytes of the index block without reversing the order when the index is rebuilt. Rebuilding a `REVERSE` index without the `NOREVERSE` keyword produces a rebuilt, reverse-keyed index.

Restrictions on Reverse Indexes

Reverse indexes are subject to the following restrictions:

- You cannot reverse a bitmap index or an index-organized table.
- You cannot specify `REVERSE` or `NOREVERSE` for a partition or subpartition.

> **See Also:**
>
> "Storing Index Blocks in Reverse Order: Example"

parallel_clause

Use the *`parallel_clause`* to parallelize the rebuilding of the index and to change the degree of parallelism for the index itself. All subsequent operations on the index will be executed with the degree of parallelism specified by this clause, unless overridden by a subsequent data definition language (DDL) statement with the *`parallel_clause`*. The following exceptions apply:

- If `ALTER SESSION DISABLE PARALLEL DDL` was specified before rebuilding the index, then the index will be rebuilt serially and the degree of parallelism for the index will be changed to 1.
- If `ALTER SESSION FORCE PARALLEL DDL` was specified before rebuilding the index, then the index will be rebuilt in parallel and the degree of parallelism for the index will be changed to the value that was specified in the `ALTER SESSION` statement, or `DEFAULT` if no value was specified.

> **See Also:**
>
> "Rebuilding an Index in Parallel: Example"

TABLESPACE Clause

Specify the tablespace where the rebuilt index, index partition, or index subpartition will be stored. The default is the default tablespace where the index or partition resided before you rebuilt it.

index_compression

Use the *index_compression* clauses to enable or disable index compression for the index. Specify the *prefix_compression* clause to enable or disable prefix compression for the index. Specify the *advanced_index_compression* clause to enable or disable advanced index compression for the index.

The *index_compression* clauses have the same semantics for CREATE INDEX and ALTER INDEX. For full information on these clauses, refer to *index_compression* in the documentation on CREATE INDEX.

ONLINE Clause

Specify ONLINE to allow DML operations on the table or partition during rebuilding of the index.

Restrictions on Online Indexes

Online indexes are subject to the following restrictions:

- Parallel DML is not supported during online index building. If you specify ONLINE and subsequently issue parallel DML statements, then Oracle Database returns an error.
- You cannot specify ONLINE for a bitmap join index or a cluster index.
- For a nonunique secondary index on an index-organized table, the number of index key columns plus the number of primary key columns that are included in the logical rowid in the index-organized table cannot exceed 32. The logical rowid excludes columns that are part of the index key.

logging_clause

Specify whether the ALTER INDEX ... REBUILD operation will be logged.

Refer to the *logging_clause* for a full description of this clause.

PARAMETERS Clause

This clause is valid only for domain indexes in a top-level ALTER INDEX statement and in the *rebuild_clause*. This clause specifies the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine.

The maximum length of the parameter string is 1000 characters.

If you are altering or rebuilding an entire index, then the string must refer to index-level parameters. If you are rebuilding a partition of the index, then the string must refer to partition-level parameters.

If *index* is marked UNUSABLE, then modifying the parameters alone does not make it USABLE. You must also rebuild the UNUSABLE index to make it usable.

If you have installed Oracle Text, then you can rebuild your Oracle Text domain indexes using parameters specific to that product. For more information on those parameters, refer to *Oracle Text Reference*.

Restriction on the PARAMETERS Clause

You can modify index partitions only if *index* is not marked IN_PROGRESS or FAILED, no index partitions are marked IN_PROGRESS, and the partition being modified is not marked FAILED.

> **See Also:**
>
> - *Oracle Database Data Cartridge Developer's Guide* for more information on indextype routines for domain indexes
> - CREATE INDEX for more information on domain indexes

XMLIndex_parameters_clause

This clause is valid only for XMLIndex indexes. This clause specifies the parameter string that defines the XMLIndex implementation.

The maximum length of the parameter string is 1000 characters.

If you are altering or rebuilding an entire index, then the string must refer to index-level parameters. If you are rebuilding a partition of the index, then the string must refer to partition-level parameters.

If *index* is marked UNUSABLE, then modifying the parameters alone does not make it USABLE. You must also rebuild the UNUSABLE index to make it usable.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on XMLIndex, including the syntax and semantics of the *XMLIndex_parameters_clause*

Restriction on the *XMLIndex_parameters_clause*

You can modify index partitions only if *index* is not marked IN_PROGRESS or FAILED, no index partitions are marked IN_PROGRESS, and the partition being modified is not marked FAILED.

{ DEFERRED | IMMEDIATE } INVALIDATION

This clause lets you control when the database invalidates dependent cursors while rebuilding an index or while marking an index UNUSABLE.

- If you specify DEFERRED INVALIDATION, then the database avoids or defers invalidating dependent cursors, when possible.
- If you specify IMMEDIATE INVALIDATION, then the database immediately invalidates dependent cursors, as it did in Oracle Database 12*c* Release 1 (12.1) and prior releases. This is the default.

If you omit this clause, then the value of the CURSOR_INVALIDATION initialization parameter determines when cursors are invalidated.

See Also:

- *Oracle Database SQL Tuning Guide* for more information on cursor invalidation
- *Oracle Database Reference* for more information in the `CURSOR_INVALIDATION` initialization parameter

COMPILE Clause

Use this clause to recompile an invalid index explicitly. For domain indexes, this clause is useful when the underlying indextype has been altered to support system-managed domain indexes, so that the existing domain index has been marked `INVALID`. In this situation, this `ALTER INDEX` statement migrates the domain index from a user-managed domain index to a system-managed domain index. For all types of indexes, this clause is useful when an index has been marked `INVALID` by an `ALTER TABLE` statement. In this situation, this `ALTER INDEX` statement revalidates the index without rebuilding it.

See Also:

The `CREATE INDEXTYPE` *storage_table_clause* and *Oracle Database Data Cartridge Developer's Guide* for information on creating system-managed domain indexes

ENABLE Clause

`ENABLE` applies only to a function-based index that has been disabled, either by an `ALTER INDEX` ... `DISABLE` statement, or because a user-defined function used by the index was dropped or replaced. This clause enables such an index if these conditions are true:

- The function is currently valid.
- The signature of the current function matches the signature of the function when the index was created.
- The function is currently marked as `DETERMINISTIC`.

Restrictions on Enabling Function-based Indexes

The `ENABLE` clause is subject to the following restrictions:

- You cannot specify any other clauses of `ALTER INDEX` in the same statement with `ENABLE`.
- You cannot specify this clause for an index on a temporary table. Instead, you must drop and recreate the index. You can retrieve the creation DDL for the index using the `DBMS_METADATA` package.

DISABLE Clause

`DISABLE` applies only to a function-based index. This clause lets you disable the use of a function-based index. You might want to do so, for example, while working on the body of the function. Afterward you can either rebuild the index or specify another ALTER INDEX statement with the ENABLE keyword.

USABLE | UNUSABLE

Specify UNUSABLE to mark the index or index partition(s) or index subpartition(s) UNUSABLE. The space allocated for an index or index partition or subpartition is freed immediately when the object is marked UNUSABLE. An unusable index must be rebuilt, or dropped and re-created, before it can be used. While one partition is marked UNUSABLE, the other partitions of the index are still valid. You can execute statements that require the index if the statements do not access the unusable partition. You can also split or rename the unusable partition before rebuilding it. Refer to CREATE INDEX ... USABLE | UNUSABLE for more information.

ONLINE

Specify ONLINE to indicate that DML operations on the table or partition will be allowed while marking the index UNUSABLE. If you specify this clause, then the database will not drop the index segments.

Restrictions on Marking Indexes Unusable

The following restrictions apply to marking indexes unusable:

- You cannot specify UNUSABLE for an index on a temporary table.
- When a global index is marked UNUSABLE during a partition maintenance operation, the database does not drop the unusable index segments.

VISIBLE | INVISIBLE

Use this clause to specify whether the index is visible or invisible to the optimizer. Refer to "VISIBLE | INVISIBLE" in CREATE INDEX for a full description of this clause.

RENAME Clause

Use this clause to rename an index. The *new_index_name* is a single identifier and does not include the schema name.

Restriction on Renaming Indexes

For a domain index, neither *index* nor any partitions of *index* can be marked IN_PROGRESS or FAILED.

> **See Also:**
>
> "Renaming an Index: Example"

COALESCE Clause

Specify COALESCE to instruct Oracle Database to merge the contents of index blocks where possible to free blocks for reuse.

CLEANUP

Specify CLEANUP to remove orphaned index entries for records that were previously dropped or truncated by a table partition maintenance operation.

To determine whether an index contains orphaned index entries, you can query the ORPHANED_ENTRIES column of the USER_, DBA_, ALL_INDEXES data dictionary views. Refer to *Oracle Database Reference* for more information.

parallel_clause

Use the *parallel_clause* to specify whether to parallelize the coalesce operation.

For complete information on this clause, refer to *parallel_clause* in the documentation on CREATE TABLE.

Restrictions on Coalescing Index Blocks

Coalescing of index blocks is subject to the following restrictions:

- You cannot specify this clause for an index on a temporary table.
- Do not specify this clause for the primary key index of an index-organized table. Instead use the COALESCE clause of ALTER TABLE.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information on space management and coalescing indexes
> - COALESCE Clause for information on coalescing the space of an index-organized table
> - *shrink_clause* for an alternative method of compacting index segments

MONITORING USAGE | NOMONITORING USAGE

Use this clause to determine whether Oracle Database should monitor index use.

- Specify MONITORING USAGE to begin monitoring the index. Oracle Database first clears existing information on index use, and then monitors the index for use until a subsequent ALTER INDEX ... NOMONITORING USAGE statement is executed.
- To terminate monitoring of the index, specify NOMONITORING USAGE.

To see whether the index has been used since this ALTER INDEX ... NOMONITORING USAGE statement was issued, query the USED column of the USER_OBJECT_USAGE data dictionary view.

> **See Also:**
>
> *Oracle Database Reference* for information on the USER_OBJECT_USAGE data dictionary view

UPDATE BLOCK REFERENCES Clause

The UPDATE BLOCK REFERENCES clause is valid only for normal and domain indexes on index-organized tables. Specify this clause to update all the stale guess data block addresses stored as part of the index row with the correct database address for the corresponding block identified by the primary key.

For a domain index, Oracle Database executes the `ODCIIndexAlter` routine with the `alter_option` parameter set to `AlterIndexUpdBlockRefs`. This routine enables the cartridge code to update the stale guess data block addresses in the index.

Restriction on UPDATE BLOCK REFERENCES

You cannot combine this clause with any other clause of `ALTER INDEX`.

alter_index_partitioning

The partitioning clauses of the `ALTER INDEX` statement are valid only for partitioned indexes.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

Restrictions on Modifying Index Partitions

Modifying index partitions is subject to the following restrictions:

- You cannot specify any of these clauses for an index on a temporary table.
- You can combine several operations on the base index into one `ALTER INDEX` statement (except `RENAME` and `REBUILD`), but you cannot combine partition operations with other partition operations or with operations on the base index.

modify_index_default_attrs

Specify new values for the default attributes of a partitioned index.

Restriction on Modifying Partition Default Attributes

The only attribute you can specify for a hash-partitioned global index or for an index on a hash-partitioned table is `TABLESPACE`.

TABLESPACE

Specify the default tablespace for new partitions of an index or subpartitions of an index partition.

logging_clause

Specify the default logging attribute of a partitioned index or an index partition.

Refer to *logging_clause* for a full description of this clause.

FOR PARTITION

Use the `FOR PARTITION` clause to specify the default attributes for the subpartitions of a partition of a local index on a composite-partitioned table.

Restriction on FOR PARTITION

You cannot specify `FOR PARTITION` for a list partition.

> **See Also:**
>
> "Modifying Default Attributes: Example"

add_hash_index_partition

Use this clause to add a partition to a global hash-partitioned index. Oracle Database adds hash partitions and populates them with index entries rehashed from an existing hash partition of the index, as determined by the hash function. If you omit the partition name, then Oracle Database assigns a name of the form `SYS_Pn`. If you omit the `TABLESPACE` clause, then Oracle Database places the partition in the tablespace specified for the index. If no tablespace is specified for the index, then Oracle Database places the partition in the default tablespace of the user, if one has been specified, or in the system default tablespace.

modify_index_partition

Use the *`modify_index_partition`* clause to modify the real physical attributes, logging attribute, or storage characteristics of index partition *`partition`* or its subpartitions. For a hash-partitioned global index, the only subclause of this clause you can specify is `UNUSABLE`.

COALESCE

Specify this clause to merge the contents of index partition blocks where possible to free blocks for reuse.

CLEANUP

Specify `CLEANUP` to remove orphaned index entries for records that were previously dropped or truncated by a table partition maintenance operation.

To determine whether an index partition contains orphaned index entries, you can query the `ORPHANED_ENTRIES` column of the `USER_`, `DBA_`, `ALL_PART_INDEXES` data dictionary views. Refer to *Oracle Database Reference* for more information.

UPDATE BLOCK REFERENCES

The `UPDATE BLOCK REFERENCES` clause is valid only for normal indexes on index-organized tables. Use this clause to update all stale guess data block addresses stored in the secondary index partition.

Restrictions on UPDATE BLOCK REFERENCES

This clause is subject to the following restrictions:

- You cannot specify the *`physical_attributes_clause`* for an index on a hash-partitioned table.
- You cannot specify `UPDATE BLOCK REFERENCES` with any other clause in `ALTER INDEX`.

> **Note:**
>
> If the index is a local index on a composite-partitioned table, then the changes you specify here will override any attributes specified earlier for the subpartitions of index, as well as establish default values of attributes for future subpartitions of that partition. To change the default attributes of the partition without overriding the attributes of subpartitions, use `ALTER TABLE` ... `MODIFY DEFAULT ATTRIBUTES FOR PARTITION`.

> **See Also:**
>
> "Marking an Index Unusable: Examples"

UNUSABLE Clause

This clause has the same function for index partitions that it has for the index as a whole. Refer to "USABLE | UNUSABLE".

index_compression

This clause is relevant for composite-partitioned indexes. Use this clause to change the compression attribute for the partition and every subpartition in that partition. Oracle Database marks each index subpartition in the partition `UNUSABLE` and you must then rebuild these subpartitions. Prefix compression must already have been specified for the index before you can specify the *`prefix_compression`* clause for a partition, or advanced index compression must have already been specified for the index before you can specify the *`advanced_index_compression`* clause for a partition. You can specify this clause only at the partition level. You cannot change the compression attribute for an individual subpartition.

You can use this clause for noncomposite index partitions. However, it is more efficient to use the *`rebuild_clause`* for noncomposite partitions, which lets you rebuild and set the compression attribute in one step.

rename_index_partition

Use the *`rename_index_partition`* clauses to rename index *`partition`* or *`subpartition`* to *`new_name`*.

Restrictions on Renaming Index Partitions

Renaming index partitions is subject to the following restrictions:

- You cannot rename the subpartition of a list partition.
- For a partition of a domain index, *`index`* cannot be marked `IN_PROGRESS` or `FAILED`, none of the partitions can be marked `IN_PROGRESS`, and the partition you are renaming cannot be marked `FAILED`.

> **See Also:**
>
> "Renaming an Index Partition: Example"

drop_index_partition

Use the *drop_index_partition* clause to remove a partition and the data in it from a partitioned global index. When you drop a partition of a global index, Oracle Database marks the next index partition UNUSABLE. You cannot drop the highest partition of a global index.

> **See Also:**
>
> "Dropping an Index Partition: Example"

split_index_partition

Use the *split_index_partition* clause to split a partition of a global range-partitioned index into two partitions, adding a new partition to the index. This clause is not valid for hash-partitioned global indexes. Instead, use the *add_hash_index_partition* clause.

Splitting a partition marked UNUSABLE results in two partitions, both marked UNUSABLE. You must rebuild the partitions before you can use them.

Splitting a partition marked USABLE results in two partitions populated with index data. Both new partitions are marked USABLE.

AT Clause

Specify the new noninclusive upper bound for *split_partition_1*. The *value_list* must evaluate to less than the presplit partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one).

INTO Clause

Specify (optionally) the name and physical attributes of each of the two partitions resulting from the split.

> **See Also:**
>
> "Splitting a Partition: Example"

coalesce_index_partition

This clause is valid only for hash-partitioned global indexes. Oracle Database reduces by one the number of index partitions. Oracle Database selects the partition to coalesce based on the requirements of the hash function. Use this clause if you want to distribute index entries of a selected partition into one of the remaining partitions and then remove the selected partition.

modify_index_subpartition

Use the *modify_index_subpartition* clause to mark UNUSABLE or allocate or deallocate storage for a subpartition of a local index on a composite-partitioned table. All other attributes of such a subpartition are inherited from partition-level default attributes.

Examples

Storing Index Blocks in Reverse Order: Example

The following statement rebuilds index ord_customer_ix (created in "Creating an Index: Example") so that the bytes of the index block are stored in reverse order:

```
ALTER INDEX ord_customer_ix REBUILD REVERSE;
```

Rebuilding an Index in Parallel: Example

The following statement causes the index to be rebuilt from the existing index by using parallel execution processes to scan the old and to build the new index:

```
ALTER INDEX ord_customer_ix REBUILD PARALLEL;
```

Modifying Real Index Attributes: Example

The following statement alters the oe.cust_lname_ix index so that future data blocks within this index use 5 initial transaction entries:

```
ALTER INDEX oe.cust_lname_ix
    INITRANS 5;
```

If the oe.cust_lname_ix index were partitioned, then this statement would also alter the default attributes of future partitions of the index. Partitions added in the future would then use 5 initial transaction entries and an incremental extent of 100K.

Enabling Parallel Queries: Example

The following statement sets the parallel attributes for index upper_ix (created in "Creating a Function-Based Index: Example") so that scans on the index will be parallelized:

```
ALTER INDEX upper_ix PARALLEL;
```

Renaming an Index: Example

The following statement renames an index:

```
ALTER INDEX upper_ix RENAME TO upper_name_ix;
```

Marking an Index Unusable: Examples

The following statements use the cost_ix index, which was created in "Creating a Range-Partitioned Global Index: Example". Partition p1 of that index was dropped in "Dropping an Index Partition: Example". The first statement marks index partition p2 as UNUSABLE:

```
ALTER INDEX cost_ix
   MODIFY PARTITION p2 UNUSABLE;
```

The next statement marks the entire index cost_ix as UNUSABLE:

```
ALTER INDEX cost_ix UNUSABLE;
```

Rebuilding Unusable Index Partitions: Example

The following statements rebuild partitions p2 and p3 of the cost_ix index, making the index once more usable: The rebuilding of partition p3 will not be logged:

```
ALTER INDEX cost_ix
   REBUILD PARTITION p2;
ALTER INDEX cost_ix
   REBUILD PARTITION p3 NOLOGGING;
```

Changing MAXEXTENTS: Example

The following statement changes the maximum number of extents for partition p3 and changes the logging attribute:

```
/* This example will fail if the tablespace in which partition p3
   resides is locally managed.
*/
ALTER INDEX cost_ix MODIFY PARTITION p3
   STORAGE(MAXEXTENTS 30) LOGGING;
```

Renaming an Index Partition: Example

The following statement renames an index partition of the cost_ix index (created in "Creating a Range-Partitioned Global Index: Example"):

```
ALTER INDEX cost_ix
   RENAME PARTITION p3 TO p3_Q3;
```

Splitting a Partition: Example

The following statement splits partition p2 of index cost_ix (created in "Creating a Range-Partitioned Global Index: Example") into p2a and p2b:

```
ALTER INDEX cost_ix
   SPLIT PARTITION p2 AT (1500)
   INTO ( PARTITION p2a TABLESPACE tbs_01 LOGGING,
          PARTITION p2b TABLESPACE tbs_02);
```

Dropping an Index Partition: Example

The following statement drops index partition p1 from the cost_ix index:

```
ALTER INDEX cost_ix
   DROP PARTITION p1;
```

Modifying Default Attributes: Example

The following statement alters the default attributes of local partitioned index prod_idx, which was created in "Creating an Index on a Hash-Partitioned Table: Example". Partitions added in the future will use 5 initial transaction entries:

```
ALTER INDEX prod_idx
      MODIFY DEFAULT ATTRIBUTES INITRANS 5;
```

ALTER INDEXTYPE

Purpose

Use the `ALTER INDEXTYPE` statement to add or drop an operator of the indextype or to modify the implementation type or change the properties of the indextype.

Prerequisites

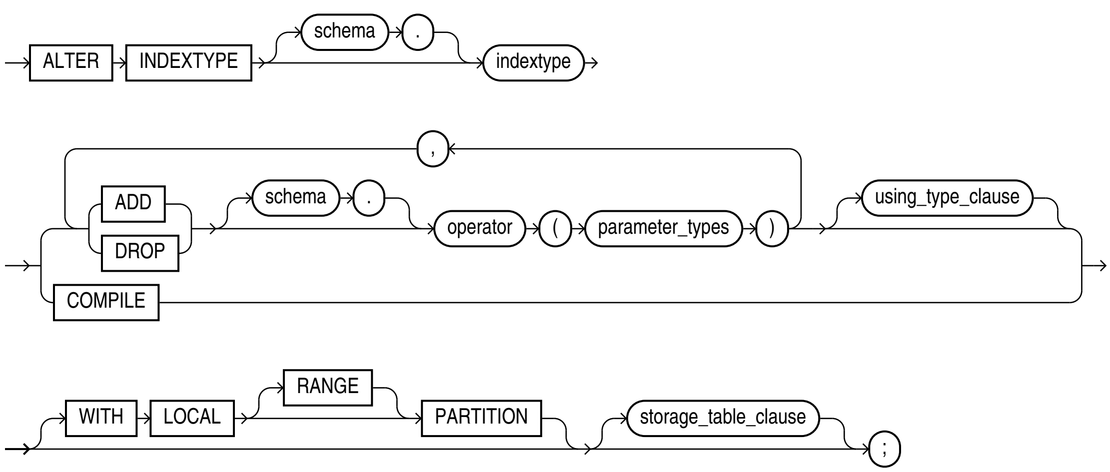
The indextype must be in your own schema or you must have the `ALTER ANY INDEXTYPE` system privilege.

To add a new operator, you must have the `EXECUTE` object privilege on the operator.

To change the implementation type, you must have the `EXECUTE` object privilege on the new implementation type.

Syntax

***alter_indextype*::=**

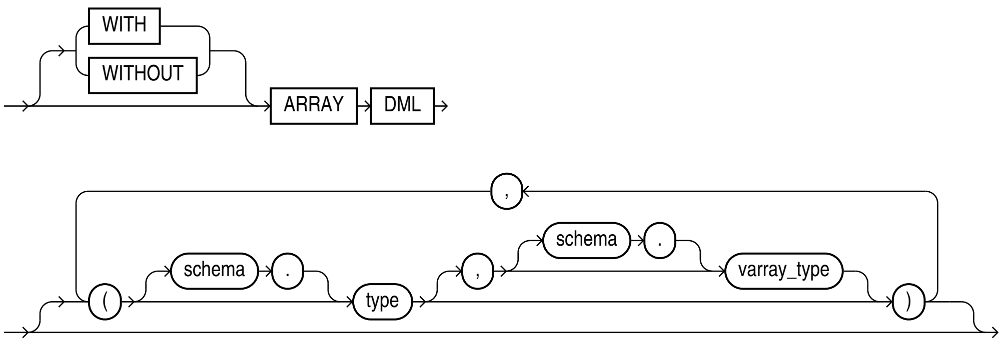


(*using_type_clause*::=, *storage_table_clause*)

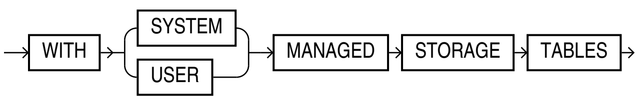
***using_type_clause*::=**

(*array_DML_clause*)

array_DML_clause



storage_table_clause



Semantics

schema

Specify the name of the schema in which the indextype resides. If you omit *`schema`*, then Oracle Database assumes the indextype is in your own schema.

indextype

Specify the name of the indextype to be modified.

ADD | DROP

Use the `ADD` or `DROP` clause to add or drop an operator.

No special privilege needed to drop.

- For *`schema`*, specify the schema containing the operator. If you omit *`schema`*, then Oracle assumes the operator is in your own schema.
- For *`operator`*, specify the name of the operator supported by the indextype.

  All the operators listed in this clause must be valid operators.
- For *`parameter_type`*, list the types of parameters to the operator.

using_type_clause

The `USING` clause lets you specify a new type to provide the implementation for the indextype.

array_DML_clause

Use this clause to modify the indextype to support the array interface for the `ODCIIndexInsert` method.

type* and *varray_type

If the data type of the column to be indexed is a user-defined object type, then you must specify this clause to identify the varray *`varray_type`* that Oracle should use to hold column values of *`type`*. If the indextype supports a list of types, then you can specify a corresponding list of varray types. If you omit *`schema`* for either *`type`* or *`varray_type`*, then Oracle assumes the type is in your own schema.

If the data type of the column to be indexed is a built-in system type, then any varray type specified for the indextype takes precedence over the ODCI types defined by the system.

COMPILE

Use this clause to recompile the indextype explicitly. This clause is required only after some upgrade operations, because Oracle Database normally recompiles the indextype automatically.

storage_table_clause

This clause has the same behavior when altering an indextype that it has when you are creating an indextype. Refer to the `CREATE INDEXTYPE` *storage_table_clause* for more information.

WITH LOCAL PARTITION

This clause has the same behavior when altering an indextype that it has when you create an indextype. Refer to the `CREATE INDEXTYPE` clause WITH LOCAL PARTITION for more information.

Examples

Altering an Indextype: Example

The following example compiles the `position_indextype` indextype created in "Creating an Indextype: Example".

```
ALTER INDEXTYPE position_indextype COMPILE;
```

ALTER INMEMORY JOIN GROUP

Purpose

Use the `ALTER INMEMORY JOIN GROUP` statement to add a table column to a join group or remove a table column from a join group.

See Also:

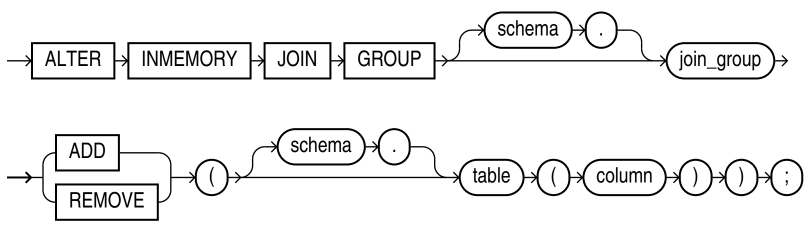
- CREATE INMEMORY JOIN GROUP and DROP INMEMORY JOIN GROUP
- *Oracle Database In-Memory Guide* for more information on join groups

Prerequisites

If the join group is not in your own schema, or if the column you want to add to or remove from the join group is in a table that is not in your own schema, then you must have the `ALTER ANY TABLE` system privilege.

Syntax

***alter_inmemory_join_group*::=**



Semantics

schema

Specify the schema containing the join group. If you omit *`schema`*, then the database assumes the join group is in your own schema.

join_group

Specify the name of the join group to be modified.

You can view existing join groups by querying the `DBA_JOINGROUPS` or `USER_JOINGROUPS` data dictionary view. Refer to *Oracle Database Reference* for more information on these views.

ADD

Specify `ADD` to add a table column to the join group. A join group can contain a maximum of 255 columns.

REMOVE

Specify `REMOVE` to remove a table column from the join group. A join group must contain at least 2 columns.

schema

Specify the schema of the table that contains the column to be added to or removed from the join group. If you omit *schema*, then Oracle Database assumes the table is in your own schema.

table

Specify the name of the table that contains the column to be added to or removed from the join group.

column

Specify the name of the column to be added to or removed from the join group.

Examples

The following example adds a column to the prod_id1 join group created in Examples in the documentation on CREATE INMEMORY JOIN GROUP:

```
ALTER INMEMORY JOIN GROUP prod_id1
  ADD(product_descriptions(product_id));
```

The following example removes a column from the prod_id1 join group:

```
ALTER INMEMORY JOIN GROUP prod_id1
  REMOVE(product_descriptions(product_id));
```

ALTER JAVA

Purpose

Use the ALTER JAVA statement to force the resolution of a Java class schema object or compilation of a Java source schema object. (You cannot call the methods of a Java class before all its external references to Java names are associated with other classes.)

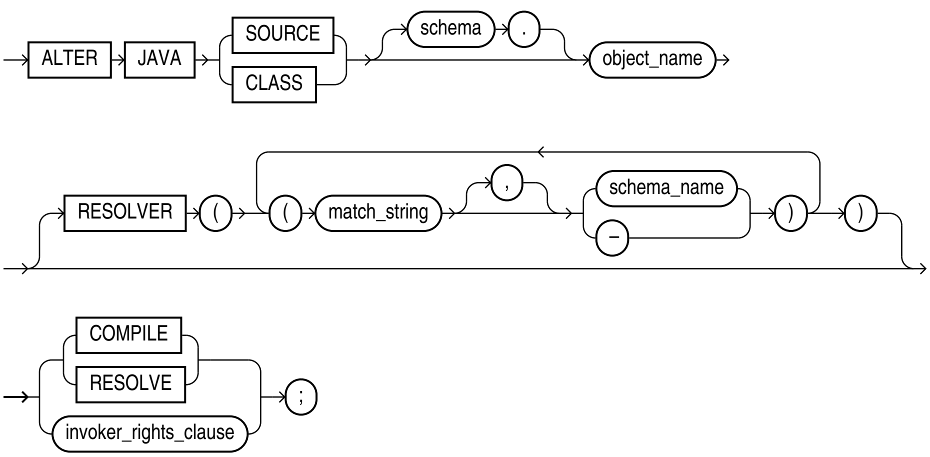
> **See Also:**
>
> *Oracle Database Java Developer's Guide* for more information on resolving Java classes and compiling Java sources

Prerequisites

The Java source or class must be in your own schema, or you must have the ALTER ANY PROCEDURE system privilege. You must also have the EXECUTE object privilege on Java classes.

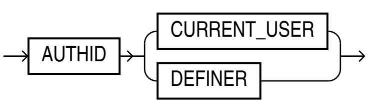
Syntax

***alter_java*::=**



(*invoker_rights_clause*::=)

***invoker_rights_clause*::=**



Semantics

JAVA SOURCE

Use `ALTER JAVA SOURCE` to compile a Java source schema object.

JAVA CLASS

Use `ALTER JAVA CLASS` to resolve a Java class schema object.

object_name

Specify a previously created Java class or source schema object. Use double quotation marks to preserve lower- or mixed-case names.

RESOLVER

The `RESOLVER` clause lets you specify how schemas are searched for referenced fully specified Java names, using the mapping pairs specified when the Java class or source was created.

> **See Also:**
>
> CREATE JAVA and "Resolving a Java Class: Example"

RESOLVE | COMPILE

`RESOLVE` and `COMPILE` are synonymous keywords. They let you specify that Oracle Database should attempt to resolve the primary Java class schema object.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

invoker_rights_clause

The *`invoker_rights_clause`* lets you specify whether the methods of the class execute with the privileges and in the schema of the user who defined it or with the privileges and in the schema of `CURRENT_USER`.

This clause also determines how Oracle Database resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER

Specify `CURRENT_USER` if you want the methods of the class to execute with the privileges of `CURRENT_USER`. This clause is the default and creates an **invoker-rights class**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of `CURRENT_USER`. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER

Specify `DEFINER` if you want the methods of the class to execute with the privileges of the user who defined the class.

This clause also specifies that external names resolve in the schema where the methods reside.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information on how `CURRENT_USER` is determined

Examples

Resolving a Java Class: Example

The following statement forces the resolution of a Java class:

```
ALTER JAVA CLASS "Agent"
   RESOLVER (("/usr/bin/bfile_dir/*" pm)(* public))
   RESOLVE;
```

11 SQL Statements: ALTER LIBRARY to ALTER SESSION

This chapter contains the following SQL statements:

- ALTER LIBRARY
- ALTER LOCKDOWN PROFILE
- ALTER MATERIALIZED VIEW
- ALTER MATERIALIZED VIEW LOG
- ALTER MATERIALIZED ZONEMAP
- ALTER OPERATOR
- ALTER OUTLINE
- ALTER PACKAGE
- ALTER PLUGGABLE DATABASE
- ALTER PROCEDURE
- ALTER PROFILE
- ALTER RESOURCE COST
- ALTER ROLE
- ALTER ROLLBACK SEGMENT
- ALTER SEQUENCE
- ALTER SESSION

ALTER LIBRARY

Purpose

The `ALTER LIBRARY` statement explicitly recompiles a library. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

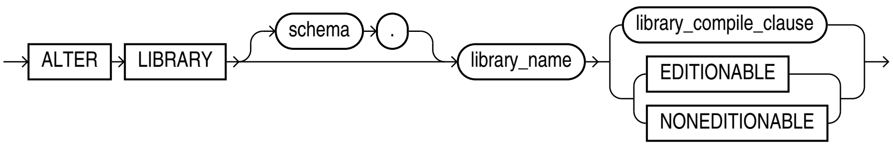
> **Note:**
>
> This statement does not change the declaration or definition of an existing library. To redeclare or redefine a library, use the "CREATE LIBRARY " with the `OR REPLACE` clause.

Prerequisites

If the library is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the library must be in your own schema or you must have the `ALTER ANY LIBRARY` system privilege.

Syntax

***alter_library*::=**



(*`library_compile_clause`*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

schema

Specify the schema containing the library. If you omit *`schema`*, then Oracle Database assumes the procedure is in your own schema.

library_name

Specify the name of the library to be recompiled.

library_compile_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling libraries.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the library becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `LIBRARY` in *`schema`*. The default is `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

ALTER LOCKDOWN PROFILE

Purpose

Use the `ALTER LOCKDOWN PROFILE` statement to alter a PDB lockdown profile. You can use PDB lockdown profiles in a multitenant environment to restrict user operations in pluggable databases (PDBs).

Immediately after you create a lockdown profile with the `CREATE LOCKDOWN PROFILE` statement, all user operations are enabled for the profile. You can then use the `ALTER LOCKDOWN PROFILE` statement to disable certain user operations for the profile. When a lockdown profile is applied to a CDB, application container, or PDB, users cannot perform the operations that are the disabled for the profile. If you later would like to reenable some of the disabled user operations, you can use the `ALTER LOCKDOWN PROFILE` statement to do so.

The `ALTER LOCKDOWN PROFILE` statement allows you to disable or enable:

- User operations associated with certain database features (using the *`lockdown_features`* clause)
- User operations associated with certain database options (using the *`lockdown_options`* clause)
- The issuance of certain SQL statements (using the *`lockdown_statements`* clause)

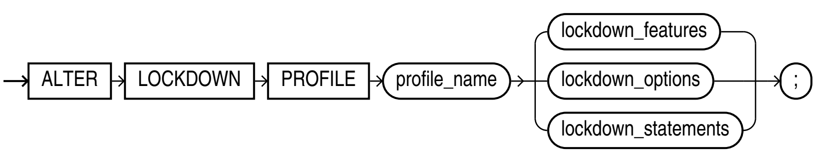
> **See Also:**
>
> - CREATE LOCKDOWN PROFILE and DROP LOCKDOWN PROFILE
> - *Oracle Database Security Guide* for more information on PDB lockdown profiles

Prerequisites

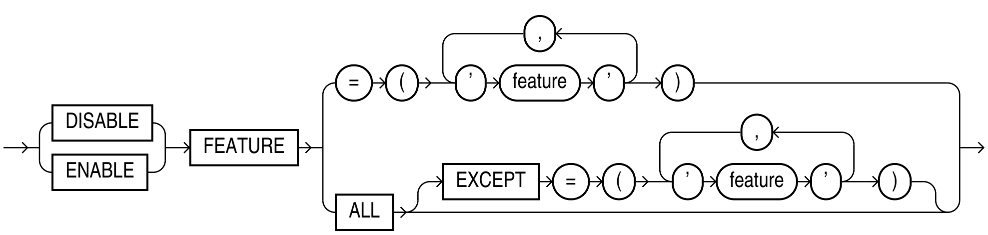
The `ALTER LOCKDOWN PROFILE` statement is valid only in a CDB. The current container must be the CDB root and you must have the `ALTER LOCKDOWN PROFILE` system privilege, either granted commonly or granted locally in the CDB root.

Syntax

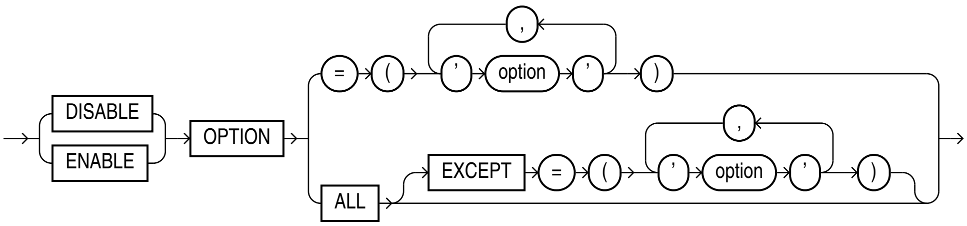
***alter_lockdown_profile*::=**



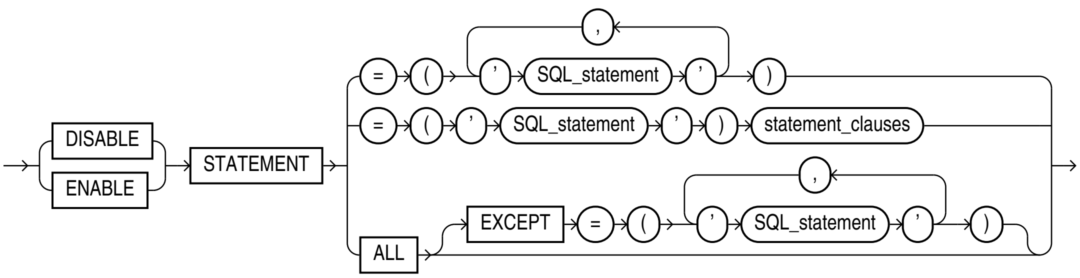
***lockdown_features*::=**



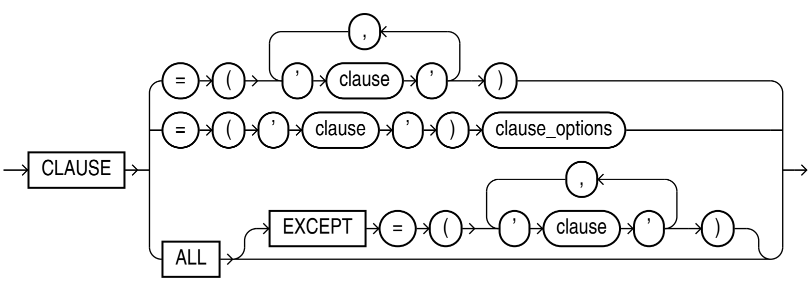
***lockdown_options*::=**



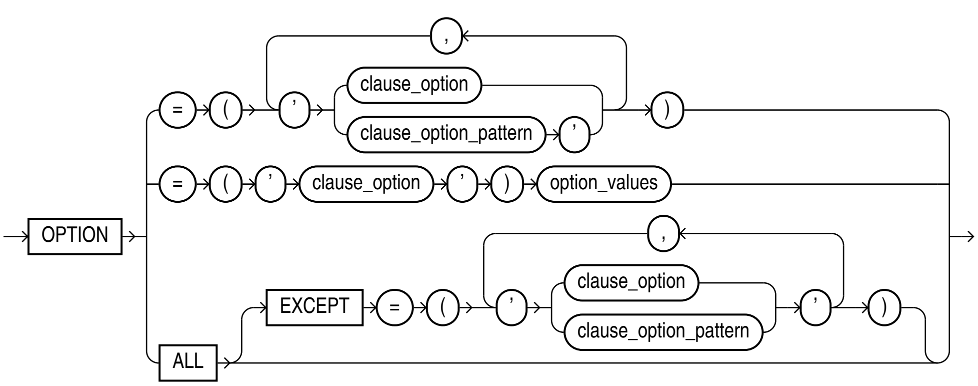
***lockdown_statements*::=**



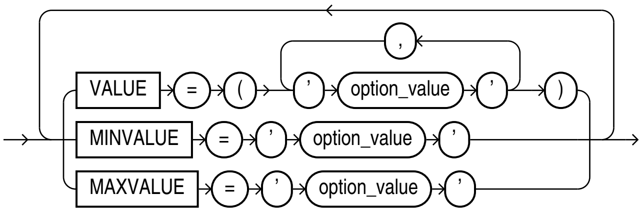
***statement_clauses*::=**



***clause_options*::=**



***option_values*::=**



Semantics

profile_name

Specify the name of the PDB lockdown profile to be altered.

You can find the names of existing PDB lockdown profiles by querying the `DBA_LOCKDOWN_PROFILES` data dictionary view.

lockdown_features

This clause lets you disable or enable user operations associated with certain database features.

- Specify `DISABLE` to disable user operations for the specified features. Users will be restricted from performing these operations in any PDB to which the profile applies.
- Specify `ENABLE` to enable user operations for the specified features. Users will be allowed to perform these operations in any PDB to which the profile applies.
- Use *`feature`* to specify the features whose operations you want to disable or enable. Table 11-1 lists the features you can specify and describes the operations associated with each feature. The table also indicates a feature bundle for each feature. For *`feature`*, you can specify a feature bundle name to disable or enable user operations for all features in that bundle, or you can specify an individual feature name. You can specify feature bundle names and feature names in any combination of uppercase and lowercase letters.
- Use `ALL` to specify all features listed in the table.
- Use `ALL EXCEPT` to specify all features listed in the table except the specified features.

If you omit this clause, then the default is `ENABLE ALL`.

Table 11-1 PDB Lockdown Profile Features

Feature Bundle	Feature	Operations
`AWR_ACCESS`	`AWR_ACCESS`	The PDB taking manual and automatic Automatic Workload Repository (AWR) snapshots
`COMMON_SCHEMA_ACCESS`	`COMMON_USER_LOCAL_SCHEMA_ACCESS`	A common user invoking an invoker's rights code unit or accessing a `BEQUEATH CURRENT_USER` view owned by any local user in the PDB

Table 11-1 (Cont.) PDB Lockdown Profile Features

Feature Bundle	Feature	Operations
COMMON_SCHEMA_ACCESS	LOCAL_USER_COMMON_SCHEMA_ACCESS	• A local user with an ANY system privilege (for example, CREATE ANY TABLE) creating or accessing objects in a common user's schema for which the privilege applies. **Note:** Disabling the LOCAL_USER_COMMON_SCHEMA_ACCESS feature does not prevent a local user with the SYSDBA privilege or specific object privileges from creating or accessing objects in a common user's schema. Therefore, Oracle recommends against granting such privileges to local users. • A local user with the BECOME USER system privilege becoming a common user • A local user altering a common user by issuing an ALTER USER statement • A local user using a common user for proxy connections
COMMON_SCHEMA_ACCESS	SECURITY_POLICIES	Creation of certain security policies by a local user on a common object, including: • Data Redaction • Fine Grained Auditing (FGA) • Real Application Security (RAS) • Virtual Private Database (VPD)
CONNECTIONS	COMMON_USER_CONNECT	A common user connecting to the PDB directly. If this feature is disabled, then in order to connect to the PDB, a common user must first connect to the CDB root and then switch to the desired PDB using the ALTER SESSION SET CONTAINER statement.
CONNECTIONS	LOCAL_SYSOPER_RESTRICTED_MODE_CONNECT	A local user with the SYSOPER privilege connecting to a PDB that is open in RESTRICTED mode
CTX_LOGGING	CTX_LOGGING	Use logging in Oracle Text PL/SQL procedures such as CTX_OUTPUT.START_LOG and CTX_OUTPUT.START_QUERY_LOG
JAVA	JAVA	Java as a whole. If this feature is disabled, then all options and features of the database that depend on Java will be disabled.
JAVA_RUNTIME	JAVA_RUNTIME	Operations through Java that require java.lang.RuntimePermission
NETWORK_ACCESS	AQ_PROTOCOLS	Using HTTP, SMTP, and OCI notification features through Oracle Streams Advanced Queuing (AQ)

Table 11-1 (Cont.) PDB Lockdown Profile Features

Feature Bundle	Feature	Operations
NETWORK_ACCESS	CTX_PROTOCOLS	• Operations that access the Oracle Text datastore types FILE_DATASTORE and URL_DATASTORE • Printing tokens as part of CTX logging with events EVENT_INDEX_PRINT_TOKEN and EVENT_OPT_PRINT_TOKEN
NETWORK_ACCESS	DBMS_DEBUG_JDWP	Using the DBMS_DEBUG_JDWP PL/SQL package
NETWORK_ACCESS	UTL_HTTP	Using the UTL_HTTP PL/SQL package
NETWORK_ACCESS	UTL_INADDR	Using the UTL_INADDR PL/SQL package
NETWORK_ACCESS	UTL_SMTP	Using the UTL_SMTP PL/SQL package
NETWORK_ACCESS	UTL_TCP	Using the UTL_TCP PL/SQL package
NETWORK_ACCESS	XDB_PROTOCOLS	Using HTTP, FTP, and other network protocols through XDB
OS_ACCESS	DROP_TABLESPACE_KEEP_DATAFILES	Dropping a tablespace in the PDB without specifying the INCLUDING CONTENTS AND DATAFILES clause in DROP TABLESPACE statement
OS_ACCESS	EXTERNAL_FILE_ACCESS	Using external files or directory objects in the PDB when PATH_PREFIX is not set for the PDB
OS_ACCESS	EXTERNAL_PROCEDURES	Using external procedure agent extproc in the PDB
OS_ACCESS	FILE_TRANSFER	Using the DBMS_FILE_TRANSFER package
OS_ACCESS	JAVA_OS_ACCESS	Using java.io.FilePermission from Java
OS_ACCESS	LOB_FILE_ACCESS	Using BFILE and CFILE data types
OS_ACCESS	TRACE_VIEW_ACCESS	Using the following trace views: • [G]V$DIAG_OPT_TRACE_RECORDS • [G]V$DIAG_SQL_TRACE_RECORDS • [G]V$DIAG_TRACE_FILE_CONTENTS • V$DIAG_SESS_OPT_TRACE_RECORDS • V$DIAG_SESS_SQL_TRACE_RECORDS
OS_ACCESS	UTL_FILE	Using UTL_FILE. If this feature is disabled, then the database blocks use of the UTL_FILE.FOPEN function.

lockdown_options

This clause lets you disable or enable user operations associate with certain database options.

- Specify `DISABLE` to disable user operations for the specified options. Users will be restricted from performing these operations in any PDB to which the profile applies.
- Specify `ENABLE` to enable user operations for the specified options. Users will be allowed to perform these operations in any PDB to which the profile applies.
- For *`option`*, you can specify the following database options in any combination of uppercase and lowercase letters:
  – `DATABASE QUEUING` – Represents user operations associated with the Oracle Database Advanced Queuing option
  – `PARTITIONING` – Represents user operations associated with the Oracle Partitioning option
- Use `ALL` to specify all options in the preceding list.
- Use `ALL EXCEPT` to specify all options in the preceding list except the specified options.

If you omit this clause, then the default is `ENABLE OPTION ALL`.

lockdown_statements

This clause lets you disable or enable the issuance of certain SQL statements.

- Specify `DISABLE` to disable the issuance of the specified SQL statements. Users will be restricted from issuing these statements in any PDB to which the profile applies.
- Specify `ENABLE` to enable the issuance of the specified SQL statements. Users will be allowed to issue these statements in any PDB to which the profile applies.
- For *`SQL_statement`*, you can specify the following statements in any combination of uppercase and lowercase letters:
  – `ALTER DATABASE`
  – `ALTER PLUGGABLE DATABASE`
  – `ALTER SESSION`
  – `ALTER SYSTEM`
- Use `ALL` to specify all statements in the preceding list.
- Use `ALL EXCEPT` to specify all statements in the preceding list except the specified statements.

If you omit this clause, then the default is `ENABLE STATEMENT ALL`.

statement_clauses

This clause lets you disable or enable specific clauses of the specified SQL statement.

- Use *`clause`* to specify the SQL keywords that form the clause you want to disable or enable. You can specify a clause in any combination of uppercase and lowercase letters.
- Use `ALL` to specify all clauses for the SQL statement.
- Use `ALL EXCEPT` to specify all clauses for the SQL statement except the specified clauses.

For *clause*, you must specify at least enough keywords to unambiguously identify a single clause for the SQL statement. The following are some examples of how to specify *clause* for the ALTER SYSTEM statement:

- To specify the *archive_log_clause*::=, specify ARCHIVE. This is sufficient because no other ALTER SYSTEM clause begins with the keyword ARCHIVE. Alternatively, you can specify ARCHIVE LOG for semantic clarity, but the LOG keyword is unnecessary.
- To specify either of the *rolling_migration_clauses*::=, you must specify START ROLLING MIGRATION or STOP ROLLING MIGRATION in order to distinguish these clauses from the similarly named *rolling_patch_clauses*::= START ROLLING PATCH and STOP ROLLING PATCH.
- You cannot specify the single keyword FLUSH, because several ALTER SYSTEM clauses begin with this keyword. You must instead specify each clause separately, such as FLUSH SHARED_POOL or FLUSH GLOBAL CONTEXT.

There is no need to specify optional keywords within a clause, because they have no effect. For example:

- The *archive_log_clause*::= has an optional INSTANCE keyword. However, you cannot enable or disable only ARCHIVE LOG clauses that contain the INSTANCE keyword. Specifying ARCHIVE LOG INSTANCE is equivalent to specifying ARCHIVE or ARCHIVE LOG.

There is no need to specify parameter values within a clause, because they have no effect. For example:

- The *shutdown_dispatcher_clause*::= requires you to specify a *dispatcher_name*. However, you cannot enable or disable SHUTDOWN clauses that contain a specific dispatcher name. Specifying SHUTDOWN dispatcher1 is equivalent to specifying SHUTDOWN.

See Also:

ALTER DATABASE , ALTER PLUGGABLE DATABASE, ALTER SESSION , and ALTER SYSTEM for complete information on the clauses for these statements

clause_options

This clause is valid only when you specify one of the following for *lockdown_statements* and *statement_clauses*:

```
{ DISABLE | ENABLE } STATEMENT = ('ALTER SESSION') CLAUSE = ('SET')
{ DISABLE | ENABLE } STATEMENT = ('ALTER SYSTEM') CLAUSE = ('SET')
```

This clause lets you disable or enable the setting or modification of specific options with the ALTER SESSION SET or ALTER SYSTEM SET statements.

- Use *clause_option* to specify the option you want to disable or enable.
- Use *clause_option_pattern* to specify a pattern that matches multiple options. Within the pattern, specify a percent sign (%) to match zero or more characters in an option name. For example, specifying 'QUERY_REWRITE_%' is equivalent to specifying both the QUERY_REWRITE_ENABLED and QUERY_REWRITE_INTEGRITY options.
- You can specify *clause_option* and *clause_option_pattern* in any combination of uppercase and lowercase letters.

- Use `ALL` to specify all options.
- Use `ALL EXCEPT` to specify all options except the specified options.

> **See Also:**
>
> The *alter_session_set_clause* clause of `ALTER SESSION` and the *alter_system_set_clause* clause of `ALTER SYSTEM` for complete information on the options you can specify for these statements

option_values

This clause is valid only when you specify one of the following for *`lockdown_statements`*, *`statement_clauses`*, and *`clause_options`*:

```
DISABLE STATEMENT = ('ALTER SESSION') CLAUSE = ('SET') OPTION =
clause_option
DISABLE STATEMENT = ('ALTER SYSTEM') CLAUSE = ('SET') OPTION =
clause_option
```

This clause lets you specify a default value for an option when disabling the setting of that option. For options that take numeric values, this clause also lets you restrict users from setting an option to certain values.

- The `VALUE` clause lets you specify a default *`option_value`* for *`clause_option`*, which will go into effect for any PDB to which the profile applies after you close and reopen the PDB. If *`clause_option`* accepts multiple default values, then you can specify more than one *`option_value`* in a comma-separated list. The purpose of using this clause is to simultaneously set a default value for an option and restrict users from setting or modifying the value.
- The `MINVALUE` clause lets you restricts users from setting the value of *`clause_option`* to a value less than *`option_value`*. You can specify this clause only for options that take a numeric value.
- The `MAXVALUE` clause lets you restricts users from setting the value of *`clause_option`* to a value greater than *`option_value`*. You can specify this clause only for options that take a numeric value.
- You can specify both the `MINVALUE` and `MAXVALUE` clauses together to restrict users from setting the value of *`clause_options`* to any value less than `MINVALUE` or greater than `MAXVALUE`.
- `MINVALUE` and `MAXVALUE` settings take effect immediately when the lockdown profile is assigned to a PDB; you need not close and reopen the PDB.

> **See Also:**
>
> *Oracle Database Reference* for complete information on the values allowed for the various options

Examples

The following statement creates PDB lockdown profile `hr_prof`:

```
CREATE LOCKDOWN PROFILE hr_prof;
```

The remaining examples in this section alter `hr_prof`.

Disabling Features for PDB Lockdown Profiles: Examples

The following statement disables all features in the feature bundle `NETWORK_ACCESS`:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE FEATURE = ('NETWORK_ACCESS');
```

The following statement disables the `LOB_FILE_ACCESS` and `TRACE_VIEW ACCESS` features:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE FEATURE = ('LOB_FILE_ACCESS', 'TRACE_VIEW_ACCESS');
```

The following statement disables all features except the `COMMON_USER_LOCAL_SCHEMA_ACCESS` and `LOCAL_USER_COMMON_SCHEMA_ACCESS` features:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE FEATURE ALL EXCEPT = ('COMMON_USER_LOCAL_SCHEMA_ACCESS',
'LOCAL_USER_COMMON_SCHEMA_ACCESS');
```

The following statement disables all features:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE FEATURE ALL;
```

Enabling Features for PDB Lockdown Profiles: Examples

The following statement enables the `UTL_HTTP` and `UTL_SMTP` features, as well as all features in the feature bundle `OS_ACCESS`:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE FEATURE = ('UTL_HTTP', 'UTL_SMTP', 'OS_ACCESS');
```

The following statement enables all features except the `AQ_PROTOCOLS` and `CTX_PROTOCOLS` features:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE FEATURE ALL EXCEPT = ('AQ_PROTOCOLS', 'CTX_PROTOCOLS');
```

The following statement enables all features:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE FEATURE ALL;
```

Disabling Options for PDB Lockdown Profiles: Examples

The following statement disables user operations associated with the Oracle Database Advanced Queuing option:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE OPTION = ('DATABASE QUEUING');
```

The following statement disables user operations associated with the Oracle Partitioning option:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE OPTION = ('PARTITIONING');
```

Enabling Options for PDB Lockdown Profiles: Examples

The following statement enables user operations associated with the Oracle Database Advanced Queuing option:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE OPTION = ('DATABASE QUEUING');
```

The following statement enables user operations associated both with the Oracle Database Advanced Queuing option and the Oracle Partitioning option:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE OPTION ALL;
```

Disabling SQL Statements for PBB Lockdown Profiles: Examples

The following statement disables the `ALTER DATABASE` statement:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER DATABASE');
```

The following statement disables the `ALTER SYSTEM SUSPEND` and `ALTER SYSTEM RESUME` statements:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER SYSTEM')
          CLAUSE = ('SUSPEND', 'RESUME');
```

The following statement disables all clauses of the ALTER PLUGGABLE DATABASE statement, except DEFAULT TABLESPACE and DEFAULT TEMPORARY TABLESPACE:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER PLUGGABLE DATABASE')
          CLAUSE ALL EXCEPT = ('DEFAULT TABLESPACE', 'DEFAULT TEMPORARY
TABLESPACE');
```

The following statement disables using the ALTER SESSION statement to set or modify COMMIT_WAIT or CURSOR_SHARING:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER SESSION')
          CLAUSE = ('SET')
          OPTION = ('COMMIT_WAIT', 'CURSOR_SHARING');
```

The following statement disables using the ALTER SYSTEM statement to set or modify the value of PDB_FILE_NAME_CONVERT. It also sets the default value for PDB_FILE_NAME_CONVERT to 'cdb1_pdb0', 'cdb1_pdb1'. This default value will take effect the next time the PDB is closed and reopened.

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER SYSTEM')
          CLAUSE = ('SET')
          OPTION = ('PDB_FILE_NAME_CONVERT')
          VALUE = ('cdb1_pdb0', 'cdb1_pdb1');
```

The following statement disables using the ALTER SYSTEM statement to set or modify the value of CPU_COUNT to a value less than 8:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER SYSTEM')
          CLAUSE = ('SET')
          OPTION = ('CPU_COUNT')
          MINVALUE = '8';
```

The following statement disables using the ALTER SYSTEM statement to set or modify the value of CPU_COUNT to a value greater than 2:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER SYSTEM')
          CLAUSE = ('SET')
          OPTION = ('CPU_COUNT')
          MAXVALUE = '2';
```

The following statement disables using the ALTER SYSTEM statement to set or modify the value of CPU_COUNT to a value less than 2 or greater than 6:

```
ALTER LOCKDOWN PROFILE hr_prof
  DISABLE STATEMENT = ('ALTER SYSTEM')
          CLAUSE = ('SET')
```

```
        OPTION = ('CPU_COUNT')
        MINVALUE = '2'
        MAXVALUE = '6';
```

Enabling SQL Statements for PBB Lockdown Profiles: Examples

The following statement enables all statements except `ALTER DATABASE`:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE STATEMENT ALL EXCEPT = ('ALTER DATABASE');
```

The following statement enables the `ALTER DATABASE MOUNT` and `ALTER DATABASE OPEN` statements:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE STATEMENT = ('ALTER DATABASE')
          CLAUSE = ('MOUNT', 'OPEN');
```

The following statement enables all clauses of the `ALTER PLUGGABLE DATABASE` statement, except `DEFAULT TABLESPACE` and `DEFAULT TEMPORARY TABLESPACE`:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE STATEMENT = ('ALTER PLUGGABLE DATABASE')
         CLAUSE ALL EXCEPT = ('DEFAULT TABLESPACE', 'DEFAULT TEMPORARY
TABLESPACE');
```

The following statement enables using the `ALTER SESSION` statement to set or modify `COMMIT_WAIT` or `CURSOR_SHARING`:

```
ALTER LOCKDOWN PROFILE hr_prof
  ENABLE STATEMENT = ('ALTER SESSION')
         CLAUSE = ('SET')
         OPTION = ('COMMIT_WAIT', 'CURSOR_SHARING');
```

ALTER MATERIALIZED VIEW

Purpose

A materialized view is a database object that contains the results of a query. The `FROM` clause of the query can name tables, views, and other materialized views. Collectively these source objects are called **master tables** (a replication term) or **detail tables** (a data warehousing term). This reference uses the term master tables for consistency. The databases containing the master tables are called the **master databases**.

Use the `ALTER MATERIALIZED VIEW` statement to modify an existing materialized view in one or more of the following ways:

- To change its storage characteristics
- To change its refresh method, mode, or time
- To alter its structure so that it is a different type of materialized view

- To enable or disable query rewrite

> **Note:**
>
> The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

> **See Also:**
>
> - CREATE MATERIALIZED VIEW for more information on creating materialized views
> - *Oracle Database Administrator's Guide* for information on materialized views in a replication environment
> - *Oracle Database Data Warehousing Guide* for information on materialized views in a data warehousing environment

Prerequisites

The materialized view must be in your own schema, or you must have the ALTER ANY MATERIALIZED VIEW system privilege.

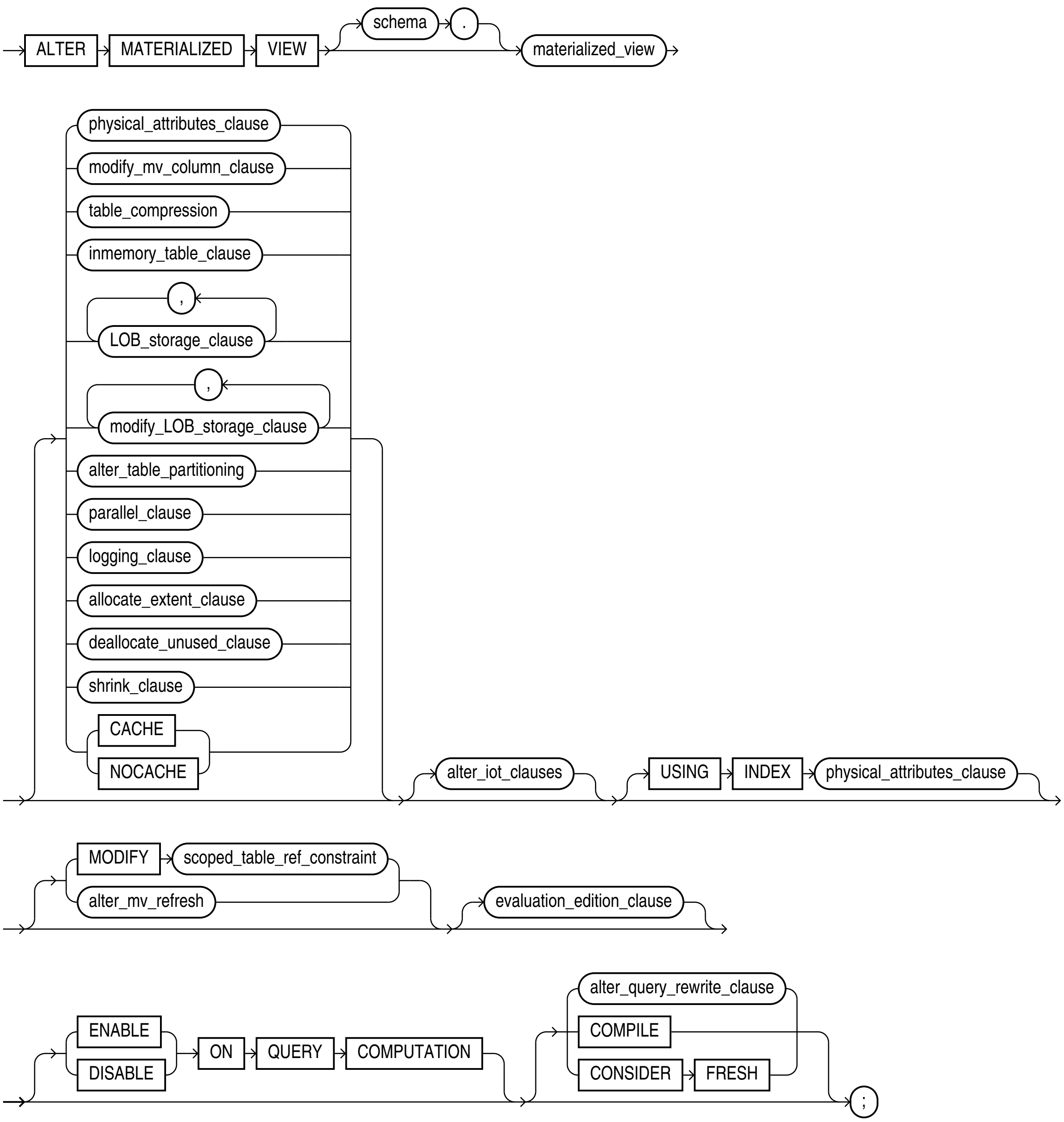
To enable a materialized view for query rewrite:

- If all of the master tables in the materialized view are in your schema, then you must have the QUERY REWRITE privilege.
- If any of the master tables are in another schema, then you must have the GLOBAL QUERY REWRITE privilege.
- If the materialized view is in another user's schema, then both you and the owner of that schema must have the appropriate QUERY REWRITE privilege, as described in the preceding two items. In addition, the owner of the materialized view must have SELECT access to any master tables that the materialized view owner does not own.

To specify an edition in the *evaluation_edition_clause* or the *unusable_editions_clause*, you must have the USE privilege on the edition.

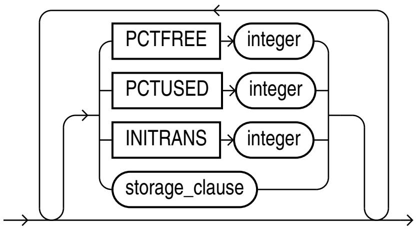
Syntax

***alter_materialized_view*::=**



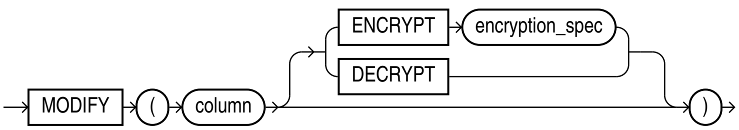
(*physical_attributes_clause*::=, *modify_mv_column_clause*::=, *table_compression*::=, *inmemory_table_clause*::=, *LOB_storage_clause*::= , *modify_LOB_storage_clause*::=, *alter_table_partitioning*::= (part of `ALTER TABLE`), *parallel_clause*::=, *logging_clause*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::=, *shrink_clause*::=, *alter_iot_clauses*::=, *scoped_table_ref_constraint*::=, *alter_mv_refresh*::=, *evaluation_edition_clause*::=, *alter_query_rewrite_clause*::=)

***physical_attributes_clause*::=**

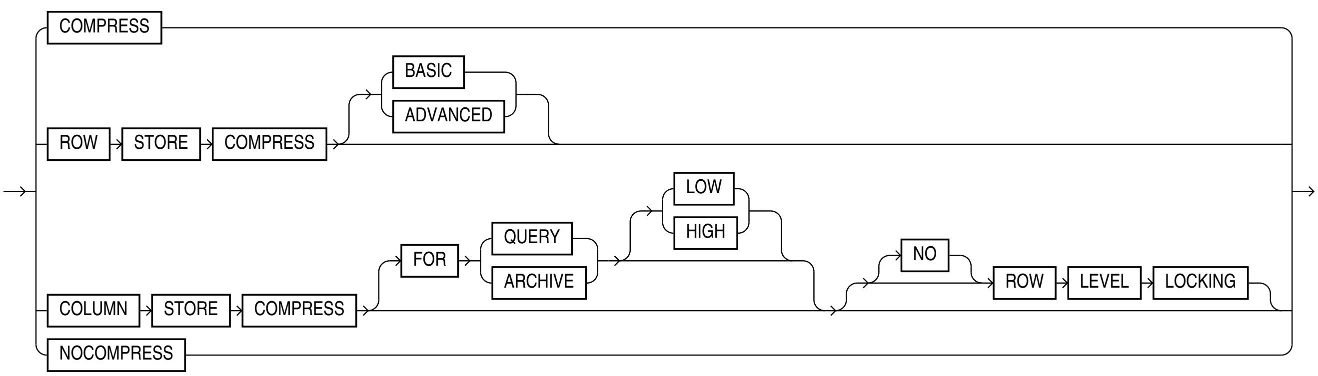


(*storage_clause::=*)

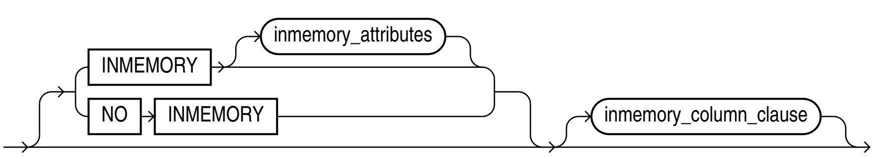
***modify_mv_column_clause*::=**



***table_compression*::=**

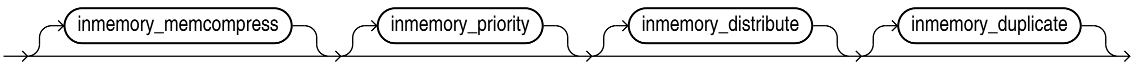


***inmemory_table_clause*::=**



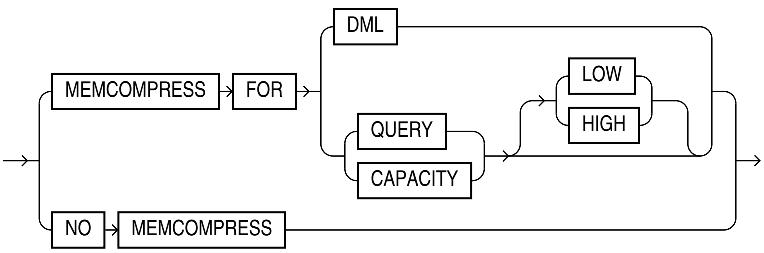
(*inmemory_attributes::=*, *inmemory_column_clause::=*)

***inmemory_attributes*::=**

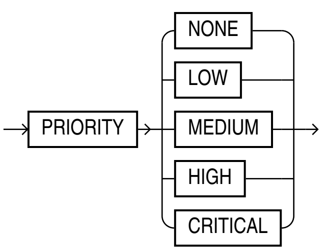


(*inmemory_memcompress*::=, *inmemory_priority*::=, *inmemory_distribute*::=, *inmemory_duplicate*::=)

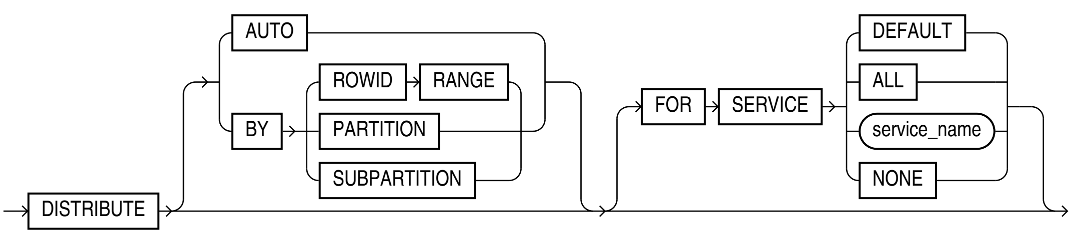
***inmemory_memcompress*::=**



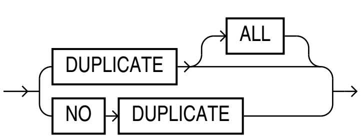
***inmemory_priority*::=**



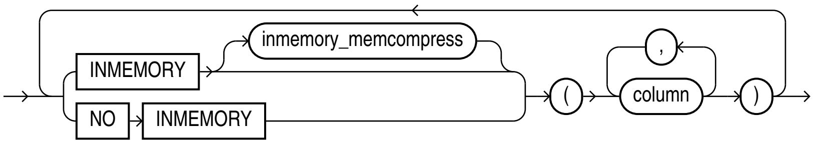
***inmemory_distribute*::=**



***inmemory_duplicate*::=**

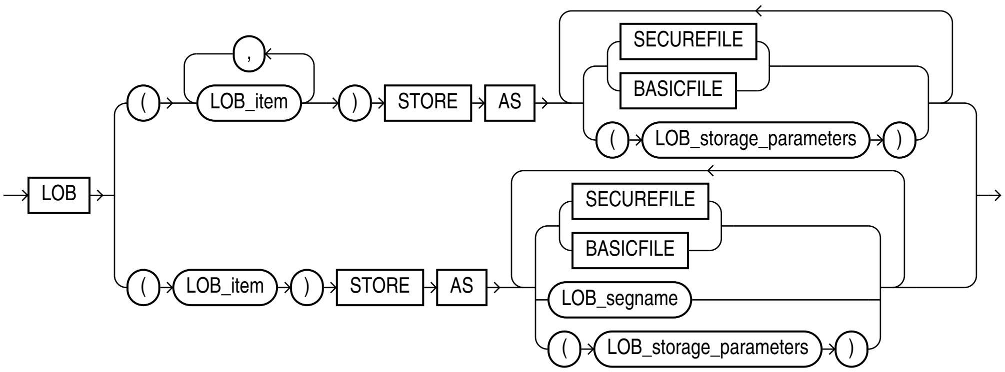


***inmemory_column_clause*::=**



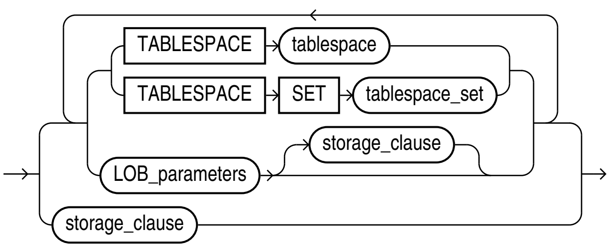
(*inmemory_memcompress::=*)

***LOB_storage_clause*::=**



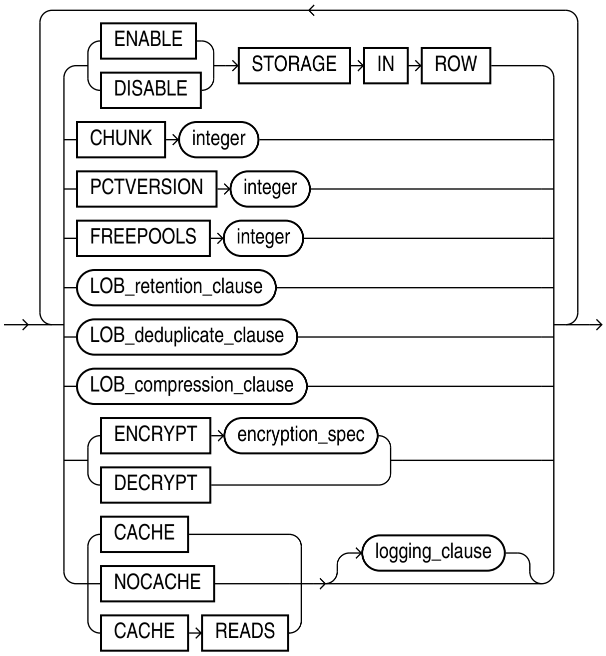
(*LOB_storage_parameters::=*)

***LOB_storage_parameters*::=**



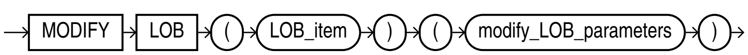
(`TABLESPACE SET`: not supported with `ALTER MATERIALIZED VIEW`, *LOB_parameters::=*, *storage_clause::=*)

***LOB_parameters*::=**



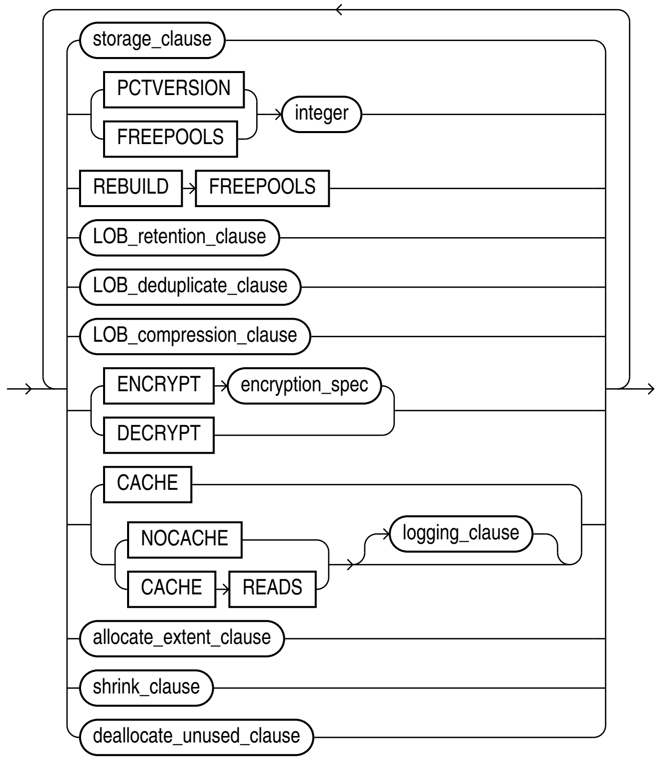
(*storage_clause*::=, *logging_clause*::=)

***modify_LOB_storage_clause*::=**



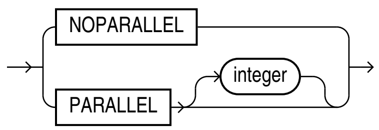
(*modify_LOB_parameters*::=)

***modify_LOB_parameters*::=**

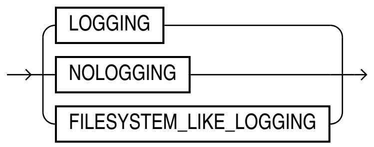


(*storage_clause*::=, *LOB_retention_clause*::=, *LOB_compression_clause*::=, *logging_clause*::=, *allocate_extent_clause*::=, *shrink_clause*::=, *deallocate_unused_clause*::=)

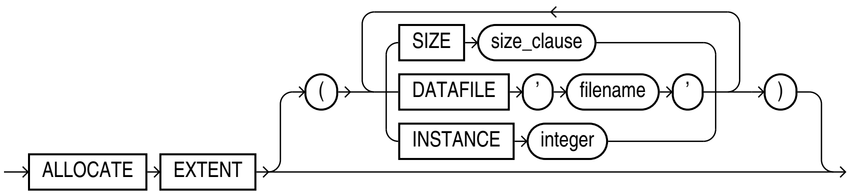
***parallel_clause*::=**



***logging_clause*::=**

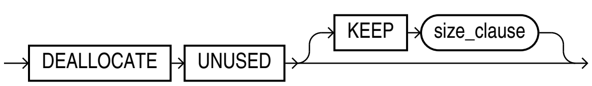


***allocate_extent_clause*::=**



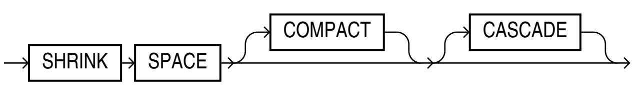
(*size_clause*::=)

***deallocate_unused_clause*::=**

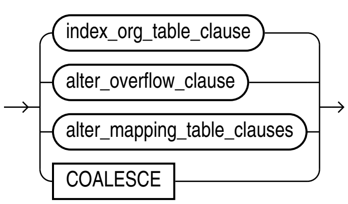


(*size_clause*::=)

***shrink_clause*::=**

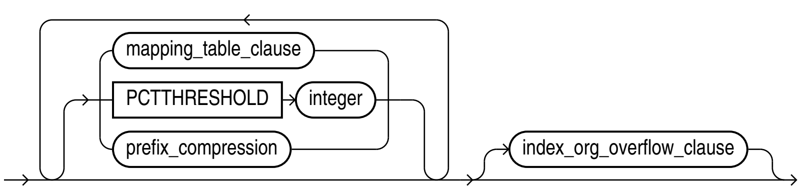


***alter_iot_clauses*::=**



(*index_org_table_clause*::=, *alter_overflow_clause*::=, `alter_mapping_table_clauses`: not supported with materialized views)

***index_org_table_clause*::=**

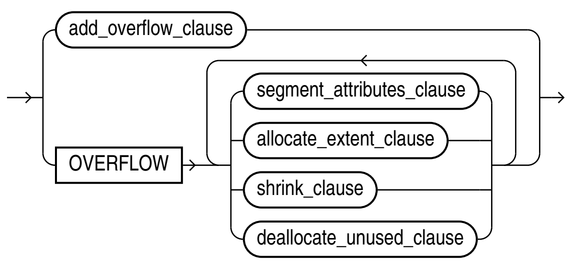


(`mapping_table_clause`: not supported with materialized views, `prefix_compression`: not supported for altering materialized views, *index_org_overflow_clause*::=)

***index_org_overflow_clause*::=**

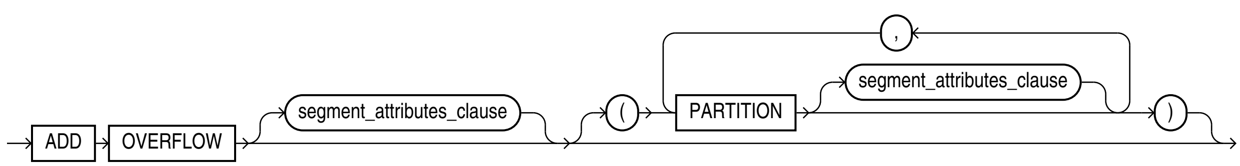
(*segment_attributes_clause*::=—part of ALTER TABLE)

***alter_overflow_clause*::=**



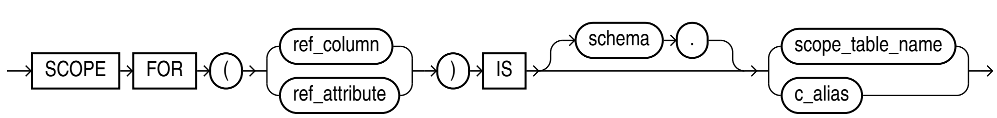
(*allocate_extent_clause*::=, *shrink_clause*::=, *deallocate_unused_clause*::=)

***add_overflow_clause*::=**

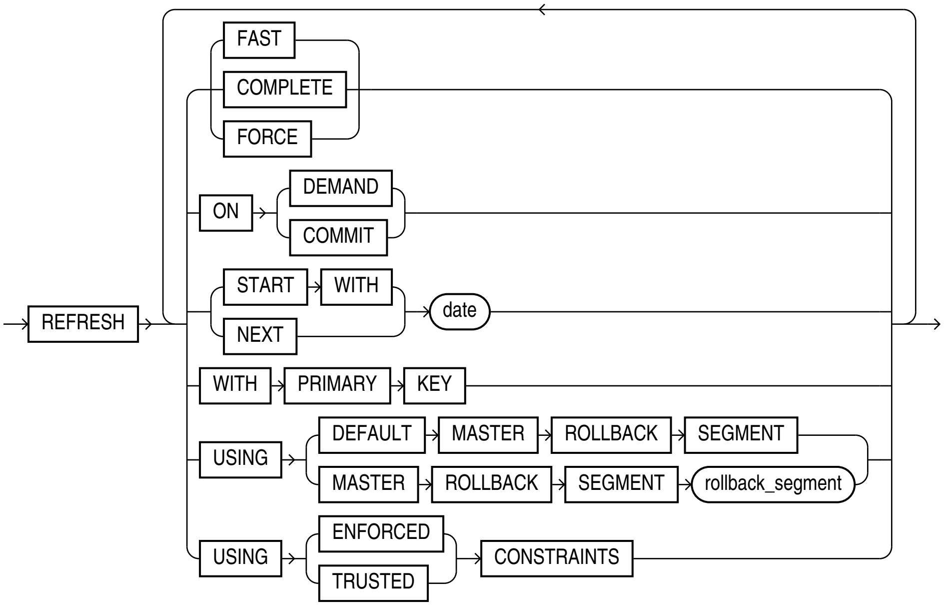


(*segment_attributes_clause*::=--part of ALTER TABLE)

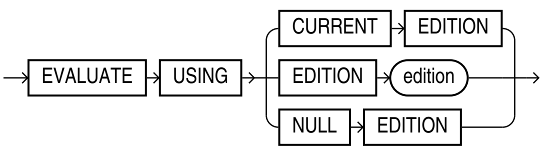
***scoped_table_ref_constraint*::=**



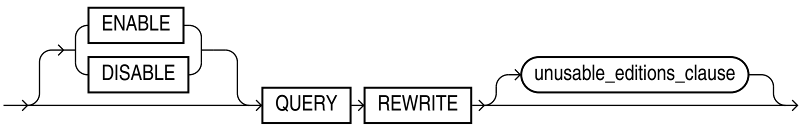
***alter_mv_refresh*::=**



evaluation_edition_clause::=

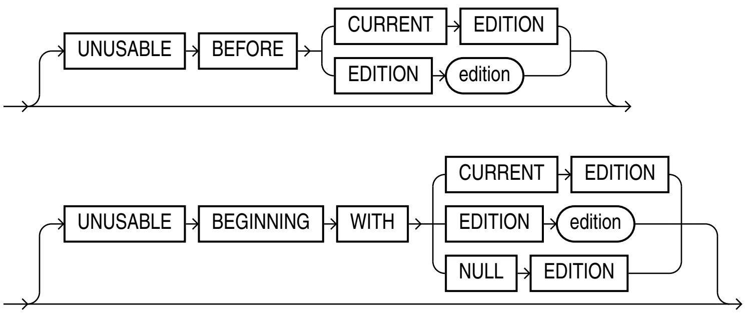


alter_query_rewrite_clause::=



> **Note:**
>
> You cannot specify only `QUERY REWRITE`. You must specify at least one of the following: `ENABLE`, `DISABLE`, or a subclause of the *`unusable_editions_clause`*.

***unusable_editions_clause*::=**



Semantics

schema

Specify the schema containing the materialized view. If you omit *schema*, then Oracle Database assumes the materialized view is in your own schema.

materialized_view

Specify the name of the materialized view to be altered.

physical_attributes_clause

Specify new values for the PCTFREE, PCTUSED, and INITRANS parameters (or, when used in the USING INDEX clause, for the INITRANS parameter only) and the storage characteristics for the materialized view. Refer to ALTER TABLE for information on the PCTFREE, PCTUSED, and INITRANS parameters and to storage_clause for information about storage characteristics.

modify_mv_column_clause

Use this clause to encrypt or decrypt this column of the materialized view. Refer to the CREATE TABLE clause *encryption_spec* for information on this clause.

table_compression

Use the *table_compression* clause to instruct Oracle Database whether to compress data segments to reduce disk and memory use. Refer to the *table_compression* clause of CREATE TABLE for the full semantics of this clause.

inmemory_table_clause

Use the *inmemory_table_clause* to enable or disable the materialized view or its columns for the In-Memory Column Store (IM column store), or to change the In-Memory attributes for the materialized view or its columns. This clause has the same semantics here as it has for the ALTER TABLE statement. Refer to the *inmemory_table_clause* of ALTER TABLE for the full semantics of this clause.

LOB_storage_clause

The *LOB_storage_clause* lets you specify the storage characteristics of a new LOB. LOB storage behaves for materialized views exactly as it does for tables. Refer to the *LOB_storage_clause* (in CREATE TABLE) for information on the LOB storage parameters.

modify_LOB_storage_clause

The *modify_LOB_storage_clause* lets you modify the physical attributes of the LOB attribute *LOB_item* or the LOB object attribute. Modification of LOB storage behaves for materialized views exactly as it does for tables.

> **See Also:**
>
> The *modify_LOB_storage_clause* of ALTER TABLE for information on the LOB storage parameters that can be modified

alter_table_partitioning

The syntax and general functioning of the partitioning clauses for materialized views is the same as for partitioned tables. Refer to *alter_table_partitioning* in the documentation on ALTER TABLE.

Restriction on Altering Materialized View Partitions

You cannot specify the *LOB_storage_clause* or *modify_LOB_storage_clause* within any of the *partitioning_clauses*.

> **Note:**
>
> If you want to keep the contents of the materialized view synchronized with those of the master table, then Oracle recommends that you manually perform a complete refresh of all materialized views dependent on the table after dropping or truncating a table partition.

MODIFY PARTITION UNUSABLE LOCAL INDEXES

Use this clause to mark UNUSABLE all the local index partitions associated with *partition*.

MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES

Use this clause to rebuild the unusable local index partitions associated with *partition*.

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for the materialized view.

For complete information on this clause, refer to *parallel_clause* in the documentation on `CREATE TABLE`.

logging_clause

Specify or change the logging characteristics of the materialized view. Refer to the *logging_clause* for a full description of this clause.

allocate_extent_clause

The *`allocate_extent_clause`* lets you explicitly allocate a new extent for the materialized view. Refer to the *allocate_extent_clause* for a full description of this clause.

deallocate_unused_clause

Use the *`deallocate_unused_clause`* to explicitly deallocate unused space at the end of the materialized view and make the freed space available for other segments. Refer to the *deallocate_unused_clause* for a full description of this clause.

shrink_clause

Use this clause to compact the materialized view segments. For complete information on this clause, refer to *shrink_clause* in the documentation on `CREATE TABLE`.

CACHE | NOCACHE

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. `NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list. Refer to "CACHE | NOCACHE | CACHE READS" in the documentation on `CREATE TABLE` for more information about this clause.

alter_iot_clauses

Use the *`alter_iot_clauses`* to change the characteristics of an index-organized materialized view. The keywords and parameters of the components of the *`alter_iot_clauses`* have the same semantics as in `ALTER TABLE`, with the restrictions that follow.

Restrictions on Altering Index-Organized Materialized Views

You cannot specify the *`mapping_table_clause`* or the *`prefix_compression`* clause of the *`index_org_table_clause`*.

> **See Also:**
>
> *index_org_table_clause* of `CREATE MATERIALIZED VIEW` for information on creating an index-organized materialized view

USING INDEX Clause

Use this clause to change the value of `INITRANS` and `STORAGE` parameters for the index Oracle Database uses to maintain the materialized view data.

Restriction on the USING INDEX clause

You cannot specify the `PCTUSED` or `PCTFREE` parameters in this clause.

MODIFY *scoped_table_ref_constraint*

Use the `MODIFY` *`scoped_table_ref_constraint`* clause to rescope a `REF` column or attribute to a new table or to an alias for a new column.

Restrictions on Rescoping REF Columns

You can rescope only one `REF` column or attribute in each `ALTER MATERIALIZED VIEW` statement, and this must be the only clause in this statement.

alter_mv_refresh

Use the *`alter_mv_refresh`* clause to change the default method and mode and the default times for automatic refreshes. If the contents of the master tables of a materialized view are modified, then the data in the materialized view must be updated to make the materialized view accurately reflect the data currently in its master table(s). This clause lets you schedule the times and specify the method and mode for Oracle Database to refresh the materialized view.

> **See Also:**
>
> - This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle Database Administrator's Guide* and *Oracle Database Data Warehousing Guide*.
> - *Oracle Database Data Warehousing Guide* to learn how to use refresh statistics to monitor the performance of materialized view refresh operations

FAST Clause

Specify `FAST` for the fast refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes are stored either in the materialized view log associated with the master table (for conventional DML changes) or in the direct loader log (for direct-path `INSERT` operations).

For both conventional DML changes and for direct-path `INSERT` operations, other conditions may restrict the eligibility of a materialized view for fast refresh.

When you change the refresh method to `FAST` in an `ALTER MATERIALIZED VIEW` statement, Oracle Database does not perform this verification. If the materialized view is not eligible for fast refresh, then Oracle Database returns an error when you attempt to refresh this view.

See Also:

- *Oracle Database Administrator’s Guide* for restrictions on fast refresh in replication environments
- *Oracle Database Data Warehousing Guide* for restrictions on fast refresh in data warehouse environments
- "Automatic Refresh: Examples"

COMPLETE Clause

Specify `COMPLETE` for the complete refresh method, which is implemented by executing the defining query of the materialized view. If you specify a complete refresh, then Oracle Database performs a complete refresh even if a fast refresh is possible.

See Also:

"Complete Refresh: Example"

FORCE Clause

Specify `FORCE` if, when a refresh occurs, you want Oracle Database to perform a fast refresh if one is possible or a complete refresh otherwise.

ON COMMIT Clause

Specify `ON COMMIT` if you want a refresh to occur whenever Oracle Database commits a transaction that operates on a master table of the materialized view.

You cannot specify both `ON COMMIT` and `ON DEMAND`. If you specify `ON COMMIT`, then you cannot also specify `START WITH` or `NEXT`.

Restriction on ON COMMIT

This clause is supported only for materialized join views and single-table materialized aggregate views.

ON DEMAND Clause

Specify `ON DEMAND` if you want the materialized view to be refreshed on demand by calling one of the three `DBMS_MVIEW` refresh procedures. If you omit both `ON COMMIT` and `ON DEMAND`, then `ON DEMAND` is the default.

You cannot specify both `ON COMMIT` and `ON DEMAND`. `START WITH` and `NEXT` take precedence over `ON DEMAND`. Therefore, in most circumstances it is not meaningful to specify `ON DEMAND` when you have specified `START WITH` or `NEXT`.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information on these procedures
- *Oracle Database Data Warehousing Guide* on the types of materialized views you can create by specifying `REFRESH ON DEMAND`

START WITH Clause

Specify `START WITH` *`date`* to indicate a date for the first automatic refresh time.

NEXT Clause

Specify `NEXT` to indicate a date expression for calculating the interval between automatic refreshes.

Both the `START WITH` and `NEXT` values must evaluate to a time in the future. If you omit the `START WITH` value, then Oracle Database determines the first automatic refresh time by evaluating the `NEXT` expression with respect to the creation time of the materialized view. If you specify a `START WITH` value but omit the `NEXT` value, then Oracle Database refreshes the materialized view only once. If you omit both the `START WITH` and `NEXT` values, or if you omit the *`alter_mv_refresh`* entirely, then Oracle Database does not automatically refresh the materialized view.

WITH PRIMARY KEY Clause

Specify `WITH PRIMARY KEY` to change a rowid materialized view to a primary key materialized view. Primary key materialized views allow materialized view master tables to be reorganized without affecting the ability of the materialized view to continue to fast refresh.

For you to specify this clause, the master table must contain an enabled primary key constraint and must have defined on it a materialized view log that logs primary key information.

See Also:

- *Oracle Database Administrator's Guide* for detailed information about primary key materialized views
- "Primary Key Materialized View: Example"

USING ROLLBACK SEGMENT Clause

This clause is not valid if your database is in automatic undo mode, because in that mode Oracle Database uses undo tablespaces instead of rollback segments. Oracle strongly recommends that you use automatic undo mode. This clause is supported for backward compatibility with replication environments containing older versions of Oracle Database that still use rollback segments.

For complete information on this clause, refer to `CREATE MATERIALIZED VIEW` ... "USING ROLLBACK SEGMENT Clause".

USING ... CONSTRAINTS Clause

This clause has the same semantics in `CREATE MATERIALIZED VIEW` and `ALTER MATERIALIZED VIEW` statements. For complete information, refer to "USING ... CONSTRAINTS Clause" in the documentation on `CREATE MATERIALIZED VIEW`.

evaluation_edition_clause

Use this clause to change the evaluation edition for the materialized view. This clause has the same semantics in `CREATE MATERIALIZED VIEW` and `ALTER MATERIALIZED VIEW` statements. For complete information on this clause, refer to *evaluation_edition_clause* in the documentation on `CREATE MATERIALIZED VIEW`.

Notes on Changing the Evaluation Edition of a Materialized View

The following notes apply when changing the evaluation edition of a materialized view:

- If you change the evaluation edition of a refresh-on-commit materialized view, then Oracle Database performs a complete refresh of the materialized view unless you specify `CONSIDER FRESH`.
- If you change the evaluation edition of a refresh-on-demand materialized view, then Oracle Database sets the staleness state of the materialized view to `STALE` unless you specify `CONSIDER FRESH`.
- For both refresh-on-commit and refresh-on-demand materialized views: If you change the evaluation edition and specify `CONSIDER FRESH`, then Oracle Database does not update the staleness state of the materialized view and does not rebuild the materialized view. Therefore, you can specify `CONSIDER FRESH` to indicate that, although the evaluation edition has changed, there is no difference in the results that *`subquery`* will produce. If the materialized view is stale and in need of either a fast refresh or a complete refresh before this statement is issued, then the state will not be changed and the materialized view may contain bad data.

{ ENABLE | DISABLE } ON QUERY COMPUTATION

This clause lets you control whether the materialized view is a real-time materialized view or a regular materialized view.

- Specify `ENABLE ON QUERY COMPUTATION` to convert a regular materialized view into a real-time materialized view by enabling on-query computation.
- Specify `DISABLE ON QUERY COMPUTATION` to convert a real-time materialized view into a regular materialized view by disabling on-query computation.

This clause has the same semantics in `CREATE MATERIALIZED VIEW` and `ALTER MATERIALIZED VIEW` statements. For complete information on this clause, refer to { ENABLE | DISABLE } ON QUERY COMPUTATION in the documentation on `CREATE MATERIALIZED VIEW`.

alter_query_rewrite_clause

Use this clause to specify whether the materialized view is eligible to be used for query rewrite.

ENABLE Clause

Specify `ENABLE` to enable the materialized view for query rewrite. If you currently specify, or previously specified, the *`unusable_editions_clause`* for the materialized view, then it is not enabled for query rewrite in the unusable editions.

See Also:

- "Enabling Query Rewrite: Example"
- *Oracle Database Data Warehousing Guide* to learn how to use refresh statistics to monitor the performance of materialized view refresh operations

Restrictions on Enabling Materialized Views

Enabling materialized views is subject to the following restrictions:

- If the materialized view is in an invalid or unusable state, then it is not eligible for query rewrite in spite of the ENABLE mode.
- You cannot enable query rewrite if the materialized view was created totally or in part from a view.
- You can enable query rewrite only if all user-defined functions in the materialized view are DETERMINISTIC.

  See Also:

  CREATE FUNCTION

- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include CURRENT_TIME or USER.

See Also:

Oracle Database Data Warehousing Guide for more information on query rewrite

DISABLE Clause

Specify DISABLE if you do not want the materialized view to be eligible for use by query rewrite. If a materialized view is in the invalid state, then it is not eligible for use by query rewrite, whether or not it is disabled. However, a disabled materialized view can be refreshed.

unusable_editions_clause

Use this clause to specify the editions in which the materialized view is not eligible for query rewrite. This clause has the same semantics in CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW statements. For complete information on this clause, refer to *unusable_editions_clause* in the documentation on CREATE MATERIALIZED VIEW.

Cursors that use the materialized view for query rewrite and were compiled in an edition that is made unusable will be invalidated.

COMPILE

Specify `COMPILE` to explicitly revalidate a materialized view. If an object upon which the materialized view depends is dropped or altered, then the materialized view remains accessible, but it is invalid for query rewrite. You can use this clause to explicitly revalidate the materialized view to make it eligible for query rewrite.

If the materialized view fails to revalidate, then it cannot be refreshed or used for query rewrite.

> **See Also:**
>
> "Compiling a Materialized View: Example"

CONSIDER FRESH

This clause lets you manage the staleness state of a materialized view after changes have been made to its master tables. `CONSIDER FRESH` directs Oracle Database to consider the materialized view fresh and therefore eligible for query rewrite in the `TRUSTED` or `STALE_TOLERATED` modes.

> **Caution:**
>
> The `CONSIDER FRESH` clause also directs Oracle Database to no longer apply any rows in a materialized view log or Partition Change Tracking changes to the materialized view prior to the issuance of the `CONSIDER FRESH` clause. In other words, the pending changes will be ignored and deleted, not applied to the materialized view. This may result in the materialized view containing more or less data than the base table.

Because Oracle Database cannot guarantee the freshness of the materialized view, query rewrite in `ENFORCED` mode is not supported. This clause also sets the staleness state of the materialized view to `UNKNOWN`. The staleness state is displayed in the `STALENESS` column of the `ALL_MVIEWS`, `DBA_MVIEWS`, and `USER_MVIEWS` data dictionary views.

A materialized view is stale if changes have been made to the contents of any of its master tables. This clause directs Oracle Database to assume that the materialized view is fresh and that no such changes have been made. Therefore, actual updates to those tables pending refresh are purged with respect to the materialized view.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for more information on query rewrite and the implications of performing partition maintenance operations on master tables
> - "CONSIDER FRESH: Example"

Examples

Automatic Refresh: Examples

The following statement changes the default refresh method for the sales_by_month_by_state materialized view (created in "Creating Materialized Aggregate Views: Example") to FAST:

```
ALTER MATERIALIZED VIEW sales_by_month_by_state
   REFRESH FAST;
```

The next automatic refresh of the materialized view will be a fast refresh provided it is a simple materialized view and its master table has a materialized view log that was created before the materialized view was created or last refreshed.

Because the REFRESH clause does not specify START WITH or NEXT values, Oracle Database will use the refresh intervals established by the REFRESH clause when the sales_by_month_by_state materialized view was created or last altered.

The following statement establishes a new interval between automatic refreshes for the sales_by_month_by_state materialized view:

```
ALTER MATERIALIZED VIEW sales_by_month_by_state
   REFRESH NEXT SYSDATE+7;
```

Because the REFRESH clause does not specify a START WITH value, the next automatic refresh occurs at the time established by the START WITH and NEXT values specified when the sales_by_month_by_state materialized view was created or last altered.

At the time of the next automatic refresh, Oracle Database refreshes the materialized view, evaluates the NEXT expression SYSDATE+7 to determine the next automatic refresh time, and continues to refresh the materialized view automatically once a week. Because the REFRESH clause does not explicitly specify a refresh method, Oracle Database continues to use the refresh method specified by the REFRESH clause of the CREATE MATERIALIZED VIEW or most recent ALTER MATERIALIZED VIEW statement.

CONSIDER FRESH: Example

The following statement instructs Oracle Database that materialized view sales_by_month_by_state should be considered fresh. This statement allows sales_by_month_by_state to be eligible for query rewrite in TRUSTED mode even after you have performed partition maintenance operations on the master tables of sales_by_month_by_state:

```
ALTER MATERIALIZED VIEW sales_by_month_by_state CONSIDER FRESH;
```

As a result of the preceding statement, any partition maintenance operations that were done to the base table since the last refresh of the materialized view will not be applied to the materialized view. For example, the add, drop, or change of data in a partition in the base table will not be reflected in the materialized view if CONSIDER FRESH is used before the next refresh of the materialized view. Refer to CONSIDER FRESH for more information.

See Also:

"Splitting Table Partitions: Examples" for a partitioning maintenance example that would require this ALTER MATERIALIZED VIEW example

Complete Refresh: Example

The following statement specifies a new refresh method, a new NEXT refresh time, and a new interval between automatic refreshes of the emp_data materialized view (created in "Periodic Refresh of Materialized Views: Example"):

```
ALTER MATERIALIZED VIEW emp_data
   REFRESH COMPLETE
   START WITH TRUNC(SYSDATE+1) + 9/24
   NEXT SYSDATE+7;
```

The START WITH value establishes the next automatic refresh for the materialized view to be 9:00 a.m. tomorrow. At that point, Oracle Database performs a complete refresh of the materialized view, evaluates the NEXT expression, and subsequently refreshes the materialized view every week.

Enabling Query Rewrite: Example

The following statement enables query rewrite on the materialized view emp_data and implicitly revalidates it:

```
ALTER MATERIALIZED VIEW emp_data
   ENABLE QUERY REWRITE;
```

Primary Key Materialized View: Example

The following statement changes the rowid materialized view order_data (created in "Creating Rowid Materialized Views: Example") to a primary key materialized view. This example requires that you have already defined a materialized view log with a primary key on order_data.

```
ALTER MATERIALIZED VIEW order_data
   REFRESH WITH PRIMARY KEY;
```

Compiling a Materialized View: Example

The following statement revalidates the materialized view store_mv:

```
ALTER MATERIALIZED VIEW order_data COMPILE;
```

ALTER MATERIALIZED VIEW LOG

Purpose

A **materialized view log** is a table associated with the master table of a materialized view. Use the ALTER MATERIALIZED VIEW LOG statement to alter the storage characteristics or type of an existing materialized view log.

Note:

The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

See Also:

- CREATE MATERIALIZED VIEW LOG for information on creating a materialized view log
- ALTER MATERIALIZED VIEW for more information on materialized views, including refreshing them
- CREATE MATERIALIZED VIEW for a description of the various types of materialized views

Prerequisites

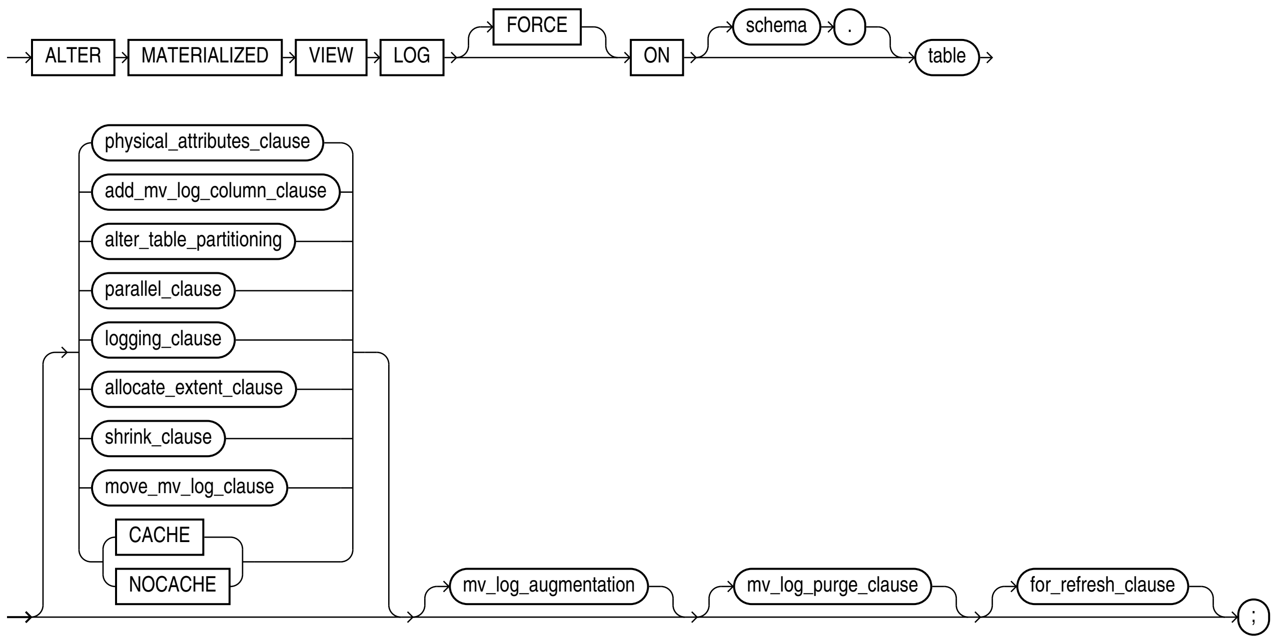
You must be the owner of the master table, or you must have the `READ` or `SELECT` privilege on the master table and the `ALTER` privilege on the materialized view log.

See Also:

Oracle Database Administrator's Guide for detailed information about the prerequisites for `ALTER MATERIALIZED VIEW LOG`

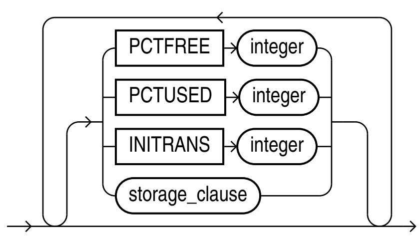
Syntax

***alter_materialized_view_log*::=**



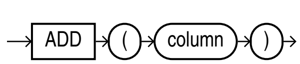
(*physical_attributes_clause*::=, *add_mv_log_column_clause*::=, *alter_table_partitioning*::= (in `ALTER TABLE`), *parallel_clause*::=, *logging_clause*::=, *allocate_extent_clause*::=, *shrink_clause*::=, *move_mv_log_clause*::=, *mv_log_augmentation*::=, *mv_log_purge_clause*::=, *for_refresh_clause*::=)

***physical_attributes_clause*::=**

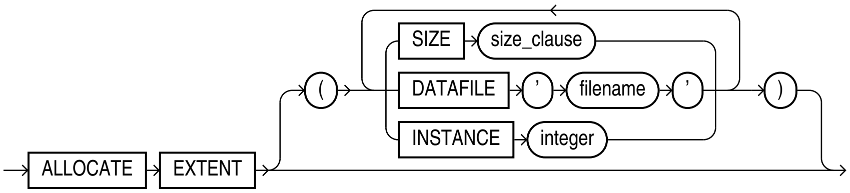


storage_clause::=

***add_mv_log_column_clause*::=**

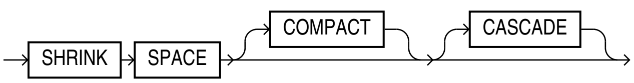


***allocate_extent_clause*::=**

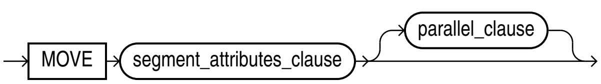


(*size_clause::=*)

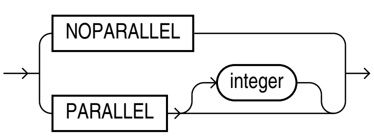
***shrink_clause*::=**



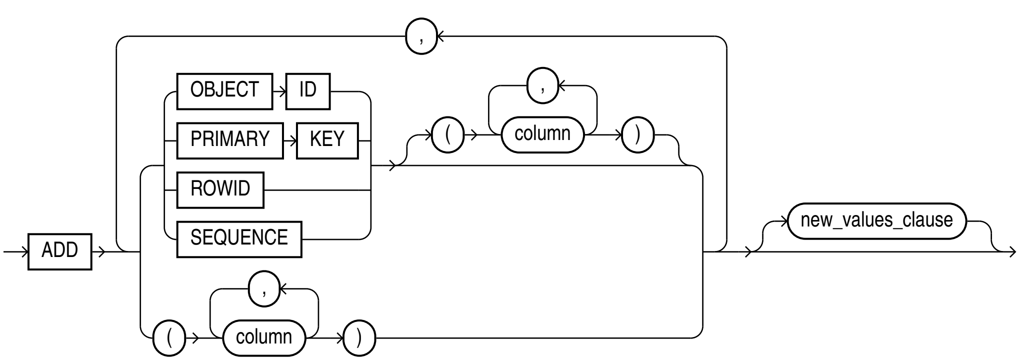
***move_mv_log_clause*::=**



***parallel_clause*::=**

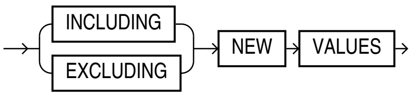


***mv_log_augmentation*::=**

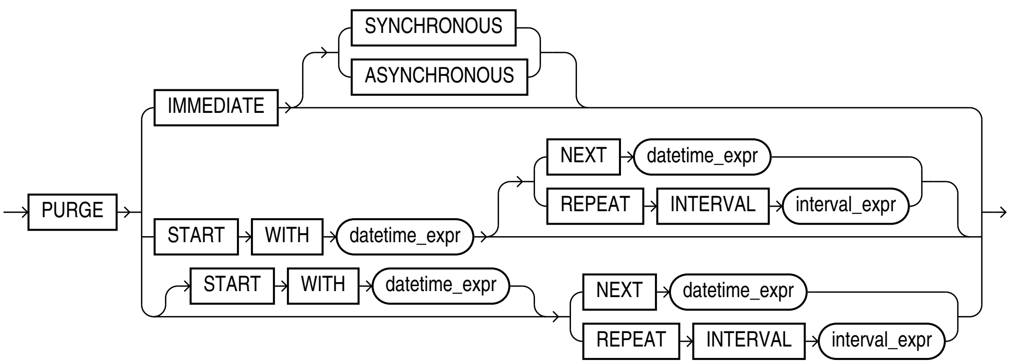


(*new_values_clause*::=

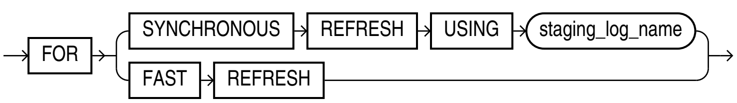
***new_values_clause*::=**



***mv_log_purge_clause*::=**



***for_refresh_clause*::=**



Semantics

FORCE

If you specify `FORCE` and any items specified with the `ADD` clause have already been specified for the materialized view log, then Oracle Database does not return an error, but silently ignores the existing elements and adds to the materialized view log any items that do not already exist in the log. Likewise, if you specify `INCLUDING NEW VALUES` and that attribute has already been specified for the materialized view log, Oracle Database ignores the redundancy and does not return an error.

schema

Specify the schema containing the master table. If you omit *`schema`*, then Oracle Database assumes the materialized view log is in your own schema.

table

Specify the name of the master table associated with the materialized view log to be altered.

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of the PCTFREE, PCTUSED, and INITRANS parameters and the storage characteristics for the materialized view log, the partition, the overflow data segment, or the default characteristics of a partitioned materialized view log.

Restriction on Materialized View Log Physical Attributes

You cannot use the *storage_clause* to modify extent parameters if the materialized view log resides in a locally managed tablespace. Refer to CREATE TABLE for a description of these parameters.

add_mv_log_column_clause

When you add a column to the master table of the materialized view log, the database does not automatically add a column to the materialized view log. Therefore, use this clause to add a column to the materialized view log. Oracle Database will encrypt the newly added column if the corresponding column of the master table is encrypted.

alter_table_partitioning

The syntax and general functioning of the partitioning clauses is the same as described for the ALTER TABLE statement. Refer to *alter_table_partitioning* in the documentation on ALTER TABLE.

Restrictions on Altering Materialized View Log Partitions

Altering materialized view log partitions is subject to the following restrictions:

- You cannot use the *LOB_storage_clause* or *modify_LOB_storage_clause* when modifying partitions of a materialized view log.
- If you attempt to drop, truncate, or exchange a materialized view log partition, then Oracle Database raises an error.

parallel_clause

The *parallel_clause* lets you specify whether parallel operations will be supported for the materialized view log.

For complete information on this clause, refer to *parallel_clause* in the documentation on CREATE TABLE.

logging_clause

Specify the logging attribute of the materialized view log. Refer to the *logging_clause* for a full description of this clause.

allocate_extent_clause

Use the *allocate_extent_clause* to explicitly allocate a new extent for the materialized view log. Refer to *allocate_extent_clause* for a full description of this clause.

shrink_clause

Use this clause to compact the materialized view log segments. For complete information on this clause, refer to *shrink_clause* in the documentation on `CREATE TABLE`.

move_mv_log_clause

Use the `MOVE` clause to move the materialized view log table to a different tablespace, to change other segment or storage attributes of the materialized view log, or to change the parallelism of the materialized view log.

Restriction on Moving Materialized View Logs

The `ENCRYPT` clause of the *`storage_clause`* of *`segment_attributes`* is not valid for materialized view logs.

CACHE | NOCACHE Clause

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this log are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. `NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list. Refer to "CACHE | NOCACHE | CACHE READS" in the documentation on `CREATE TABLE` for more information about this clause.

mv_log_augmentation

Use the `ADD` clause to augment the materialized view log so that it records the primary key values, rowid values, object ID values, or a sequence when rows in the materialized view master table are changed. This clause can also be used to record additional columns.

To stop recording any of this information, you must first drop the materialized view log and then re-create it. Dropping the materialized view log and then re-creating it forces a complete refresh for each of the existing materialized views that depend on the master table on its next refresh.

Restriction on Augmenting Materialized View Logs

You can specify only one `PRIMARY KEY`, one `ROWID`, one `OBJECT ID`, one `SEQUENCE`, and each column in the column list once for each materialized view log. You can specify only a single occurrence of `PRIMARY KEY`, `ROWID`, `OBJECT ID`, `SEQUENCE`, and column list within this `ALTER` statement. Also, if any of these values was specified at create time (either implicitly or explicitly), you cannot specify that value in this `ALTER` statement unless you use the `FORCE` option.

OBJECT ID

Specify `OBJECT ID` if you want the appropriate object identifier of all rows that are changed to be recorded in the materialized view log.

Restriction on the OBJECT ID clause

You can specify `OBJECT ID` only for logs on object tables, and you cannot specify it for storage tables.

PRIMARY KEY

Specify PRIMARY KEY if you want the primary key values of all rows that are changed to be recorded in the materialized view log.

ROWID

Specify ROWID if you want the rowid values of all rows that are changed to be recorded in the materialized view log.

SEQUENCE

Specify SEQUENCE to indicate that a sequence value providing additional ordering information should be recorded in the materialized view log.

column

Specify the additional columns whose values you want to be recorded in the materialized view log for all rows that are changed. Typically these columns are filter columns (non-primary-key columns referenced by subquery materialized views) and join columns (non-primary-key columns that define a join in the WHERE clause of the subquery).

> **See Also:**
>
> - CREATE MATERIALIZED VIEW for details on explicit and implicit inclusion of materialized view log values
> - *Oracle Database Administrator's Guide* for more information about filter columns and join columns
> - "Rowid Materialized View Log: Example"

NEW VALUES Clause

The NEW VALUES clause lets you specify whether Oracle Database saves both old and new values for update DML operations in the materialized view log. The value you set in this clause applies to all columns in the log, not only to columns you may have added in this ALTER MATERIALIZED VIEW LOG statement.

INCLUDING

Specify INCLUDING to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, then you must specify INCLUDING.

EXCLUDING

Specify EXCLUDING to disable the recording of new values in the log. You can use this clause to avoid the overhead of recording new values.

If you have a fast-refreshable single-table materialized aggregate view defined on this table, then do not specify EXCLUDING NEW VALUES unless you first change the refresh mode of the materialized view to something other than FAST.

> **See Also:**
>
> "Materialized View Log EXCLUDING NEW VALUES: Example"

mv_log_purge_clause

Use this clause alter the purge attributes of the materialized view log in the following ways:

- Change the purge from `IMMEDIATE SYNCHRONOUS` to `IMMEDIATE ASYNCHRONOUS` or from `IMMEDIATE ASYNCHRONOUS` to `IMMEDIATE SYNCHRONOUS`
- Change the purge from `IMMEDIATE` to scheduled or from scheduled to `IMMEDIATE`
- Specify a new start time and a new `next` time and `interval`

If you are altering purge from scheduled to `IMMEDIATE`, then the scheduled purged job associated with that materialized view log is dropped. If you are altering purge from `IMMEDIATE` to scheduled, then a purge job is created with the attributes provided. If you are altering scheduled purge attributes, then only those attributes specified will be changed in the scheduler purge job.

You must specify `FORCE` if you are altering log purge to its current state (that is, you are not making any change), unless you are changing scheduled purge attributes.

To learn whether the purge time or interval has already been set for this materialized view log, query the `*_MVIEW_LOGS` data dictionary views. See the `CREATE MATERIALIZED VIEW LOG` clause *mv_log_purge_clause* for the full semantics of this clause.

for_refresh_clause

Use this clause to change the refresh method for which the materialized view log will be used.

FOR SYNCHRONOUS REFRESH

Specify this clause to change from fast refresh to synchronous refresh, or complete refresh to synchronous refresh. A staging log will be created.

If you are changing from fast refresh, then ensure that the following conditions are satisfied before using this clause:

- All changes in the materialized view log have been consumed.
- Any refresh-on-demand materialized views associated with the master table have been refreshed.
- Any refresh-on-commit materialized views associated with the master table have been converted to refresh-on-demand materialized views.

After you use this clause, you cannot perform DML operations directly on the master table. You must use the procedures in the `DBMS_SYNC_REFRESH` package to prepare and execute change data operations.

FOR FAST REFRESH

Specify this clause to change from synchronous refresh to fast refresh, or complete refresh to fast refresh. A materialized view log will be created.

If you are changing from synchronous refresh to fast refresh, then ensure that all changes in the staging log have been consumed before using this clause.

After you use this clause, you can perform DML operations directly on the master table.

See the CREATE MATERIALIZED VIEW LOG clause *for_refresh_clause* for the full semantics of this clause.

Examples

Rowid Materialized View Log: Example

The following statement alters an existing primary key materialized view log to also record rowid information:

```
ALTER MATERIALIZED VIEW LOG ON order_items ADD ROWID;
```

Materialized View Log EXCLUDING NEW VALUES: Example

The following statement alters the materialized view log on hr.employees by adding a filter column and excluding new values. Any materialized aggregate views that use this log will no longer be fast refreshable. However, if fast refresh is no longer needed, this action avoids the overhead of recording new values:

```
ALTER MATERIALIZED VIEW LOG ON employees
   ADD (commission_pct)
   EXCLUDING NEW VALUES;
```

ALTER MATERIALIZED ZONEMAP

Purpose

Use the ALTER MATERIALIZED ZONEMAP statement to modify an existing zone map in one of the following ways:

- To change its attributes
- To change its default refresh method and mode
- To enable or disable its use for pruning
- To compile it, rebuild it, or make it unusable

See Also:

- CREATE MATERIALIZED ZONEMAP for information on creating zone maps
- *Oracle Database Data Warehousing Guide* for more information on zone maps

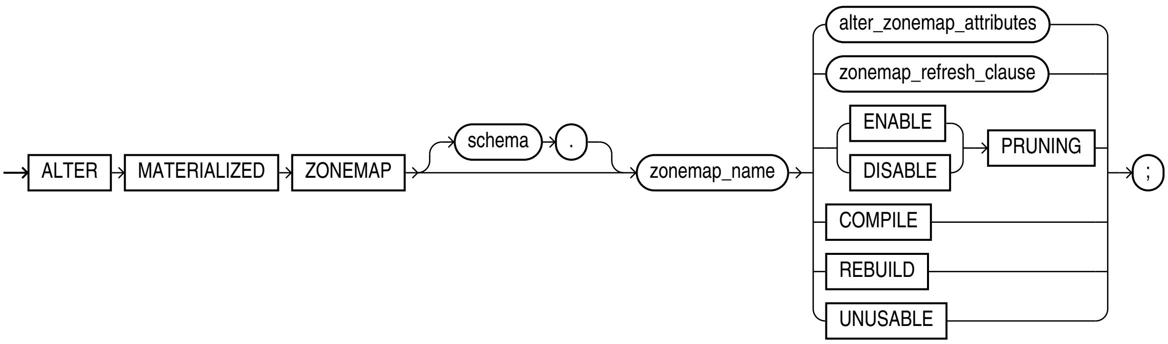
Prerequisites

The zone map must be in your own schema or you must have the ALTER ANY MATERIALIZED VIEW system privilege.

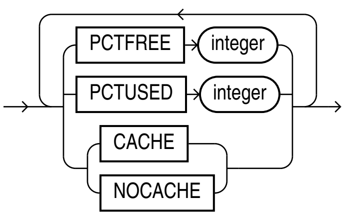
The user who owns the schema containing the zone map must have access to any base tables of the zone map that reside outside of that schema, either through a `READ` or `SELECT` object privilege on each of the tables, or through the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

Syntax

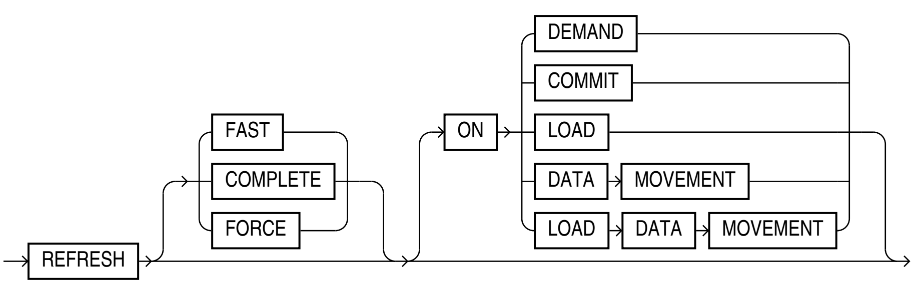
***alter_materialized_zonemap*::=**



***alter_zonemap_attributes*::=**



***zonemap_refresh_clause*::=**



Note:

When specifying the *`zonemap_refresh_clause`*, you must specify at least one clause after the `REFRESH` keyword.

Semantics

schema

Specify the schema containing the zone map. If you omit `schema`, then Oracle Database assumes the zone map is in your own schema.

zonemap_name

Specify the name of the zone map to be altered.

alter_zonemap_attributes

Use this clause to modify the following attributes for the zone map: `PCTFREE`, `PCTUSED`, and `CACHE` or `NOCACHE`. These attributes have the same semantics for `ALTER MATERIALIZED ZONEMAP` and `CREATE MATERIALIZED ZONEMAP`. For complete information on these attributes, refer to PCTFREE, PCTUSED, and CACHE | NOCACHE in the documentation on `CREATE MATERIALIZED ZONEMAP`.

zonemap_refresh_clause

Use this clause to modify the default refresh method and mode for the zone map. This clause has the same semantics for `ALTER MATERIALIZED ZONEMAP` and `CREATE MATERIALIZED ZONEMAP`. For complete information on this clause, refer to *zonemap_refresh_clause* in the documentation on `CREATE MATERIALIZED ZONEMAP`.

ENABLE | DISABLE PRUNING

Use this clause to enable or disable use of the zone map for pruning. This clause has the same semantics for `ALTER MATERIALIZED ZONEMAP` and `CREATE MATERIALIZED ZONEMAP`. For complete information on this clause, refer to ENABLE | DISABLE PRUNING in the documentation on `CREATE MATERIALIZED ZONEMAP`

COMPILE

This clause lets you explicitly compile the zone map. This operation validates the zone map after a DDL operation changes the structure of one or more of its base tables. It is usually not necessary to issue this clause because Oracle database automatically compiles a zone map that requires compilation before using it. However, if you would like to explicitly compile a zone map, then you can use this clause to do so.

The result of compiling a zone map depends on whether a base table is changed in a way that affects the zone map. For example, if a column is added to a base table, then the zone map will be valid after compilation because the change does not affect the zone map. However, if a column that is included in the defining subquery of the zone map is dropped from a base table, then the zone map will be invalid after compilation.

You can determine if a zone map requires compilation by querying the `COMPILE_STATE` column of the `ALL_`, `DBA_`, and `USER_ZONEMAPS` data dictionary views. If the value of the column is `NEEDS_COMPILE`, then the zone map requires compilation.

REBUILD

This clause lets you explicitly rebuild the zone map. This operation refreshes the data in the zone map. This clause is useful in the following situations:

- You can use this clause to refresh the data for a refresh-on-demand zone map. Refer to the ON DEMAND clause in the documentation on `CREATE MATERIALIZED ZONEMAP` for more information.
- You must issue this clause after an `EXCHANGE PARTITION` operation on one of the base tables of a zone map, regardless of the default refresh mode of the zone map.
- If a zone map is marked unusable, then you must issue this clause to mark it usable. You can determine if a zone map is marked unusable by querying the `UNUSABLE` column of the `ALL_`, `DBA_`, and `USER_ZONEMAPS` data dictionary views.

UNUSABLE

Specify this clause to make the zone map unusable. Subsequent queries will not use the zone map and the database will no longer maintain the zone map. You can make the zone map usable again by issuing an `ALTER MATERIALIZED ZONEMAP` ... `REBUILD` statement.

Examples

Modifying Zone Map Attributes: Example

The following statement modifies the `PCTFREE` and `PCTUSED` attributes of zone map `sales_zmap`, and modifies the zone map so that it does not use caching:

```
ALTER MATERIALIZED ZONEMAP sales_zmap
  PCTFREE 20 PCTUSED 50 NOCACHE;
```

Modifying the Default Refresh Method and Mode for a Zone Map: Example

The following statement changes the default refresh method to `FAST` and the default refresh mode to `ON COMMIT` for zone map `sales_zmap`:

```
ALTER MATERIALIZED ZONEMAP sales_zmap
  REFRESH FAST ON COMMIT;
```

Disabling Use of a Zone Map for Pruning: Example

The following statement disables use of zone map `sales_zmap` for pruning:

```
ALTER MATERIALIZED ZONEMAP sales_zmap
  DISABLE PRUNING;
```

Compiling a Zone Map: Example

The following statement compiles zone map `sales_zmap`:

```
ALTER MATERIALIZED ZONEMAP sales_zmap
  COMPILE;
```

Rebuilding a Zone Map: Example

The following statement rebuilds zone map `sales_zmap`:

```
ALTER MATERIALIZED ZONEMAP sales_zmap
  REBUILD;
```

Making a Zone Map Unusable: Example

The following statement makes zone map `sales_zmap` unusable:

```
ALTER MATERIALIZED ZONEMAP sales_zmap
  UNUSABLE;
```

ALTER OPERATOR

Purpose

Use the `ALTER OPERATOR` statement to add bindings to, drop bindings from, or compile an existing operator.

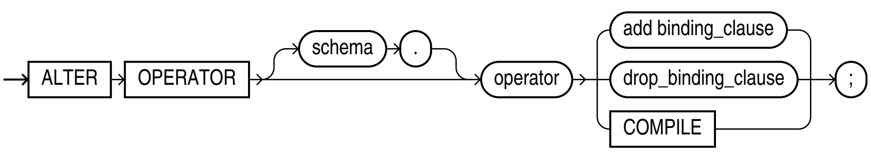
> **See Also:**
>
> CREATE OPERATOR

Prerequisites

The operator must already have been created by a previous `CREATE OPERATOR` statement. The operator must be in your own schema or you must have the `ALTER ANY OPERATOR` system privilege. You must have the `EXECUTE` object privilege on the operators and functions referenced in the `ALTER OPERATOR` statement.

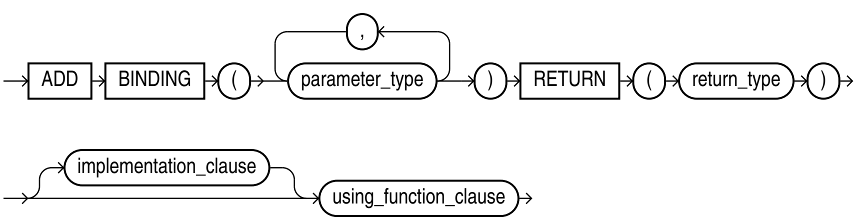
Syntax

***alter_operator*::=**



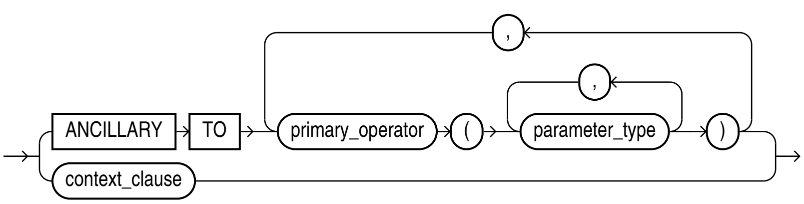
(*add_binding_clause::=*, *drop_binding_clause::=*)

***add_binding_clause*::=**



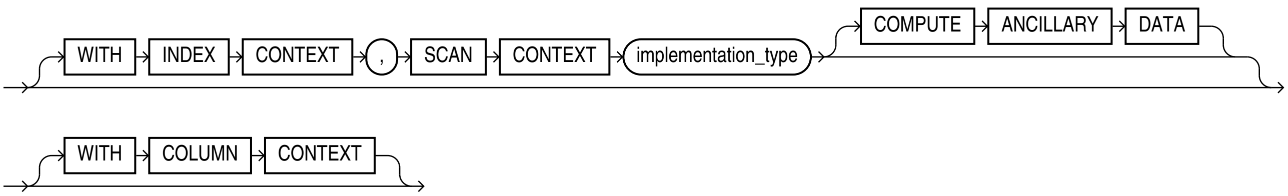
(*implementation_clause::=*, *using_function_clause::=*)

***implementation_clause*::=**

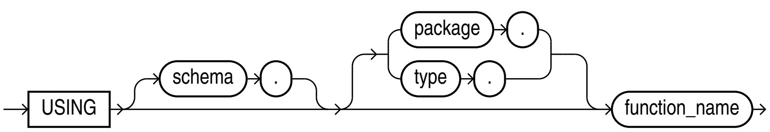


(*context_clause*::=)

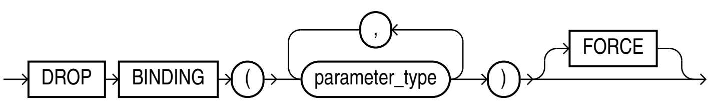
***context_clause*::=**



***using_function_clause*::=**



***drop_binding_clause*::=**



Semantics

schema

Specify the schema containing the operator. If you omit this clause, then Oracle Database assumes the operator is in your own schema.

operator

Specify the name of the operator to be altered.

add_binding_clause

Use this clause to add an operator binding and specify its parameter data types and return type. The signature must be different from the signature of any existing binding for this operator.

If a binding of an operator is associated with an indextype and you add another binding to the operator, then Oracle Database does not automatically associate the new binding with the indextype. If you want to make such an association, then you must issue an explicit `ALTER INDEXTYPE` ... `ADD OPERATOR` statement.

implementation_clause

This clause has the same semantics in `CREATE OPERATOR` and `ALTER OPERATOR` statements. For full information, refer to *implementation_clause* in the documentation on `CREATE OPERATOR`.

context_clause

This clause has the same semantics in CREATE OPERATOR and ALTER OPERATOR statements. For full information, refer to context_clause in the documentation on CREATE OPERATOR.

using_function_clause

This clause has the same semantics in CREATE OPERATOR and ALTER OPERATOR statements. For full information, refer to *using_function_clause* in the documentation on CREATE OPERATOR.

drop_binding_clause

Use this clause to specify the list of parameter data types of the binding you want to drop from the operator. You must specify FORCE if the binding has any dependent objects, such as an indextype or an ancillary operator binding. If you specify FORCE, then Oracle Database marks INVALID all objects that are dependent on the binding. The dependent objects are revalidated the next time they are referenced in a DDL or DML statement or a query.

You cannot use this clause to drop the only binding associated with this operator. Instead you must use the DROP OPERATOR statement. Refer to DROP OPERATOR for more information.

COMPILE

Specify COMPILE to cause Oracle Database to recompile the operator.

Examples

Compiling a User-defined Operator: Example

The following example compiles the operator eq_op (which was created in "Creating User-Defined Operators: Example"):

```
ALTER OPERATOR eq_op COMPILE;
```

ALTER OUTLINE

Purpose

> **Note:**
>
> Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. SQL plan management creates SQL plan baselines, which offer superior SQL performance stability compared with stored outlines.
>
> You can migrate existing stored outlines to SQL plan baselines by using the `MIGRATE_STORED_OUTLINE` function of the `DBMS_SPM` package or Enterprise Manager Cloud Control. When the migration is complete, the stored outlines are marked as migrated and can be removed. You can drop all migrated stored outlines on your system by using the `DROP_MIGRATED_STORED_OUTLINE` function of the `DBMS_SPM` package.
>
> See Also: *Oracle Database SQL Tuning Guide* for more information about SQL plan management and *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SPM` package

Use the `ALTER OUTLINE` statement to rename a stored outline, reassign it to a different category, or regenerate it by compiling the outline's SQL statement and replacing the old outline data with the outline created under current conditions.

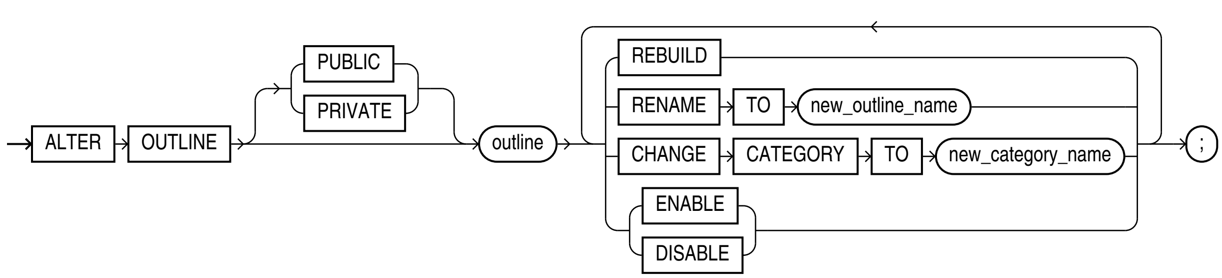
> **See Also:**
>
> CREATE OUTLINE for information on creating an outline

Prerequisites

To modify an outline, you must have the `ALTER ANY OUTLINE` system privilege.

Syntax

***alter_outline*::=**



Semantics

PUBLIC | PRIVATE

Specify `PUBLIC` if you want to modify the public version of this outline. This is the default.

Specify `PRIVATE` if you want to modify an outline that is private to the current session and whose data is stored in the current parsing schema.

outline

Specify the name of the outline to be modified.

REBUILD

Specify `REBUILD` to regenerate the execution plan for *`outline`* using current conditions.

> **See Also:**
>
> "Rebuilding an Outline: Example"

RENAME TO Clause

Use the `RENAME TO` clause to specify an outline name to replace *`outline`*.

CHANGE CATEGORY TO Clause

Use the `CHANGE CATEGORY TO` clause to specify the name of the category into which the *`outline`* will be moved.

ENABLE | DISABLE

Use this clause to selectively enable or disable this outline. Outlines are enabled by default. The `DISABLE` keyword lets you disable one outline without affecting the use of other outlines.

Examples

Rebuilding an Outline: Example

The following statement regenerates a stored outline called `salaries` by compiling the text of the outline and replacing the old outline data with the outline created under current conditions.

```
ALTER OUTLINE salaries REBUILD;
```

ALTER PACKAGE

Purpose

Packages are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the ALTER PACKAGE statement to explicitly recompile a package specification, body, or both. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects together. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

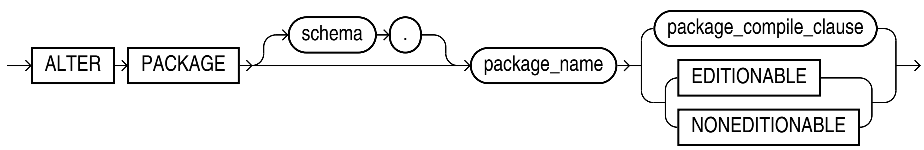
> **Note:**
>
> This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the CREATE PACKAGE or the CREATE PACKAGE BODY statement with the OR REPLACE clause.

Prerequisites

For you to modify a package, the package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

***alter_package*::=**



(*package_compile_clause*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

schema

Specify the schema containing the package. If you omit *schema*, then Oracle Database assumes the package is in your own schema.

package_name

Specify the name of the package to be recompiled.

package_compile_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling packages.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the package becomes an editioned or noneditioned object if editioning is later enabled for the schema object type PACKAGE in *schema*. The default is EDITIONABLE. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

ALTER PLUGGABLE DATABASE

Purpose

Use the ALTER PLUGGABLE DATABASE statement to modify a pluggable database (PDB). The PDB can be a traditional PDB, an application container, or an application PDB.

This statement enables you to perform the following tasks:

- Unplug a PDB from a multitenant container database (CDB) (using the *pdb_unplug_clause*)
- Modify the settings of a PDB (using the *pdb_settings_clauses*)
- Bring PDB data files online or take them offline (using the *pdb_datafile_clause*)
- Back up and recover a PDB (using the *pdb_recovery_clauses*)
- Modify the state of a PDB (using the *pdb_change_state* clause)
- Modify the state of multiple PDBs within a CDB (using the *pdb_change_state_from_root* clause)
- Perform operations on applications in an application container (using the *application_clauses*)

> **Note:**
>
> You can perform all ALTER PLUGGABLE DATABASE tasks by connecting to a PDB and running the corresponding ALTER DATABASE statement. This functionality is provided to maintain backward compatibility for applications that have been migrated to a CDB environment. The exception is modifying PDB storage limits, for which you must use the *pdb_storage_clause* of ALTER PLUGGABLE DATABASE.

> **See Also:**
>
> CREATE PLUGGABLE DATABASE for information on creating PDBs

Prerequisites

You must be connected to a CDB.

To specify the *pdb_unplug_clause*, the current container must be the root or the application root, you must be authenticated AS SYSDBA or AS SYSOPER, and the SYSDBA or SYSOPER privilege must be either granted to you commonly, or granted to you locally in the root and locally in the PDB you want to unplug.

To specify the *pdb_settings_clauses*, the current container must be the PDB whose settings you want to modify and you must have the ALTER DATABASE privilege, either granted commonly or granted locally in the PDB. To specify the *pdb_logging_clauses* or the RENAME GLOBAL_NAME clause, you must also have the RESTRICTED SESSION privilege, either granted commonly or granted locally in the PDB being renamed, and the PDB must be in READ WRITE RESTRICTED mode.

To specify the *pdb_datafile_clause*, the current container must be the PDB whose datafiles you want to bring online or take offline and you must have the ALTER DATABASE privilege, either granted commonly or granted locally in the PDB.

To specify the *pdb_recovery_clauses*, the current container must be the PDB you want to back up or recover and you must have the ALTER DATABASE privilege, either granted commonly or granted locally in the PDB.

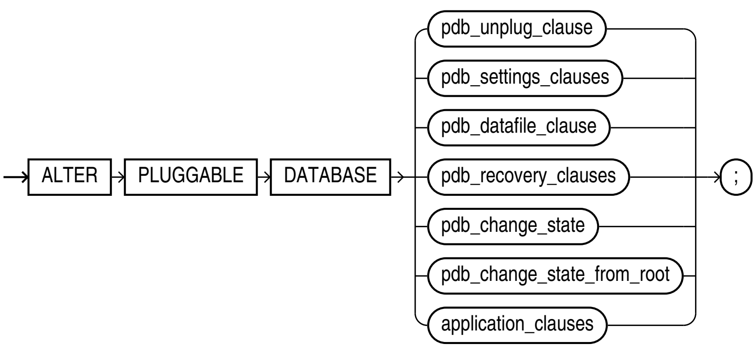
To specify the *pdb_change_state* clause, the current container must be the PDB whose state you want to change and you must be authenticated AS SYSBACKUP, AS SYSDBA, AS SYSDG, or AS SYSOPER.

To specify the *pdb_change_state_from_root* clause, the current container must be the root or the application root, you must be authenticated AS SYSBACKUP, AS SYSDBA, AS SYSDG, or AS SYSOPER, and the SYSBACKUP, SYSDBA, SYSDG, or SYSOPER privilege must be either granted to you commonly, or granted to you locally in the root or application root, and locally in the PDB(s) whose state(s) you want to change.

To specify the *application_clauses*, the current container must be an application container, you must be authenticated AS SYSBACKUP or AS SYSDBA, and the SYSBACKUP or SYSDBA privilege must be either granted to you commonly, or granted to you locally in the application root and locally in the application PDB(s) in which you want to perform application operations.

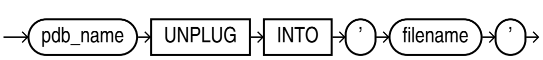
Syntax

***alter_pluggable_database*::=**

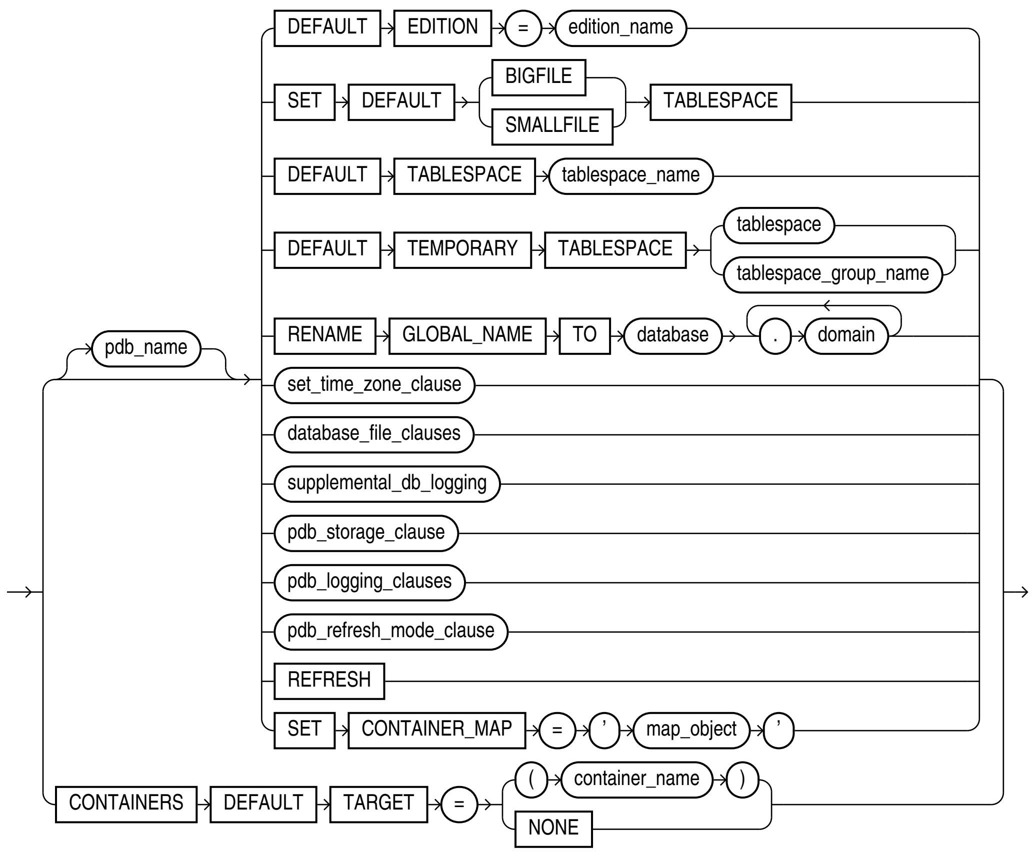


(*pdb_unplug_clause*::=, *pdb_settings_clauses*::=, *pdb_datafile_clause*::=, *pdb_recovery_clauses*, *pdb_change_state*::=, *pdb_change_state_from_root*::=, *application_clauses*::=)

***pdb_unplug_clause*::=**

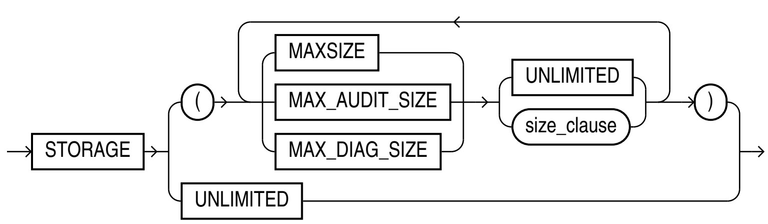


***pdb_settings_clauses*::=**



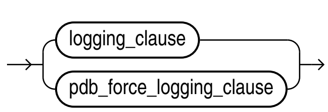
(*set_time_zone_clause*::=, *database_file_clauses*::=, *supplemental_db_logging*::=, *pdb_storage_clause*::=, *pdb_logging_clauses*::=, *pdb_refresh_mode_clause*::=)

***pdb_storage_clause*::=**

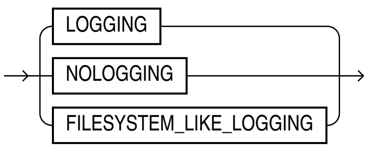


(*size_clause*::=)

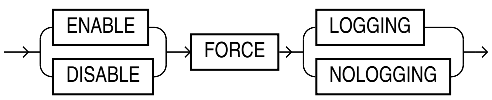
***pdb_logging_clauses*::=**



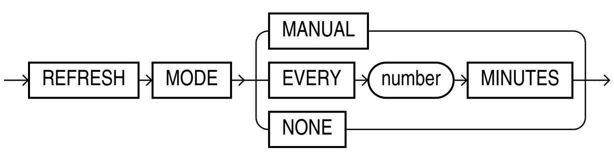
***logging_clause*::=**



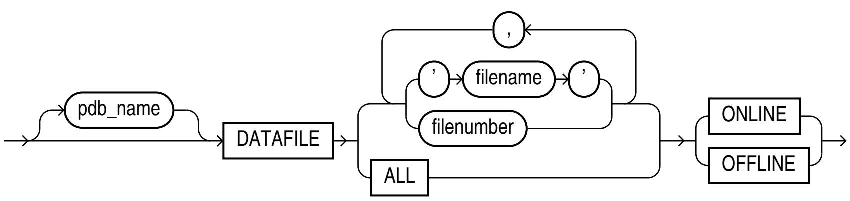
***pdb_force_logging_clause*::=**



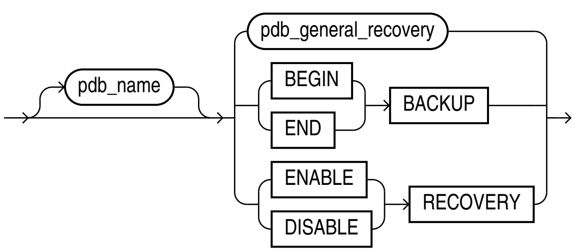
***pdb_refresh_mode_clause*::=**



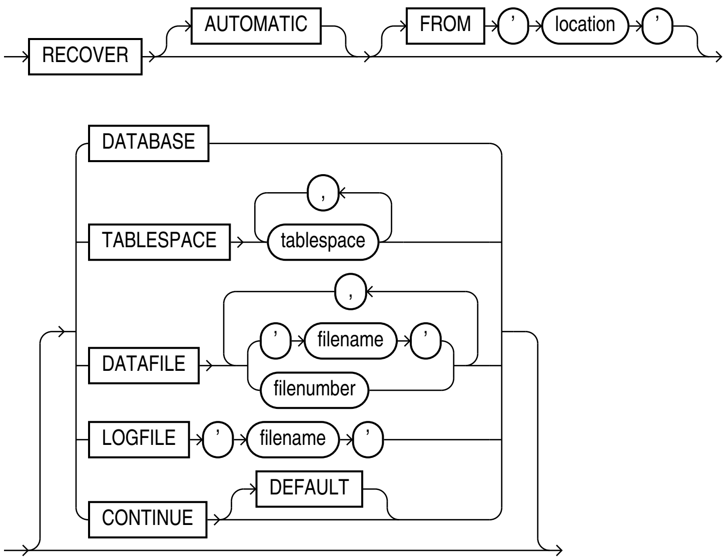
***pdb_datafile_clause*::=**



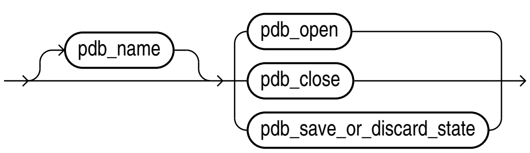
pdb_recovery_clauses



***pdb_general_recovery*::=**

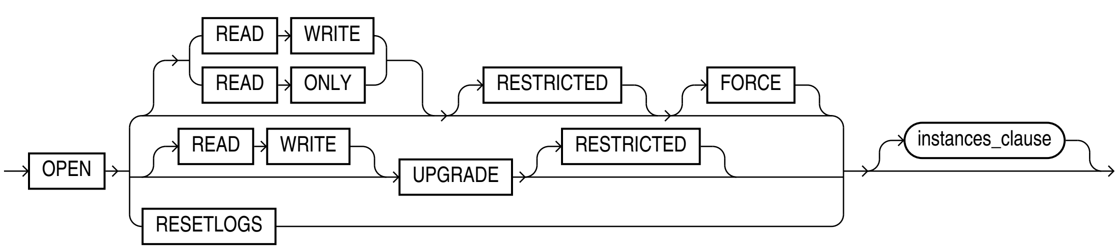


***pdb_change_state*::=**

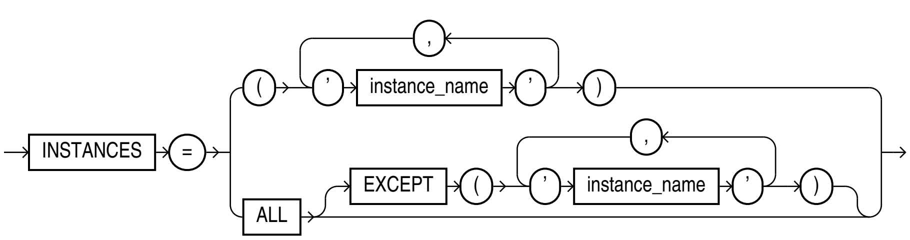


(*pdb_open*::=, *pdb_close*::=, *pdb_save_or_discard_state*::=)

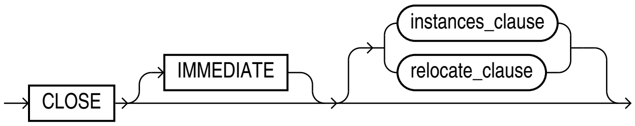
***pdb_open*::=**



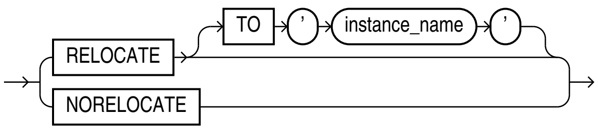
***instances_clause*::=**



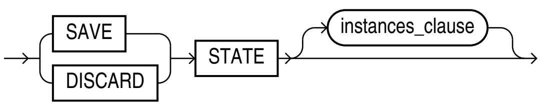
***pdb_close*::=**



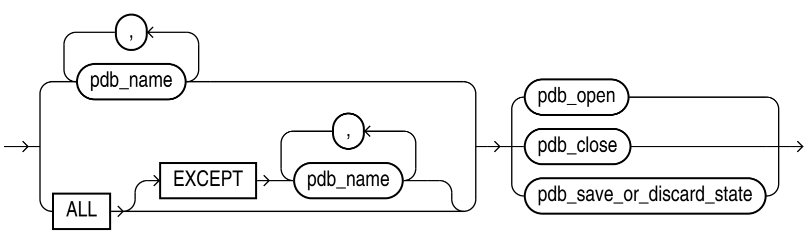
***relocate_clause*::=**



***pdb_save_or_discard_state*::=**

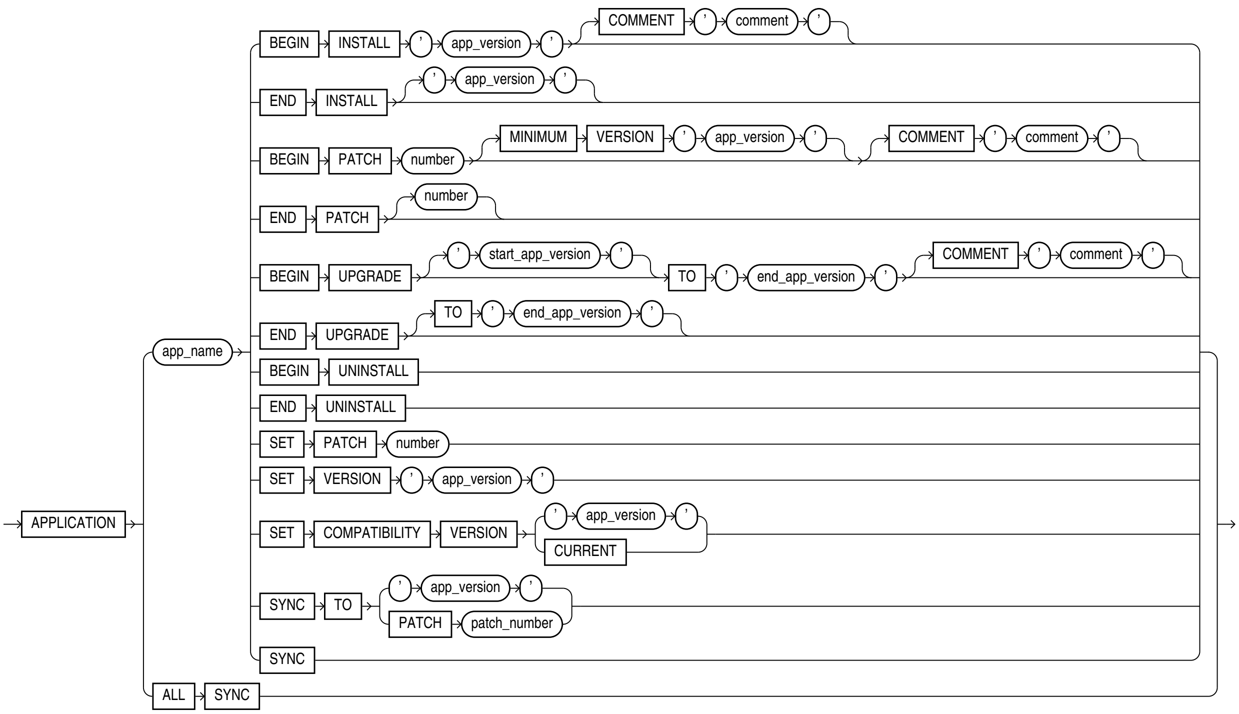


***pdb_change_state_from_root*::=**



(*pdb_open*::=, *pdb_close*::=, *pdb_save_or_discard_state*::=)

application_clauses::=



Semantics

pdb_unplug_clause

This clause lets you unplug a PDB from a CDB. When you unplug a PDB, Oracle stores information about the PDB in a file on your operating system. You can subsequently use this file to plug the PDB into a CDB.

For *pdb_name*, specify the name of the PDB you want to unplug. The PDB must be closed—that is, the open mode must be `MOUNTED`. In an Oracle Real Application Clusters (Oracle RAC) environment, the PDB must be closed in all Oracle RAC instances

For *filename*, specify the full path name of the operating system file in which to store information about the PDB. The file name that you specify determines the type of information stored and how it is stored.

- If you specify a file name that ends with the extension .xml, then Oracle creates an XML file containing metadata about the PDB. You can then copy the XML file and the PDB's data files to a new location and specify the XML file name when plugging the PDB into a CDB. In this case, you must copy the PDB's data files separately.
- If you specify a file name that ends with the extension .pdb, then Oracle creates a .pdb archive file. This is a compressed file that includes an XML file containing metadata about the PDB, as well as the PDB's data files. You can then copy this single archive file to a new location and specify the archive file name when plugging the PDB into a CDB. This eliminates having to copy the PDB's data files separately. When you use a .pdb archive file when plugging in a PDB, this file is

extracted when you plug in the PDB, and the PDB's files are placed in the same directory as the .pdb archive file.

After a PDB is unplugged, it remains in the CDB with an open mode of `MOUNTED` and a status of `UNPLUGGED`. The only operation you can perform on an unplugged PDB is `DROP PLUGGABLE DATABASE`, which will remove it from the CDB. You must drop the PDB before you can plug it into the same CDB or another CDB.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information on unplugging a PDB
> - The *create_pdb_from_xml* clause of `CREATE PLUGGABLE DATABASE` for information on plugging a PDB into a CDB

pdb_settings_clauses

These clauses lets you modify various settings for a PDB.

pdb_name

You can optionally use *`pdb_name`* to specify the name of the PDB whose settings you want to modify.

DEFAULT EDITION Clause

Use this clause to designate the specified edition as the default edition for the PDB. For the full semantics of this clause, refer to "DEFAULT EDITION Clause" in the `ALTER DATABASE` documentation.

SET DEFAULT TABLESPACE Clause

Use this clause to specify or change the default type of tablespaces subsequently created in the PDB. For the full semantics of this clause, refer to "SET DEFAULT TABLESPACE Clause" in the `ALTER DATABASE` documentation.

DEFAULT TABLESPACE Clause

Use this clause to establish or change the default permanent tablespace of the PDB. For the full semantics of this clause, refer to "DEFAULT TABLESPACE Clause" in the `ALTER DATABASE` documentation.

DEFAULT TEMPORARY TABLESPACE Clause

Use this clause to change the default temporary tablespace of the PDB to a new tablespace or tablespace group. For the full semantics of this clause, refer to "DEFAULT [LOCAL] TEMPORARY TABLESPACE Clause" in the `ALTER DATABASE` documentation.

RENAME GLOBAL_NAME TO Clause

Use this clause to change the global name of the PDB. The new global name must be unique within the CDB. For an Oracle Real Application Clusters (Oracle RAC) database, the PDB must be open in `READ WRITE RESTRICTED` mode on the current instance only. The PDB must be closed on all other instances. For the full semantics of this clause, refer to "RENAME GLOBAL_NAME Clause" in the ALTER DATABASE documentation.

> **Note:**
>
> When you change the global name of a PDB, be sure to change the PLUGGABLE DATABASE property for database services that are used to connect to the PDB.

set_time_zone_clauses

Use this clause to modify the time zone setting for the PDB. For the full semantics of this clause, refer to *set_time_zone_clause* in the ALTER DATABASE documentation.

database_file_clauses

Use this clause to modify data files and temp files for the PDB. For the full semantics of this clause, refer to *database_file_clauses* in the ALTER DATABASE documentation.

supplemental_db_logging

Use these clauses to instruct Oracle Database to add or stop adding supplemental data into the log stream for the PDB.

- Specify the ADD SUPPLEMENTAL LOG clause to add supplemental data into the log stream for the PDB. In order to issue this clause, supplemental logging must have been enabled for the CDB root with the ALTER DATABASE ... ADD SUPPLEMENTAL LOG ... statement. The level of supplemental logging that you specify for the PDB does not need to match that of the CDB root. That is, you can specify any of the clauses DATA, *supplemental_id_key_clause*, or *supplemental_plsql_clause* for the PDB, regardless of which clause was specified when enabling supplemental logging for the CDB root.
- Specify the DROP SUPPLEMENTAL LOG clause to stop adding supplemental data into the log stream for the PDB.

For the full semantics of this clause, refer to *supplemental_db_logging* in the ALTER DATABASE documentation.

pdb_storage_clause

Use this clause to modify the storage limits for a PDB.

This clause has the same semantics as the *pdb_storage_clause* in the CREATE PLUGGABLE DATABASE documentation, with the following additions:

- If you specify MAXSIZE *size_clause*, then the value you specify for *size_clause* must be greater than or equal to the combined size of the existing tablespaces belonging to the PDB. Otherwise, an error occurs.
- If you specify MAX_AUDIT_SIZE *size_clause*, then the value you specify for *size_clause* must be greater than or equal to the amount of storage used by the existing unified audit OS spillover (.bin format) files in the PDB. Otherwise, an error occurs.

- If you specify `MAX_DIAG_SIZE` *`size_clause`*, then the value you specify for *`size_clause`* must be greater than or equal to the amount of storage for diagnostics in the Automatic Diagnostic Repository (ADR) that is currently used by the PDB. Otherwise an error occurs.

pdb_logging_clauses

Use these clauses to set or change the logging characteristics of the PDB.

logging_clause

Use this clause to change the default logging attribute for tablespaces subsequently created within the PDB. This clause has the same semantics as the *logging_clause* in the `CREATE PLUGGABLE DATABASE` documentation.

pdb_force_logging_clause

Use this clause to place a PDB into force logging or force nologging mode or take a PDB out of force logging or force nologging mode.

Force logging mode instructs the database to log all changes in the PDB, except changes in temporary tablespaces and temporary segments. Force nologging mode instructs the database to not log any changes in the PDB.

CDB-wide force logging mode takes precedence over PDB-level force nologging mode. PDB-level force logging mode and force nologging mode take precedence over and are independent of any `LOGGING`, `NOLOGGING`, or `FORCE LOGGING` settings you specify for individual tablespaces in the PDB and any `LOGGING` or `NOLOGGING` settings you specify for individual database objects in the PDB.

- Specify `ENABLE FORCE LOGGING` to place the PDB in force logging mode. If the PDB is currently in force nologging mode, then specifying this clause results in an error. You must first specify `DISABLE FORCE NOLOGGING`.
- Specify `DISABLE FORCE LOGGING` to take the PDB out of force logging mode. If the PDB is not currently in force logging mode, then specifying this clause results in an error.
- Specify `ENABLE FORCE NOLOGGING` to place the PDB in force nologging mode. If the PDB is currently in force logging mode, then specifying this clause results in an error. You must first specify `DISABLE FORCE LOGGING`.
- Specify `DISABLE FORCE NOLOGGING` to take the PDB out of force nologging mode. If the PDB is not currently in force nologging mode, then specifying this clause results in an error.

This clause does not change the default `LOGGING` or `NOLOGGING` mode of the PDB specified by the *logging_clause*.

pdb_refresh_mode_clause

This clause lets you change the refresh mode of a PDB. You can specify this clause only for a refreshable PDB, that is, a PDB whose current refresh mode is `MANUAL` or `EVERY` *`number`* `MINUTES`. You can switch a PDB from manual refresh to automatic refresh, or from automatic refresh to manual refresh. You can also use this clause to change the number of minutes between automatic refreshes. You can switch a PDB from manual or automatic refresh to no refresh, but you cannot enable manual or automatic refresh for a PDB that is not refreshable. For the complete semantics of this clause, refer to the *pdb_refresh_mode_clause* in the documentation on `CREATE PLUGGABLE DATABASE`.

REFRESH

Specify this clause to perform a manual refresh of a refreshable PDB, that is, a PDB whose current refresh mode is MANUAL or EVERY *number* MINUTES. The PDB must be closed. For more information on refreshable PDBs, refer to the *pdb_refresh_mode_clause* in the documentation on CREATE PLUGGABLE DATABASE.

SET CONTAINER_MAP

Use this clause to specify the CONTAINER_MAP database property for an application container. The current container must be the application root. The *map_object* is of the form [*schema*.]*table*. For *schema*, specify the schema containing *table*. If you omit *schema*, then the database assumed that the table is in your own schema. For *table*, specify a range-, list-, or hash-partitioned table.

CONTAINERS DEFAULT TARGET

Use this clause to specify the default container for DML statements in an application container. You must be connect to the application root.

- For *container_name*, specify the name of the default container. The default container can be any container in the application container, including the application root or an application PDB. You can specify only one default container.
- If you specify NONE, then the default container is the CDB root. This is the default.

When a DML statement is issued in the application root without specifying containers in the WHERE clause, the DML statement affects the default container for the application container.

pdb_datafile_clause

This clause lets you bring data files associated with a PDB online or take them offline. The PDB must be closed when you issue this clause.

- For *pdb_name*, specify the name of the PDB. If the current container is the PDB, then you can omit *pdb_name*.
- The DATAFILE clauses let you specify the data files you want to bring online or take offline. Use *filename* or *filenumber* to identify specific data files by name or by number. You can view data file names and numbers by querying the NAME and FILE# columns of the V$DATAFILE dynamic performance view. Use ALL to specify all datafiles associated with the PDB.
- Specify ONLINE to bring the data files online or OFFLINE to take the data files offline.

pdb_recovery_clauses

Use the *pdb_recovery_clauses* to back up and recover a PDB.

pdb_name

You can optionally use *pdb_name* to specify the name of the PDB you want to back up or recover.

pdb_general_recovery

This clause lets you control media recovery for the PDB or standby database or for specified tablespaces or files. The *pdb_general_recovery* clause has the same semantics as the *general_recovery* clause of ALTER DATABASE. Refer to the *general_recovery* clause of ALTER DATABASE for more information.

BACKUP Clauses

Use these clauses to move all of the data files in the PDB into or out of online backup mode (also called hot backup mode). These clauses have the same semantics in ALTER PLUGGABLE DATABASE and ALTER DATABASE. Refer to the "BACKUP Clauses" of ALTER DATABASE for more information.

RECOVERY Clauses

Use these clauses to enable or disable a PDB for recovery. The PDB must be closed—that is, the open mode must be MOUNTED.

- Specify ENABLE RECOVERY to bring all data files that belong to a PDB online and enable the PDB for recovery.
- Specify DISABLE RECOVERY to take all data files that belong to a PDB offline and disable the PDB for recovery.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on the RECOVERY clauses

pdb_change_state

This clause enables you to change the state, or open mode, of a PDB. Table 11-2 lists the open modes of a PDB.

- Specify the *pdb_open* clause to change the open mode to READ WRITE, READ ONLY, or MIGRATE.
- Specify the *pdb_close* clause to change the open mode to MOUNTED.

Table 11-2 PDB Open Modes

Open Mode	Description
READ WRITE	A PDB in open read/write mode allows queries and user transactions to proceed and allows users to generate redo logs.
READ ONLY	A PDB in open read-only mode allows queries but does not allow user changes.
MIGRATE	When a PDB is in open migrate mode, you can run database upgrade scripts on the PDB.
MOUNTED	When a PDB is in mounted mode, it behaves like a non-CDB in mounted mode. It does not allow changes to any objects, and it is accessible only to database administrators. It cannot read from or write to data files. Information about the PDB is removed from memory caches. Cold backups of the PDB are possible.

You can view the open mode of a PDB by querying the `OPEN_MODE` column of the `V$PDBS` view.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for a complete description of PDB open modes

pdb_name

You can optionally use *`pdb_name`* to specify the name of the PDB whose open mode you want to change.

pdb_open

This clause lets you change the open mode of a PDB to `READ WRITE`, `READ ONLY`, or `MIGRATE`. When you specify this clause, the PDB must be in `MOUNTED` mode unless you specify the `FORCE` keyword.

If you do not specify `READ WRITE` or `READ ONLY`, then the default is `READ WRITE`. The exception is when the PDB belongs to a CDB that is used as a physical standby database, in which case the default is `READ ONLY`.

READ WRITE

Specify this clause to change the open mode to `READ WRITE`.

READ ONLY

Specify this clause to change the open mode to `READ ONLY`.

[READ WRITE] UPGRADE

Specify this clause to change the open mode to `MIGRATE`. The `READ WRITE` keywords are optional and are provided for semantic clarity.

RESTRICTED

If you specify the optional `RESTRICTED` keyword, then the PDB is accessible only to users with the `RESTRICTED SESSION` privilege in the PDB.

If the PDB is in `READ WRITE` or `READ ONLY` mode, and you specify the `RESTRICTED` and `FORCE` keywords while changing the open mode, then all sessions connected to the PDB that do not have the `RESTRICTED SESSION` privilege in the PDB are terminated, and their transactions are rolled back.

FORCE

Specify this keyword to change the open mode of a PDB from `READ WRITE` to `READ ONLY`, or from `READ ONLY` to `READ WRITE`. The `FORCE` keyword allows users to remain connected to the PDB while the open mode is changed.

When you specify `FORCE` to change the open mode of a PDB from `READ WRITE` to `READ ONLY`, any `READ WRITE` transaction that is open when you change the open mode will not be allowed to perform any more DML operations or to `COMMIT`.

Restriction on FORCE

You cannot specify the FORCE keyword if the PDB is currently in MIGRATE mode, and you cannot specify the FORCE keyword to change a currently open PDB to MIGRATE mode.

RESETLOGS

Specify this clause to create a new PDB incarnation and open the PDB in READ WRITE mode after point-in-time recovery of the PDB.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information on performing point-in-time recovery of CDBs and PDBs

instances_clause

In an Oracle Real Application Clusters environment, use this clause to modify the state of the PDB in the specified Oracle RAC instances. If you omit this clause, then the state of the PDB is modified only in the current instance.

- Use *instance_name* to specify one or more instance names, in a comma-separated list enclosed in parenthesis. This modifies the state of the PDB only in those instances.
- Specify ALL to modify the state of the PDB in all instances.
- Specify ALL EXCEPT to modify the state of the PDB in all instances except the specified instances.

If the PDB is already open in one or more instances, then you can open it in additional instances, but it must be opened in the same mode as in the instances in which it is already open.

pdb_close

This clause lets you change the open mode of a PDB to MOUNTED. When you specify this clause, the PDB must be in READ WRITE, READ ONLY, or MIGRATE mode. This clause is the PDB equivalent of the SQL*Plus SHUTDOWN command.

IMMEDIATE

If you specify the optional IMMEDIATE keyword, then this clause is the PDB equivalent of the SQL*Plus SHUTDOWN command with the immediate mode. Otherwise, the PDB is shut down with the normal mode.

> **See Also:**
>
> *SQL*Plus User's Guide and Reference* for more information on the SQL*Plus SHUTDOWN command

instances_clause

In an Oracle Real Application Clusters environment, use this clause to modify the state of the PDB in the specified Oracle RAC instances. You can close a PDB in some instances and leave it open in others. Refer to the instances_clause for the full semantics of this clause.

relocate_clause

In an Oracle Real Application Clusters environment, use this clause to instruct the database to reopen the PDB on a different Oracle RAC instance.

- Specify `RELOCATE` to reopen the PDB on a different instance that is selected by Oracle Database.
- Specify `RELOCATE TO 'instance_name'` to reopen the PDB in the specified instance.
- Specify `NORELOCATE` to close the PDB in the current instance. This is the default.

pdb_save_or_discard_state

Use this clause to instruct the database to save or discard the open mode of the PDB when the CDB restarts.

- If you specify `SAVE`, then the PDB's open mode after the CDB restarts will be identical to its open mode just before the CDB restarted.
- If you specify `DISCARD`, then the PDB's open mode after the CDB restarts will be `MOUNTED`. This is the default.

instances_clause

In an Oracle Real Application Clusters environment, use this clause to instruct the database to save or discard the open mode of the PDB in the specified Oracle RAC instances. If you omit this clause, then the database applies the `SAVE` or `DISCARD` setting only to the PDB in the current instance.

- Use *`instance_name`* to specify one or more instance names, in a comma-separated list enclosed in parenthesis. This applies the `SAVE` or `DISCARD` setting to the PDB only in those instances.
- Specify `ALL` to apply the `SAVE` or `DISCARD` setting to the PDB in all instances.
- Specify `ALL EXCEPT` to apply the `SAVE` or `DISCARD` setting to the PDB in all instances except the specified instances.

pdb_change_state_from_root

This clause enables you to modify the state of one or more PDBs.

- Specify the *`pdb_name`* for one or more PDBs whose state you want to modify.
- Specify `ALL` to modify the state of all PDBs in the CDB.
- Specify `ALL EXCEPT` to modify the state of all PDBs in the CDB except those specified by using *`pdb_name`*.

If a PDB is already in the specified state, then the PDB's state is unchanged and no error is returned. If the state of a PDB cannot be changed, then an error occurs only for that PDB.

Refer to pdb_open and pdb_close for the full semantics of these clauses.

application_clauses

The `APPLICATION` clauses allow you to perform the following operations in an application container:

- Install, patch, upgrade, and uninstall applications

- Register application versions and patch numbers
- Keep applications in sync between the application root and application PDBs

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on administering application containers

Specifying Application Names

Most of the *application_clauses* require you to specify an application name. The maximum length of an application name is 30 bytes. The name must satisfy the requirements listed in "Database Object Naming Rules ". The application name must be unique within an application container.

Specifying Application Versions

Several of the *application_clauses* require you to specify an application version. The application version can be up to 30 bytes in length and can contain alphanumeric characters, punctuations marks, and spaces. The application version is case-sensitive and must be enclosed in single quotation marks.

Specifying Comments

Several of the *application_clauses* allow you to specify a comment to associate with an application install, patch, or upgrade operation. For *comment*, enter a character string enclosed in single quotation marks.

INSTALL Clauses

Use the `INSTALL` clauses when installing an application in an application container. The current container must be the application root, not an application PDB.

- Specify the `BEGIN INSTALL` clause before you start installing the application.
  - Use *app_name* to assign a name to the application.
  - Use *app_version* to assign a version to the application.
  - The optional `COMMENT` clause allows you to enter a comment to be associated with the application version created by this installation.
- Specify the `END INSTALL` clause after you have finished installing the application.
  - You must specify the same *app_name* that you specified for the corresponding `BEGIN INSTALL` clause.
  - You need not specify *app_version*, but if you do, then you must specify the same version that you specified for the corresponding `BEGIN INSTALL` clause.

PATCH Clauses

Use the `PATCH` clauses when patching an application in an application container. The current container must be the application root, not an application PDB.

- Specify the `BEGIN PATCH` clause before you start patching the application.
  - For *app_name*, specify the name of the application you want to patch.

  - For *number*, specify the patch number.
  - The optional MINIMUM VERSION clause allows you to specify the minimum version at which the application must be before the patch can be applied. For *app_version*, specify the minimum application version. If the current application version is lower than the minimum application version, then an error occurs. If you omit this clause, then the minimum version is the current application version.
  - The optional COMMENT clause allows you to enter a comment to be associated with the patch.
- Specify the END PATCH clause after you finish patching the application.
  - You must specify the same *app_name* that you specified for the corresponding BEGIN PATCH clause.
  - You need not specify *number*, but if you do, then you must specify the same value that you specified for the corresponding BEGIN PATCH clause.

UPGRADE Clauses

Use the UPGRADE clauses when upgrading an application in an application container. The current container must be the application root, not an application PDB.

- Specify the BEGIN UPGRADE clause before you start upgrading the application.
  - For *app_name*, specify the name of the application you want to upgrade.
  - For *start_app_version*, specify the version from which you are upgrading the application. If this version does not match the current application version, then an error occurs.
  - For *end_app_version*, specify the version to which you are upgrading the application.
  - The optional COMMENT clause allows you to enter a comment to be associated with the upgrade.
- Specify the END UPGRADE clause after you finish upgrading the application.
  - You must specify the same *app_name* that you specified for the corresponding BEGIN UPGRADE clause.
  - You need not specify TO *end_app_version*, but if you do, then you must specify the same version that you specified for the corresponding BEGIN UPGRADE clause.

UNINSTALL Clauses

Use the UNINSTALL clauses when uninstalling an application from an application container. The current container must be the application root, not an application PDB.

- Specify the BEGIN UNINSTALL clause before you start uninstalling the application.
  - For *app_name*, specify the name of the application you want to uninstall.
- Specify the END UNINSTALL clause after you have finished uninstalling the application.
  - You must specify the same *app_name* that you specified for the corresponding BEGIN UNINSTALL clause.

SET PATCH

Use the `SET PATCH` clause to register the patch number of an application that is already installed in an application container. This clause allows you to assign a patch number to an application that was not patched using the `PATCH` clauses. This is useful if the application was migrated from a PDB in an earlier Oracle Database release, when the `PATCH` clauses were not available. The current container can be the application root or an application PDB.

- For *`app_name`*, specify the name of an existing application.
- Use *`number`* to assign a patch number to the existing application.

SET VERSION

Use the `SET VERSION` clause to register the version of an application that is already installed in an application container. This clause allows you to assign a name and a version to an application that was not installed using the `INSTALL` clauses. This is useful if the application was migrated from a PDB in an earlier Oracle Database release, when the `INSTALL` clauses were not available. The current container can be the application root or an application PDB.

- Use *`app_name`* to assign a name to the existing application.
- Use *`app_version`* to assign a version to the existing application.

SET COMPATIBILITY VERSION

Use the `SET COMPATIBILITY VERSION` clause to set the compatibility version for an application.

The compatibility version of an application is the earliest version of the application possible for the application PDBs that belong to the application container. The current container must be the application root, not an application PDB.

> **Note:**
>
> You cannot plug in an application PDB that uses an application version earlier than the compatibility setting of the application container.

- Use *`app_name`* to specify the name of the application.
- Use *`app_version`* to specify the compatibility version for the application.
- If you specify `CURRENT`, then the compatibility version is set to the version of the application in the application root.

The compatibility version is enforced when the compatibility version is set and when an application PDB is created. If there are application root clones that resulted from application upgrades, then all application root clones that correspond to versions earlier than the compatibility version are implicitly dropped.

SYNC TO

You can synchronize an application to a particular version or a patch number. There are two variations:

1. `SYNC TO` *`app_version`*
2. `SYNC TO PATCH` *`patch_number`*

Example

Assume that you perform the following operations on application `salesapp` :

1. Install version 1.0
2. Patch 101
3. Upgrade to version 2.0
4. Patch 102
5. Upgrade to 3.0

`ALTER PLUGGABLE DATABASE APPLICATION salesapp SYNC TO 2.0` replays all statements up to and including ' Upgrade to version 2.0'.

`ALTER PLUGGABLE DATABASE APPLICATION salesapp SYNC TO PATCH 102` replays all statements up to and including ' Patch 102'.

SYNC

Use the `SYNC` clause to sync an application in an application PDB to the version and patch level of the same application in the application root. This is useful after changes have been made to the application in the application root. The current container must be an application PDB.

- For *`app_name`*, specify the name of an application that exists in the application root. The application may or may not exist in the application PDB.

ALL SYNC

Use the `ALL SYNC` clause to sync all applications in an application PDB with all applications in the application root. This clause is useful if you have recently added the application PDB to the CDB and would like to sync its applications with the application container. The current container must be an application PDB.

Examples

Unplugging a PDB from a CDB: Example

The following statement unplugs PDB `pdb1` and stores metadata for the PDB into XML file `/oracle/data/pdb1.xml`:

```
ALTER PLUGGABLE DATABASE pdb1
  UNPLUG INTO '/oracle/data/pdb1.xml';
```

Modifying the Settings of a PDB: Example

The following statement changes the limit for the amount of storage used by all tablespaces in PDB `pdb2` to 500M:

```
ALTER PLUGGABLE DATABASE pdb2
  STORAGE (MAXSIZE 500M);
```

Taking the Data Files of a PDB Offline: Example

The following statement takes the data files associated with PDB `pdb3` offline:

```
ALTER PLUGGABLE DATABASE pdb3
  DATAFILE ALL OFFLINE;
```

Changing the State of a PDB: Examples

Assume that PDB `pdb4` is closed—that is, its open mode is `MOUNTED`. The following statement opens `pdb4` with open mode `READ ONLY`:

```
ALTER PLUGGABLE DATABASE pdb4
  OPEN READ ONLY;
```

The following statement uses the `FORCE` keyword to change the open mode of `pdb4` from `READ ONLY` to `READ WRITE`:

```
ALTER PLUGGABLE DATABASE pdb4
  OPEN READ WRITE FORCE;
```

The following statement closes PDB `pdb4`:

```
ALTER PLUGGABLE DATABASE pdb4
  CLOSE;
```

The following statement opens PDB pdb4 with open mode `READ ONLY`. Because the `RESTRICTED` keyword is specified, the PDB is accessible only to users with the `RESTRICTED SESSION` privilege in the PDB.

```
ALTER PLUGGABLE DATABASE pdb4
  OPEN READ ONLY RESTRICTED;
```

Assume that PDB `pdb5` is closed—that is, its open mode is `MOUNTED`. In an Oracle Real Application Clusters environment, the following statement opens PDB `pdb5` with open mode `READ WRITE` in instances `ORCLDB_1` and `ORCLDB_2`:

```
ALTER PLUGGABLE DATABASE pdb5
  OPEN READ WRITE INSTANCES = ('ORCLDB_1', 'ORCLDB_2');
```

In an Oracle Real Application Clusters environment, the following statement closes PDB `pdb6` in the current instance and instructs the database to reopen `pdb6` in instance `ORCLDB_3`:

```
ALTER PLUGGABLE DATABASE pdb6
  CLOSE RELOCATE TO 'ORCLDB_3';
```

Changing the State of All PDBs in a CDB: Example

Assume that the current container is the root. The following statement opens all PDBs in the CDB with open mode `READ ONLY`:

```
ALTER PLUGGABLE DATABASE ALL
  OPEN READ ONLY;
```

ALTER PROCEDURE

Purpose

Packages are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `ALTER PROCEDURE` statement to explicitly recompile a standalone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the `ALTER PACKAGE` statement (see ALTER PACKAGE).

> **Note:**
>
> This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a procedure, use the `CREATE PROCEDURE` statement with the `OR REPLACE` clause (see CREATE PROCEDURE).

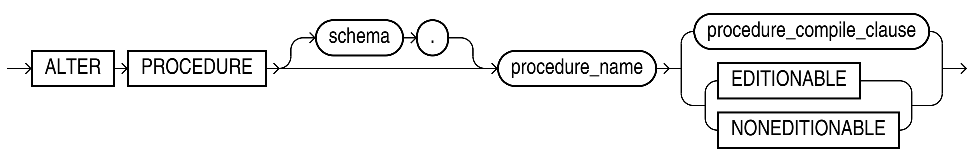
The `ALTER PROCEDURE` statement is quite similar to the `ALTER FUNCTION` statement. Refer to ALTER FUNCTION for more information.

Prerequisites

The procedure must be in your own schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

***alter_procedure*::=**



(*procedure_compile_clause*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

schema

Specify the schema containing the procedure. If you omit *schema*, then Oracle Database assumes the procedure is in your own schema.

procedure_name

Specify the name of the procedure to be recompiled.

procedure_compile_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling procedures.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the procedure becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `PROCEDURE` in *schema*. The default is `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

ALTER PROFILE

Purpose

Use the `ALTER PROFILE` statement to add, modify, or remove a resource limit or password management parameter in a profile.

Changes made to a profile with an `ALTER PROFILE` statement affect users only in their subsequent sessions, not in their current sessions.

> **See Also:**
>
> CREATE PROFILE for information on creating a profile

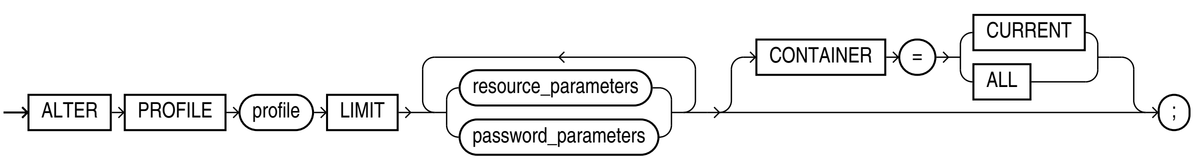
Prerequisites

You must have the `ALTER PROFILE` system privilege.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB).

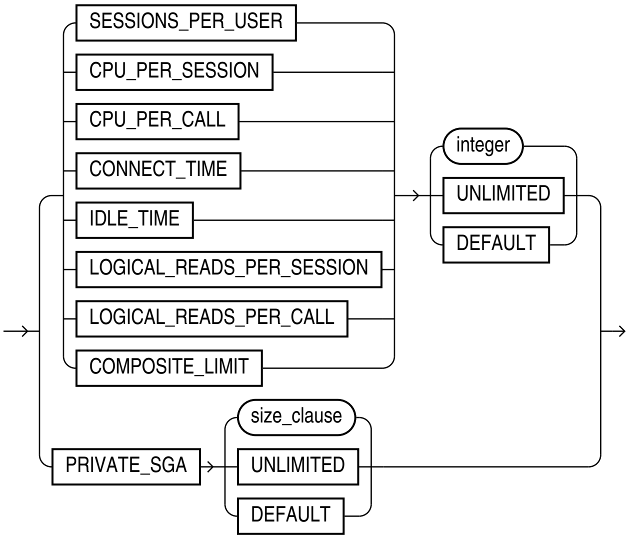
Syntax

alter_profile::=



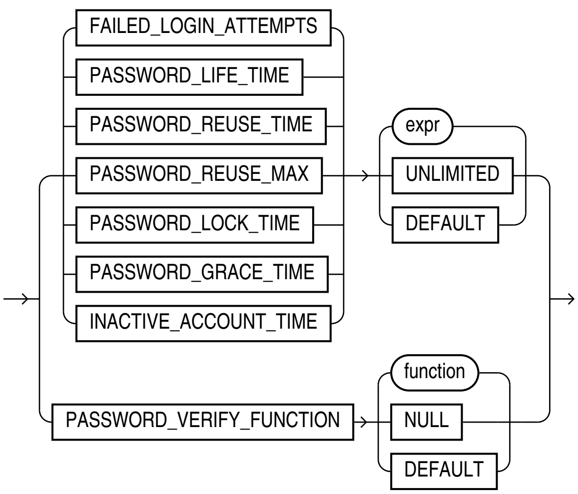
(*resource_parameters*::=, *password_parameters*::=)

resource_parameters::=



(*size_clause*::=)

***password_parameters*::=**



Semantics

The keywords, parameters, and clauses in the `ALTER PROFILE` statement all have the same meaning as in the `CREATE PROFILE` statement.

You cannot remove a limit from the `DEFAULT` profile.

Refer to CREATE PROFILE and to the examples in the next section for more information.

Examples

Making a Password Unavailable: Example

The following statement makes the password of the new_profile profile (created in "Creating a Profile: Example") unavailable for reuse for 90 days:

```
ALTER PROFILE new_profile
   LIMIT PASSWORD_REUSE_TIME 90
   PASSWORD_REUSE_MAX UNLIMITED;
```

Setting Default Password Values: Example

The following statement defaults the PASSWORD_REUSE_TIME value of the app_user profile (created in "Setting Profile Resource Limits: Example") to its defined value in the DEFAULT profile:

```
ALTER PROFILE app_user
   LIMIT PASSWORD_REUSE_TIME DEFAULT
   PASSWORD_REUSE_MAX UNLIMITED;
```

Limiting Login Attempts and Password Lock Time: Example

The following statement alters profile app_user with FAILED_LOGIN_ATTEMPTS set to 5 and PASSWORD_LOCK_TIME set to 1:

```
ALTER PROFILE app_user LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LOCK_TIME 1;
```

This statement causes any user account to which the app_user profile is assigned to become locked for one day after five consecutive unsuccessful login attempts.

Changing Password Lifetime and Grace Period: Example

The following statement modifies the profile app_user2 PASSWORD_LIFE_TIME to 90 days and PASSWORD_GRACE_TIME to 5 days:

```
ALTER PROFILE app_user2 LIMIT
   PASSWORD_LIFE_TIME 90
   PASSWORD_GRACE_TIME 5;
```

Limiting Account Inactivity: Example

The following statement modifies the profile app_user2 INACTIVE_ACCOUNT_TIME to 30 consecutive days:

```
ALTER PROFILE app_user2 LIMIT
  INACTIVE_ACCOUNT_TIME 30;
```

If the account has already been inactive for a certain number of days, then those days count toward the new 30 day limit.

Limiting Concurrent Sessions: Example

This statement defines a new limit of 5 concurrent sessions for the app_user profile:

```
ALTER PROFILE app_user LIMIT SESSIONS_PER_USER 5;
```

If the app_user profile does not currently define a limit for SESSIONS_PER_USER, then the preceding statement adds the limit of 5 to the profile. If the profile already defines a limit, then the preceding statement redefines it to 5. Any user assigned the app_user profile is subsequently limited to 5 concurrent sessions.

Removing Profile Limits: Example

This statement removes the `IDLE_TIME` limit from the `app_user` profile:

```
ALTER PROFILE app_user LIMIT IDLE_TIME DEFAULT;
```

Any user assigned the `app_user` profile is subject in their subsequent sessions to the `IDLE_TIME` limit defined in the `DEFAULT` profile.

Limiting Profile Idle Time: Example

This statement defines a limit of 2 minutes of idle time for the `DEFAULT` profile:

```
ALTER PROFILE default LIMIT IDLE_TIME  2;
```

This `IDLE_TIME` limit applies to these users:

- Users who are not explicitly assigned any profile
- Users who are explicitly assigned a profile that does not define an `IDLE_TIME` limit

This statement defines unlimited idle time for the `app_user2` profile:

```
ALTER PROFILE app_user2 LIMIT IDLE_TIME UNLIMITED;
```

Any user assigned the `app_user2` profile is subsequently permitted unlimited idle time.

ALTER RESOURCE COST

Purpose

Use the `ALTER RESOURCE COST` statement to specify or change the formula by which Oracle Database calculates the total resource cost used in a session.

Although Oracle Database monitors the use of other resources, only the four resources shown in the syntax can contribute to the total resource cost for a session.

This statement lets you apply weights to the four resources. Oracle Database then applies the weights to the value of these resources that were specified for a profile to establish a formula for calculating total resource cost. You can limit this cost for a session with the `COMPOSITE_LIMIT` parameter of the `CREATE PROFILE` statement. If the resource cost of a session exceeds the limit, then Oracle Database aborts the session and returns an error. If you use the `ALTER RESOURCE COST` statement to change the weight assigned to each resource, then Oracle Database uses these new weights to calculate the total resource cost for all current and subsequent sessions.

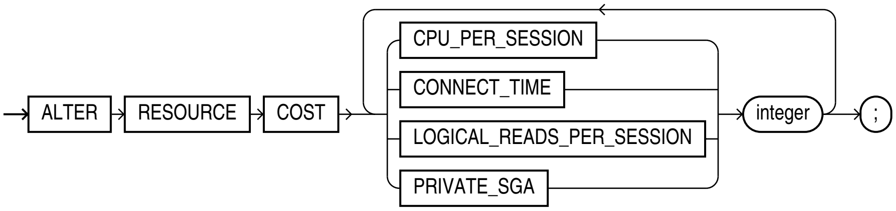
> **See Also:**
>
> CREATE PROFILE for information on all resources and on establishing resource limits

Prerequisites

You must have the `ALTER RESOURCE COST` system privilege.

Syntax

alter_resource_cost::=



Semantics

Oracle Database calculates the total resource cost by first multiplying the amount of each resource used in the session by the weight of the resource, and then summing the products for all four resources. For any session, this cost is limited by the value of the `COMPOSITE_LIMIT` parameter in the user's profile. Both the products and the total cost are expressed in units called **service units**.

CPU_PER_SESSION

Use this keyword to apply a weight to the `CPU_PER_SESSION` resource.

CONNECT_TIME

Use this keyword to apply a weight to the `CONNECT_TIME` resource.

LOGICAL_READS_PER_SESSION

Use this clause to apply a weight to the `LOGICAL_READS_PER_SESSION` resource. Logical reads include blocks read from both memory and disk.

PRIVATE_SGA

Use this clause to apply a weight to the `PRIVATE_SGA` resource. This limit applies only if you are using shared server architecture and allocating private space in the SGA for your session.

integer

Specify the weight of each resource. The weight that you assign to each resource determines how much the use of that resource contributes to the total resource cost. If you do not assign a weight to a resource, then the weight defaults to 0, and use of the resource subsequently does not contribute to the cost. The weights you assign apply to all subsequent sessions in the database.

Examples

Altering Resource Costs: Examples

The following statement assigns weights to the resources `CPU_PER_SESSION` and `CONNECT_TIME`:

```
ALTER RESOURCE COST
   CPU_PER_SESSION 100
   CONNECT_TIME      1;
```

The weights establish this cost formula for a session:

```
cost = (100 * CPU_PER_SESSION) + (1 * CONNECT_TIME)
```

In this example, the values of `CPU_PER_SESSION` and `CONNECT_TIME` are either values in the `DEFAULT` profile or in the profile of the user of the session.

Because the preceding statement assigns no weight to the resources `LOGICAL_READS_PER_SESSION` and `PRIVATE_SGA`, these resources do not appear in the formula.

If a user is assigned a profile with a `COMPOSITE_LIMIT` value of 500, then a session exceeds this limit whenever `cost` exceeds 500. For example, a session using 0.04 seconds of CPU time and 101 minutes of elapsed time exceeds the limit. A session using 0.0301 seconds of CPU time and 200 minutes of elapsed time also exceeds the limit.

You can subsequently change the weights with another `ALTER RESOURCE` statement:

```
ALTER RESOURCE COST
   LOGICAL_READS_PER_SESSION 2
   CONNECT_TIME 0;
```

These new weights establish a new cost formula:

```
cost = (100 * CPU_PER_SESSION) + (2 * LOGICAL_READ_PER_SECOND)
```

where the values of `CPU_PER_SESSION` and `LOGICAL_READS_PER_SECOND` are either the values in the `DEFAULT` profile or in the profile of the user of this session.

This `ALTER RESOURCE COST` statement changes the formula in these ways:

- The statement omits a weight for the `CPU_PER_SESSION` resource. That resource was already assigned a weight, so the resource remains in the formula with its original weight.
- The statement assigns a weight to the `LOGICAL_READS_PER_SESSION` resource, so this resource now appears in the formula.
- The statement assigns a weight of 0 to the `CONNECT_TIME` resource, so this resource no longer appears in the formula.
- The statement omits a weight for the `PRIVATE_SGA` resource. That resource was not already assigned a weight, so the resource still does not appear in the formula.

ALTER ROLE

Purpose

Use the `ALTER ROLE` statement to change the authorization needed to enable a role.

See Also:

- CREATE ROLE for information on creating a role
- SET ROLE for information on enabling or disabling a role for your session

Prerequisites

You must either have been granted the role with the `ADMIN OPTION` or have `ALTER ANY ROLE` system privilege.

Before you alter a role to `IDENTIFIED GLOBALLY`, you must:

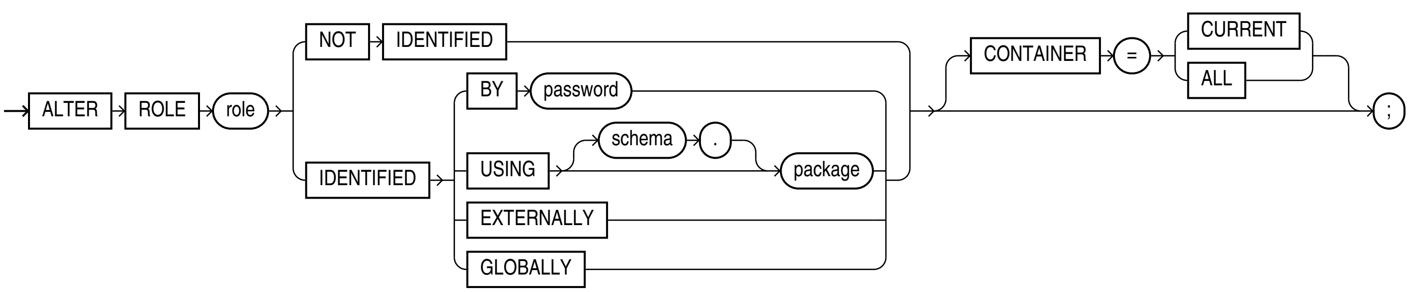
- Revoke all grants of roles identified externally to the role and
- Revoke the grant of the role from all users, roles, and `PUBLIC`.

The one exception to this rule is that you should not revoke the role from the user who is currently altering the role.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB).

Syntax

***alter_role*::=**



Semantics

The keywords, parameters, and clauses in the `ALTER ROLE` statement all have the same meaning as in the `CREATE ROLE` statement.

Restriction on Altering a Role

You cannot alter a `NOT IDENTIFIED` role to any of the `IDENTIFIED` types if it is granted to another role.

Notes on Altering a Role

The following notes apply when altering a role:

- User sessions in which the role is already enabled are not affected.

- If you change a role identified by password to an application role (with the `USING` *`package`* clause), then password information associated with the role is lost. Oracle Database will use the new authentication mechanism the next time the role is to be enabled.
- If you have the `ALTER ANY ROLE` system privilege and you change a role that is `IDENTIFIED GLOBALLY` to `IDENTIFIED BY` *`password`*, `IDENTIFIED EXTERNALLY`, or `NOT IDENTIFIED`, then Oracle Database grants you the altered role with the `ADMIN OPTION`, as it would have if you had created the role identified nonglobally.

For more information, refer to CREATE ROLE and to the examples that follow.

Examples

Changing Role Identification: Example

The following statement changes the role `warehouse_user` (created in "Creating a Role: Example") to `NOT IDENTIFIED`:

```
ALTER ROLE warehouse_user NOT IDENTIFIED;
```

Changing a Role Password: Example

This statement changes the password on the `dw_manager` role (created in "Creating a Role: Example") to `data`:

```
ALTER ROLE dw_manager
   IDENTIFIED BY data;
```

Users granted the `dw_manager` role must subsequently use the new password `data` to enable the role.

Application Roles: Example

The following example changes the `dw_manager` role to an application role using the `hr.admin` package:

```
ALTER ROLE dw_manager IDENTIFIED USING hr.admin;
```

ALTER ROLLBACK SEGMENT

> **Note:**
>
> Oracle strongly recommends that you run your database in automatic undo management mode instead of using rollback segments. Do not use rollback segments unless you must do so for compatibility with earlier versions of Oracle Database. Refer to *Oracle Database Administrator's Guide* for information on automatic undo management.

Purpose

Use the `ALTER ROLLBACK SEGMENT` statement to bring a rollback segment online or offline, change its storage characteristics, or shrink it to an optimal or specified size.

This section assumes that your database is running in rollback undo mode (the UNDO_MANAGEMENT initialization parameter is set to MANUAL or not set at all). If your database is running in automatic undo mode (the UNDO_MANAGEMENT initialization parameter is set to AUTO, which is the default), then user-created rollback segments are irrelevant.

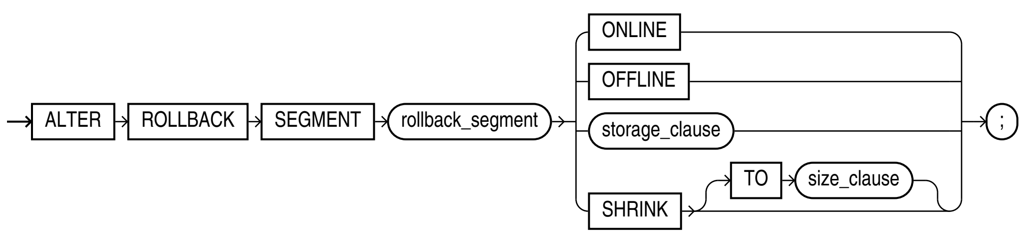
> **See Also:**
>
> - CREATE ROLLBACK SEGMENT for information on creating a rollback segment
> - *Oracle Database Reference* for information on the UNDO_MANAGEMENT parameter

Prerequisites

You must have the ALTER ROLLBACK SEGMENT system privilege.

Syntax

***alter_rollback_segment*::=**



(*storage_clause* , *size_clause*::=)

Semantics

rollback_segment

Specify the name of an existing rollback segment.

ONLINE

Specify ONLINE to bring the rollback segment online. When you create a rollback segment, it is initially offline and not available for transactions. This clause brings the rollback segment online, making it available for transactions by your instance. You can also bring a rollback segment online when you start your instance with the initialization parameter ROLLBACK_SEGMENTS.

> **See Also:**
>
> "Bringing a Rollback Segment Online: Example"

OFFLINE

Specify `OFFLINE` to take the rollback segment offline.

- If the rollback segment does not contain any information needed to roll back an active transaction, then Oracle Database takes it offline immediately.
- If the rollback segment does contain information for active transactions, then the database makes the rollback segment unavailable for future transactions and takes it offline after all the active transactions are committed or rolled back.

When the rollback segment is offline, it can be brought online by any instance.

To see whether a rollback segment is online or offline, query `STATUS` column of the data dictionary view `DBA_ROLLBACK_SEGS`. Online rollback segments have a value of `IN_USE`. Offline rollback segments have a value of `AVAILABLE`.

Restriction on Taking Rollback Segments Offline

You cannot take the `SYSTEM` rollback segment offline.

storage_clause

Use the *`storage_clause`* to change the storage characteristics of the rollback segment.

Restrictions on Rollback Segment Storage

You cannot change the value of `INITIAL` parameter. If the rollback segment is in a locally managed tablespace, then the only storage parameter you can change is `OPTIMAL`. If the rollback segment is in a dictionary-managed tablespace, then the only storage parameters you can change are `NEXT`, `MINEXTENTS`, `MAXEXTENTS` and `OPTIMAL`.

> **See Also:**
>
> *storage_clause* for syntax and additional information

SHRINK Clause

Specify `SHRINK` if you want Oracle Database to attempt to shrink the rollback segment to an optimal or specified size. The success and amount of shrinkage depend on the available free space in the rollback segment and how active transactions are holding space in the rollback segment.

If you do not specify `TO` *`size_clause`*, then the size defaults to the `OPTIMAL` value of the *`storage_clause`* of the `CREATE ROLLBACK SEGMENT` statement that created the rollback segment. If `OPTIMAL` was not specified, then the size defaults to the `MINEXTENTS` value of the *`storage_clause`* of the `CREATE ROLLBACK SEGMENT` statement.

Regardless of whether you specify `TO` *`size_clause`*:

- The value to which Oracle Database shrinks the rollback segment is valid for the execution of the statement. Thereafter, the size reverts to the `OPTIMAL` value of the `CREATE ROLLBACK SEGMENT` statement.
- The rollback segment cannot shrink to less than two extents.

To determine the actual size of a rollback segment after attempting to shrink it, query the BYTES, BLOCKS, and EXTENTS columns of the DBA_SEGMENTS view.

Restriction on Shrinking Rollback Segments

In an Oracle Real Application Clusters environment, you can shrink only rollback segments that are online to your instance.

> **See Also:**
>
> *size_clause* for information on that clause, and "Resizing a Rollback Segment: Example"

Examples

The following examples use the rbs_one rollback segment, which was created in "Creating a Rollback Segment: Example".

Bringing a Rollback Segment Online: Example

This statement brings the rollback segment rbs_one online:

```
ALTER ROLLBACK SEGMENT rbs_one ONLINE;
```

Resizing a Rollback Segment: Example

This statement shrinks the rollback segment rbs_one:

```
ALTER ROLLBACK SEGMENT rbs_one
   SHRINK TO 100M;
```

ALTER SEQUENCE

Purpose

Use the ALTER SEQUENCE statement to change the increment, minimum and maximum values, cached numbers, and behavior of an existing sequence. This statement affects only future sequence numbers.

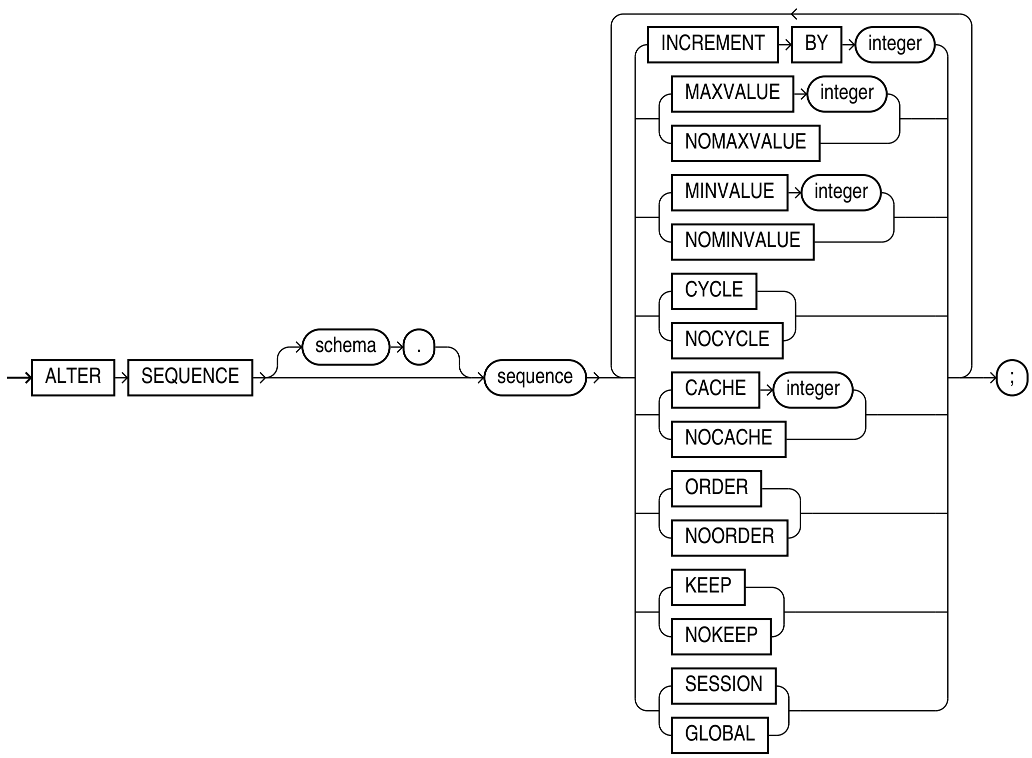
> **See Also:**
>
> CREATE SEQUENCE for additional information on sequences

Prerequisites

The sequence must be in your own schema, or you must have the ALTER object privilege on the sequence, or you must have the ALTER ANY SEQUENCE system privilege.

Syntax

***alter_sequence*::=**



Semantics

The keywords and parameters in this statement serve the same purposes they serve when you create a sequence.

- To restart the sequence at a different number, you must drop and re-create it.
- If you change the `INCREMENT BY` value before the first invocation of `NEXTVAL`, then some sequence numbers will be skipped. Therefore, if you want to retain the original `START WITH` value, you must drop the sequence and re-create it with the original `START WITH` value and the new `INCREMENT BY` value.
- If you alter the sequence by specifying the `KEEP` or `NOKEEP` clause between runtime and failover of a request, then the original value of `NEXTVAL` is not retained during replay for Application Continuity for that request.
- Oracle Database performs some validations. For example, a new `MAXVALUE` cannot be imposed that is less than the current sequence number.

> **See Also:**
>
> CREATE SEQUENCE for information on creating a sequence and DROP SEQUENCE for information on dropping and re-creating a sequence

Examples

Modifying a Sequence: Examples

This statement sets a new maximum value for the `customers_seq` sequence, which was created in "Creating a Sequence: Example":

```
ALTER SEQUENCE customers_seq
   MAXVALUE 1500;
```

This statement turns on `CYCLE` and `CACHE` for the `customers_seq` sequence:

```
ALTER SEQUENCE customers_seq
   CYCLE
   CACHE 5;
```

ALTER SESSION

Purpose

Use the `ALTER SESSION` statement to set or modify any of the conditions or parameters that affect your connection to the database. The statement stays in effect until you disconnect from the database.

Prerequisites

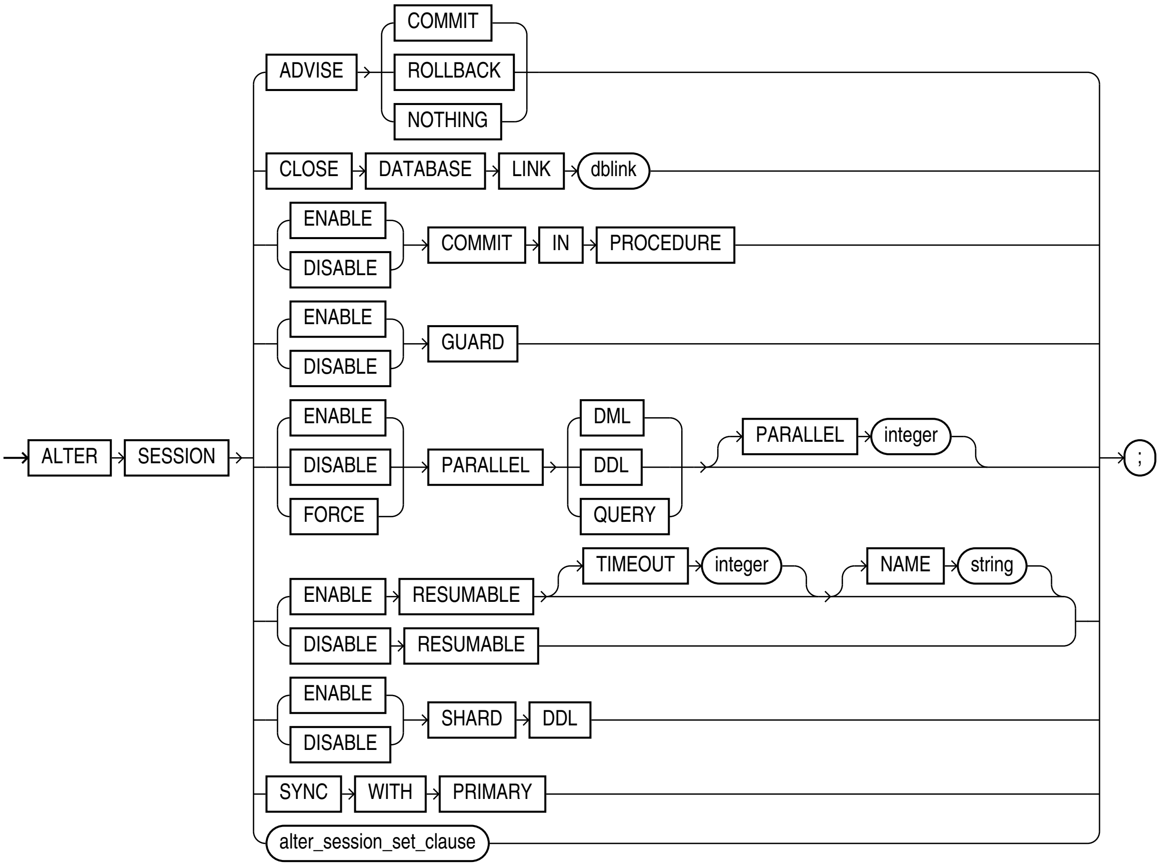
To enable and disable the SQL trace facility, you must have `ALTER SESSION` system privilege.

To enable or disable resumable space allocation, you must have the `RESUMABLE` system privilege.

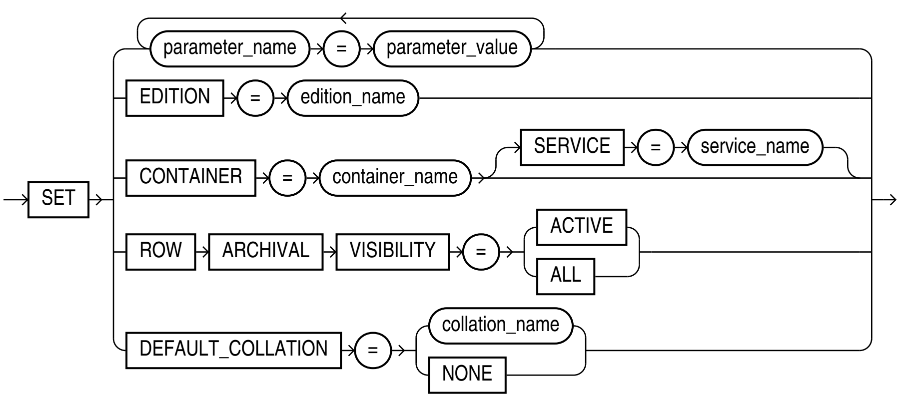
You do not need any privileges to perform the other operations of this statement unless otherwise indicated.

Syntax

***alter_session*::=**



***alter_session_set_clause*::=**



Semantics

ADVISE Clause

The ADVISE clause sends advice to a remote database to force a distributed transaction. The advice appears in the ADVICE column of the DBA_2PC_PENDING view on the remote database (the values are 'C' for COMMIT, 'R' for ROLLBACK, and ' ' for NOTHING). If the transaction becomes in doubt, then the administrator of that database can use this advice to decide whether to commit or roll back the transaction.

You can send different advice to different remote databases by issuing multiple ALTER SESSION statements with the ADVISE clause in a single transaction. Each such statement sends advice to the databases referenced in the following statements in the transaction until another such statement is issued.

> **See Also:**
>
> "Forcing a Distributed Transaction: Example"

CLOSE DATABASE LINK Clause

Specify CLOSE DATABASE LINK to close the database link *dblink*. When you issue a statement that uses a database link, Oracle Database creates a session for you on the remote database using that link. The connection remains open until you end your local session or until the number of database links for your session exceeds the value of the initialization parameter OPEN_LINKS. If you want to reduce the network overhead associated with keeping the link open, then use this clause to close the link explicitly if you do not plan to use it again in your session.

> **See Also:**
>
> Closing a Database Link: Example

ENABLE | DISABLE COMMIT IN PROCEDURE

Procedures and stored functions written in PL/SQL can issue COMMIT and ROLLBACK statements. If your application would be disrupted by a COMMIT or ROLLBACK statement not issued directly by the application itself, then specify DISABLE COMMIT IN PROCEDURE clause to prevent procedures and stored functions called during your session from issuing these statements.

You can subsequently allow procedures and stored functions to issue COMMIT and ROLLBACK statements in your session by issuing the ENABLE COMMIT IN PROCEDURE.

Some applications automatically prohibit COMMIT and ROLLBACK statements in procedures and stored functions. Refer to your application documentation for more information.

ENABLE | DISABLE GUARD

The *security_clause* of ALTER DATABASE lets you prevent anyone other than the SYS user from making any changes to data or database objects on the primary or standby database. This clause lets you override that setting for the current session.

> **See Also:**
>
> *security_clause* for more information on the GUARD setting

PARALLEL DML | DDL | QUERY

The PARALLEL parameter determines whether all subsequent DML, DDL, or query statements in the session will be considered for parallel execution. This clause enables you to override the degree of parallelism of tables during the current session without changing the tables themselves. Uncommitted transactions must either be committed or rolled back prior to executing this clause for DML.

> **See Also:**
>
> "Enabling Parallel DML: Example"

ENABLE Clause

Specify ENABLE to execute subsequent statements in the session in parallel. This is the default for DDL and query statements.

- DML: DML statements are executed in parallel mode if a parallel hint or a parallel clause is specified.
- DDL: DDL statements are executed in parallel mode if a parallel clause is specified.
- QUERY: Queries are executed in parallel mode if a parallel hint or a parallel clause is specified.

Restriction on the ENABLE clause

You cannot specify the optional PARALLEL *integer* with ENABLE.

DISABLE Clause

Specify DISABLE to execute subsequent statements in the session serially. This is the default for DML statements.

- DML: DML statements are executed serially.
- DDL: DDL statements are executed serially.
- QUERY: Queries are executed serially.

Restriction on the DISABLE clause

You cannot specify the optional PARALLEL *integer* with DISABLE.

FORCE Clause

FORCE forces parallel execution of subsequent statements in the session. If no parallel clause or hint is specified, then a default degree of parallelism is used. This clause overrides any *parallel_clause* specified in subsequent statements in the session but is overridden by a parallel hint.

- DML: Provided no parallel DML restrictions are violated, subsequent DML statements in the session are executed with the default degree of parallelism, unless a degree is specified in this clause.
- DDL: Subsequent DDL statements in the session are executed with the default degree of parallelism, unless a degree is specified in this clause. Resulting database objects will have associated with them the prevailing degree of parallelism.

  Specifying FORCE DDL automatically causes all tables created in this session to be created with a default level of parallelism. The effect is the same as if you had specified the *parallel_clause* (with the default degree) in the CREATE TABLE statement.
- QUERY: Subsequent queries are executed with the default degree of parallelism, unless a degree is specified in this clause.

PARALLEL *integer*

Specify an integer to explicitly specify a degree of parallelism:

- For FORCE DDL, the degree overrides any parallel clause in subsequent DDL statements.
- For FORCE DML and QUERY, the degree overrides the degree currently stored for the table in the data dictionary.
- A degree specified in a statement through a hint will override the degree being forced.

The following types of DML operations are not parallelized regardless of this clause:

- Operations on cluster tables
- Operations with embedded functions that either write or read database or package states
- Operations on tables with triggers that could fire
- Operations on tables or schema objects containing object types, or LONG or LOB data types

RESUMABLE Clauses

These clauses let you enable and disable resumable space allocation. This feature allows an operation to be suspended in the event of an out-of-space error condition and to resume automatically from the point of interruption when the error condition is fixed.

> **Note:**
>
> Resumable space allocation is fully supported for operations on locally managed tablespaces. Some restrictions apply if you are using dictionary-managed tablespaces. For information on these restrictions, refer to *Oracle Database Administrator's Guide*.

ENABLE RESUMABLE

This clause enables resumable space allocation for the session.

TIMEOUT

`TIMEOUT` lets you specify (in seconds) the time during which an operation can remain suspended while waiting for the error condition to be fixed. If the error condition is not fixed within the `TIMEOUT` period, then Oracle Database aborts the suspended operation.

NAME

`NAME` lets you specify a user-defined text string to help users identify the statements issued during the session while the session is in resumable mode. Oracle Database inserts the text string into the `USER_RESUMABLE` and `DBA_RESUMABLE` data dictionary views. If you do not specify `NAME`, then Oracle Database inserts the default string '`User` *`username`*(*`userid`*), `Session` *`sessionid`*, `Instance` *`instanceid`*'.

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views

DISABLE RESUMABLE

This clause disables resumable space allocation for the session.

SHARD DDL Clauses

These clauses are valid only if you are connected to a sharded database. They let you control whether DDLs issued in the session are issued against the shard catalog database and all shards, or against only the shard catalog database.

- If you specify `ENABLE SHARD DDL`, then DDLs issued in the session are issued against the shard catalog database and all shards. This mode is the default for the SDB user—a user that exists in the shard catalog database and in all shards.
- If you specify `DISABLE SHARD DDL`, then DDLs issued in the session are issued against only the shard catalog database. This mode is the default for a local user —a user that exists only in the shard catalog database.

SYNC WITH PRIMARY

Use this clause to synchronize redo apply on a physical standby database with the primary database. An `ALTER SESSION` statement with this clause blocks until redo apply has applied all redo data received by the standby at the time the statement is issued. This clause returns an error, and synchronization does not occur, if the redo transport state for the standby database is not `SYNCHRONIZED` or if redo apply is not active.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on this session parameter

alter_session_set_clause

Use the *alter_session_set_clause* to set initialization parameter values or to set an edition for the current session.

Initialization Parameters

You can set two types of parameters using this clause:

- Initialization parameters that are dynamic in the scope of the ALTER SESSION statement (listed in "Initialization Parameters and ALTER SESSION")
- Session parameters (listed in "Session Parameters and ALTER SESSION ")

You can set values for multiple parameters in the same *alter_session_set_clause*.

EDITION

Specify EDITION = *edition* to set the specified edition as the edition in the database session. You must have the USE object privilege on *edition*, *edition* must already have been created, and it must be USABLE.

When this statement is successful, the database discards PL/SQL package state corresponding to editionable packages but retains package state corresponding to packages that are not editionable.

You can also set the edition for the current session at startup with the EDITION parameter of the SQL*Plus CONNECT command. However, you cannot specify an ALTER SESSION SET EDITION statement in a recursive SQL or PL/SQL block.

You can determine the edition in use by the current session with the following query:

```
SELECT SYS_CONTEXT('USERENV', 'CURRENT_EDITION_NAME') FROM DUAL;
```

See Also:

CREATE EDITION for more information on editions and *Oracle Database PL/SQL Language Reference* for information on how editions are designated as USABLE

CONTAINER

Use this clause in a multitenant container database (CDB) to switch to the container specified by *container_name*.

To use this clause, you must be a common user with the SET CONTAINER privilege, either granted commonly or granted locally in *container_name*.

For *container_name*, specify one of the following:

- CDB$ROOT to switch to the root
- PDB$SEED to switch to the seed
- A pluggable database (PDB) name to switch to that PDB. You can view the names of the PDBs in a CDB by querying the DBA_PDBS view.

You can determine the container to which the current session is connected by using the SQL*Plus `SHOW CON_NAME` command or with the following SQL query:

```
SELECT SYS_CONTEXT('USERENV', 'CON_NAME') FROM DUAL;
```

SERVICE

By default, when you switch to a container, the session uses the default service for the container. Specify the `SERVICE` clause to use a different service for the container. For *`service_name`*, specify the name of the service you want to use.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on switching to a container

ROW ARCHIVAL VISIBILITY

Use this clause to configure row archival visibility for the session. This clause lets you implement In-Database Archiving, which allows you to designate table rows as active or archived. You can then perform queries on only the active rows within the table.

- If you specify `ACTIVE`, then the database will consider only active rows when performing queries on tables that are enabled for row archival. This is the default.
- If you specify `ALL`, then the database will consider all rows when performing queries on tables that are enabled for row archival.

This clause has no effect on queries on tables that are not enabled for row archival.

> **See Also:**
>
> - The `CREATE TABLE` ROW ARCHIVAL clause to learn how to enable a new table for row archival
> - The `ALTER TABLE` [NO] ROW ARCHIVAL clause to learn how to enable or disable an existing table for row archival
> - *Oracle Database VLDB and Partitioning Guide* for more information on In-Database Archiving

DEFAULT_COLLATION

Use this clause to set the default collation for the session.

- Use *`collation_name`* to specify the default collation for the session. You can specify the name of any valid named collation or pseudo-collation. This collation becomes the *effective schema default collation*. This collation is assigned to tables, views, and materialized views that are subsequently created in any schema for the duration of the session. The default collation for the session does not get propagated to any remote sessions connected to the current session using DB links.

- If you specify `NONE`, then there is no default collation for the session. In this case, the default collation for a particular schema becomes the *effective schema default collation* for that schema. That default collation is assigned to tables, views, and materialized views that are subsequently created in the schema for the duration of the session.

In either of the preceding cases, you can override the effective schema default collation and assign a default collation to a particular table, materialized view, or view by specifying the `DEFAULT COLLATION` clause of the `CREATE` or `ALTER` statement for the table, materialized view, or view.

The effective schema default collation also affects the DDL statements `CREATE FUNCTION`, `CREATE PACKAGE`, `CREATE PROCEDURE`, `CREATE TRIGGER`, and `CREATE TYPE`. Refer to *Oracle Database PL/SQL Language Reference* for more details on these statements.

You can query the default collation for a session with the following statement:

```
SELECT SYS_CONTEXT('USERENV', 'SESSION_DEFAULT_COLLATION') FROM DUAL;
```

You can specify the `SET DEFAULT_COLLATION` clause only if the `COMPATIBLE` initialization parameter is set to `12.2` or greater, and the `MAX_STRING_SIZE` initialization parameter is set to `EXTENDED`.

> **See Also:**
>
> The DEFAULT COLLATION Clause clause of `CREATE USER` for more information on the default collation of a schema

> **Note:**
>
> The effective schema default collation for a session should not be confused with the session parameter `NLS_SORT`. The effective schema default collation is used by DDL statements to decide the default data-bound collation of tables, views, and materialized views when they are created. The session parameter `NLS_SORT` points to a named collation that is used when Oracle executes a query, a DML statement, or PL/SQL code containing a SQL operation whose determined collation is a pseudo-collation, such as `USING_NLS_COMP` or `USING_NLS_SORT`. Refer to *Oracle Database Globalization Support Guide* for more information.

Initialization Parameters and ALTER SESSION

Some initialization parameter are dynamic in the scope of `ALTER SESSION`. When you set these parameters using `ALTER SESSION`, the value you set persists only for the duration of the current session.To determine whether a parameter can be altered using an `ALTER SESSION` statement, query the `ISSES_MODIFIABLE` column of the `V$PARAMETER` dynamic performance view.

> **Note:**
>
> Before changing the values of initialization parameters, refer to their full description in *Oracle Database Reference*.

A number of parameters that can be set using `ALTER SESSION` are not initialization parameters. You can set them only with `ALTER SESSION`, not in an initialization parameter file. Those session parameters are described in "Session Parameters and ALTER SESSION ".

Session Parameters and ALTER SESSION

The following parameters are session parameters only, not initialization parameters:

CONSTRAINT[S]

Syntax:

```
CONSTRAINT[S] = { IMMEDIATE | DEFERRED | DEFAULT }
```

The `CONSTRAINT[S]` parameter determines when conditions specified by a deferrable constraint are enforced.

- `IMMEDIATE` indicates that the conditions specified by the deferrable constraint are checked immediately after each DML statement. This setting is equivalent to issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement at the beginning of each transaction in your session.
- `DEFERRED` indicates that the conditions specified by the deferrable constraint are checked when the transaction is committed. This setting is equivalent to issuing the `SET CONSTRAINTS ALL DEFERRED` statement at the beginning of each transaction in your session.
- `DEFAULT` restores all constraints at the beginning of each transaction to their initial state of `DEFERRED` or `IMMEDIATE`.

CURRENT_SCHEMA

Syntax:

```
CURRENT_SCHEMA = schema
```

The `CURRENT_SCHEMA` parameter changes the current schema of the session to the specified schema. Subsequent unqualified references to schema objects during the session will resolve to objects in the specified schema. The setting persists for the duration of the session or until you issue another `ALTER SESSION SET CURRENT_SCHEMA` statement.

This setting offers a convenient way to perform operations on objects in a schema other than that of the current user without having to qualify the objects with the schema name. This setting changes the current schema, but it does not change the session user or the current user, nor does it give the session user any additional system or object privileges for the session.

ERROR_ON_OVERLAP_TIME

Syntax:

```
ERROR_ON_OVERLAP_TIME = {TRUE | FALSE}
```

The `ERROR_ON_OVERLAP_TIME` parameter determines how Oracle Database should handle an ambiguous boundary datetime value—a case in which it is not clear whether the datetime is in standard or daylight saving time.

- Specify `TRUE` to return an error for the ambiguous overlap timestamp.
- Specify `FALSE` to default the ambiguous overlap timestamp to the standard time. This is the default.

Refer to "Support for Daylight Saving Times " for more information on boundary datetime values.

FLAGGER

Syntax:

```
FLAGGER = { ENTRY | OFF }
```

The `FLAGGER` parameter specifies FIPS flagging (as specified in Federal Information Processing Standard 127-2), which causes an error message to be generated when a SQL statement issued is an extension of the Entry Level of SQL-92 (officially, ANSI X3.135-1992, a standard that is now superseded by SQL:2016). `FLAGGER` is a session parameter only, not an initialization parameter.

After flagging is set in a session, a subsequent `ALTER SESSION SET FLAGGER` statement will work, but generates the message, ORA-00097. This allows FIPS flagging to be altered without disconnecting the session. `OFF` turns off flagging.

> **See Also:**
>
> Oracle and Standard SQL, for more information about Oracle compliance with current ANSI SQL standards

INSTANCE

Syntax:

```
INSTANCE = integer
```

Setting the `INSTANCE` parameter lets you access another instance as if you were connected to your own instance. `INSTANCE` is a session parameter only, not an initialization parameter. In an Oracle Real Application Clusters (Oracle RAC) environment, each Oracle RAC instance retains static or dynamic ownership of disk space for optimal DML performance based on the setting of this parameter.

ISOLATION_LEVEL

Syntax:

```
ISOLATION_LEVEL = {SERIALIZABLE | READ COMMITTED}
```

The `ISOLATION_LEVEL` parameter specifies how transactions containing database modifications are handled. `ISOLATION_LEVEL` is a session parameter only, not an initialization parameter.

- `SERIALIZABLE` indicates that transactions in the session use the serializable transaction isolation mode as specified in the SQL standard. If a serializable transaction attempts to execute a DML statement that updates rows currently being updated by another uncommitted transaction at the start of the serializable transaction, then the DML statement fails. A serializable transaction can see its own updates.
- `READ COMMITTED` indicates that transactions in the session will use the default Oracle Database transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement will wait until the row locks are released.

> **Note:**
>
> Serializable transactions do not work with deferred segment creation or interval partitioning. Trying to insert data into an empty table with no segment created, or into a partition of an interval partitioned table that does not yet have a segment, causes an error.

STANDBY_MAX_DATA_DELAY

Syntax:

```
STANDBY_MAX_DATA_DELAY =  { integer | NONE }
```

In an Active Data Guard environment, this session parameter can be used to specify a session-specific apply lag tolerance, measured in seconds, for queries issued by non-administrative users to a physical standby database that is in real-time query mode. This capability allows queries to be safely offloaded from the primary database to a physical standby database, because it is possible to detect if the standby database has become unacceptably stale.

If `STANDBY_MAX_DATA_DELAY` is set to the default value of `NONE`, queries issued to a physical standby database will be executed regardless of the apply lag on that database.

If `STANDBY_MAX_DATA_DELAY` is set to a nonzero value, a query issued to a physical standby database will be executed only if the apply lag is less than or equal to `STANDBY_MAX_DATA_DELAY`. Otherwise, an `ORA-3172` error is returned to alert the client that the apply lag is too large.

If `STANDBY_MAX_DATA_DELAY` is set to `0`, a query issued to a physical standby database is guaranteed to return the exact same result as if the query were issued on the primary database, unless the standby database is lagging behind the primary database, in which case an ORA-3172 error is returned.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on Active Data Guard and using this session parameter

TIME_ZONE

Syntax:

```
TIME_ZONE =  '[+ | -] hh:mi'
             | LOCAL
             | DBTIMEZONE
             | 'time_zone_region'
```

The TIME_ZONE parameter specifies the default local time zone offset or region name for the current SQL session. TIME_ZONE is a session parameter only, not an initialization parameter. To determine the time zone of the current session, query the built-in function SESSIONTIMEZONE (see SESSIONTIMEZONE).

- Specify a format mask ('[+|-]*hh:mi*') indicating the hours and minutes before or after UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The valid range for *hh:mi* is -12:00 to +14:00.
- Specify LOCAL to set the default local time zone offset of the current SQL session to the original default local time zone offset that was established when the current SQL session was started.
- Specify DBTIMEZONE to set the current session time zone to match the value set for the database time zone. If you specify this setting, then the DBTIMEZONE function will return the database time zone as a UTC offset or a time zone region, depending on how the database time zone has been set.
- Specify a valid *time_zone_region*. To see a listing of valid time zone region names, query the TZNAME column of the V$TIMEZONE_NAMES dynamic performance view. If you specify this setting, then the SESSIONTIMEZONE function will return the region name.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for a complete listing of the time zone region names in both files

> **Note:**
>
> You can also set the default client session time zone using the `ORA_SDTZ` environment variable. Refer to *Oracle Database Globalization Support Guide* for more information on this variable.

USE_PRIVATE_OUTLINES

Syntax:

```
USE_PRIVATE_OUTLINES = { TRUE | FALSE | category_name }
```

The `USE_PRIVATE_OUTLINES` parameter lets you control the use of private outlines. When this parameter is enabled and an outlined SQL statement is issued, the optimizer retrieves the outline from the session private area rather than the public area used when `USE_STORED_OUTLINES` is enabled. If no outline exists in the session private area, then the optimizer will not use an outline to compile the statement. `USE_PRIVATE_OUTLINES` is not an initialization parameter.

- `TRUE` causes the optimizer to use private outlines stored in the `DEFAULT` category when compiling requests.
- `FALSE` specifies that the optimizer should not use stored private outlines. This is the default. If `USE_STORED_OUTLINES` is enabled, then the optimizer will use stored public outlines.
- *`category_name`* causes the optimizer to use outlines stored in the *`category_name`* category when compiling requests.

Restriction on USE_PRIVATE_OUTLINES

You cannot enable this parameter if `USE_STORED_OUTLINES` is enabled.

USE_STORED_OUTLINES

> **Note:**
>
> Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. Refer to *Oracle Database SQL Tuning Guide* for more information about SQL plan management.

Syntax:

```
USE_STORED_OUTLINES = { TRUE | FALSE | category_name }
```

The USE_STORED_OUTLINES parameter determines whether the optimizer will use stored public outlines to generate execution plans. USE_STORED_OUTLINES is not an initialization parameter.

- TRUE causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- FALSE specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

Restriction on USED_STORED_OUTLINES

You cannot enable this parameter if USE_PRIVATE_OUTLINES is enabled.

Examples

Enabling Parallel DML: Example

Issue the following statement to enable parallel DML mode for the current session:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Forcing a Distributed Transaction: Example

The following transaction inserts an employee record into the employees table on the database identified by the database link remote and deletes an employee record from the employees table on the database identified by local:

```
ALTER SESSION
   ADVISE COMMIT;

INSERT INTO employees@remote
   VALUES (8002, 'Juan', 'Fernandez', 'juanf@example.com', NULL,
   TO_DATE('04-OCT-1992', 'DD-MON-YYYY'), 'SA_CLERK', 3000,
   NULL, 121, 20);

ALTER SESSION
   ADVISE ROLLBACK;

DELETE FROM employees@local
   WHERE employee_id = 8002;

COMMIT;
```

This transaction has two ALTER SESSION statements with the ADVISE clause. If the transaction becomes in doubt, then remote is sent the advice 'COMMIT' by virtue of the first ALTER SESSION statement and local is sent the advice 'ROLLBACK' by virtue of the second statement.

Closing a Database Link: Example

This statement updates the jobs table on the local database using a database link, commits the transaction, and explicitly closes the database link:

```
UPDATE jobs@local SET min_salary = 3000
   WHERE job_id = 'SH_CLERK';

COMMIT;

ALTER SESSION
   CLOSE DATABASE LINK local;
```

Changing the Date Format Dynamically: Example

The following statement dynamically changes the default date format for your session to 'YYYY MM DD-HH24:MI:SS':

```
ALTER SESSION
   SET NLS_DATE_FORMAT = 'YYYY MM DD HH24:MI:SS';
```

Oracle Database uses the new default date format:

```
SELECT TO_CHAR(SYSDATE) Today
   FROM DUAL;

TODAY
-------------------
2001 04 12 12:30:38
```

Changing the Date Language Dynamically: Example

The following statement changes the language for date format elements to French:

```
ALTER SESSION
   SET NLS_DATE_LANGUAGE = French;

SELECT TO_CHAR(SYSDATE, 'Day DD Month YYYY') Today
   FROM DUAL;

TODAY
---------------------------
Jeudi    12 Avril     2001
```

Changing the ISO Currency: Example

The following statement dynamically changes the ISO currency symbol to the ISO currency symbol for the territory America:

```
ALTER SESSION
   SET NLS_ISO_CURRENCY = America;

SELECT TO_CHAR( SUM(salary), 'C999G999D99') Total
   FROM employees;

TOTAL
------------------
     USD694,900.00
```

Changing the Decimal Character and Group Separator: Example

The following statement dynamically changes the decimal character to comma (,) and the group separator to period (.):

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ',.' ;
```

Oracle Database returns these new characters when you use their number format elements:

```
ALTER SESSION SET NLS_CURRENCY = 'FF';

SELECT TO_CHAR( SUM(salary), 'L999G999D99') Total FROM employees;

TOTAL
```

```
---------------------
         FF694.900,00
```

Changing the NLS Currency: Example

The following statement dynamically changes the local currency symbol to 'DM':

```
ALTER SESSION
   SET NLS_CURRENCY = 'DM';

SELECT TO_CHAR( SUM(salary), 'L999G999D99') Total
   FROM employees;

TOTAL
---------------------
         DM694.900,00
```

Changing the NLS Language: Example

The following statement dynamically changes to French the language in which error messages are displayed:

```
ALTER SESSION
   SET NLS_LANGUAGE = FRENCH;

Session modifiee.

SELECT * FROM DMP;

ORA-00942: Table ou vue inexistante
```

Changing the Linguistic Sort Sequence: Example

The following statement dynamically changes the linguistic sort sequence to Spanish:

```
ALTER SESSION
   SET NLS_SORT = XSpanish;
```

Oracle Database sorts character values based on their position in the Spanish linguistic sort sequence.

Enabling Query Rewrite: Example

This statement enables query rewrite in the current session for all materialized views that have not been explicitly disabled:

```
ALTER SESSION
  SET QUERY_REWRITE_ENABLED = TRUE;
```

12

SQL Statements: ALTER SYNONYM to COMMENT

This chapter contains the following SQL statements:

- ALTER SYNONYM
- ALTER SYSTEM
- ALTER TABLE
- ALTER TABLESPACE
- ALTER TABLESPACE SET
- ALTER TRIGGER
- ALTER TYPE
- ALTER USER
- ALTER VIEW
- ANALYZE
- ASSOCIATE STATISTICS
- AUDIT (Traditional Auditing)
- AUDIT (Unified Auditing)
- CALL
- COMMENT

ALTER SYNONYM

Purpose

Use the `ALTER SYNONYM` statement to modify an existing synonym.

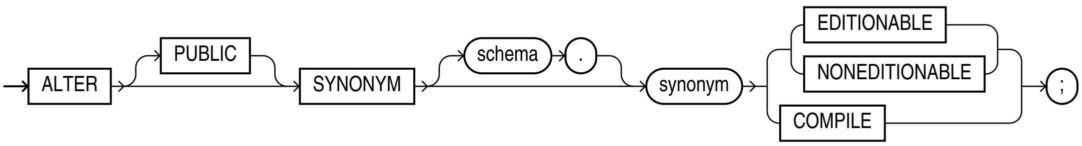
Prerequisites

To modify a private synonym in another user's schema, you must have the `CREATE ANY SYNONYM` and `DROP ANY SYNONYM` system privileges.

To modify a `PUBLIC` synonym, you must have the `CREATE PUBLIC SYNONYM` and `DROP PUBLIC SYNONYM` system privileges.

Syntax

***alter_synonym*::=**



Semantics

PUBLIC

Specify `PUBLIC` if *`synonym`* is a public synonym. You cannot use this clause to change a public synonym to a private synonym, or vice versa.

schema

Specify the schema containing the synonym. If you omit *`schema`*, then Oracle Database assumes the synonym is in your own schema.

synonym

Specify the name of the synonym to be altered.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the synonym becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `SYNONYM` in *`schema`*. The default is `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

Restriction on EDITIONABLE | NONEDITIONABLE

You cannot specify these clauses for a public synonym because editioning is always enabled for the object type `SYNONYM` in the `PUBLIC` schema.

COMPILE

Use this clause to compile *`synonym`*. A synonym places a dependency on its target object and becomes invalid if the target object is changed or dropped. When you compile an invalid synonym, it becomes valid again.

> **Note:**
>
> You can determine if a synonym is valid or invalid by querying the `STATUS` column of the `ALL_`, `DBA_`, and `USER_OBJECTS` data dictionary views.

Examples

The following examples modify synonyms that were created in the `CREATE SYNONYM` "Examples".

The following statement compiles synonym `offices`:

```
ALTER SYNONYM offices COMPILE;
```

The following statement compiles public synonym `emp_table`:

```
ALTER PUBLIC SYNONYM emp_table COMPILE;
```

The following statement causes synonym `offices` to remain a noneditioned object if editioning is later enabled for schema object type `SYNONYM` in the schema that contains the synonym `offices`:

```
ALTER SYNONYM offices NONEDITIONABLE;
```

ALTER SYSTEM

Purpose

Use the `ALTER SYSTEM` statement to dynamically alter your Oracle Database instance. The settings stay in effect as long as the database is mounted.

When you use the `ALTER SYSTEM` statement in a multitenant container database (CDB), you can specify some clauses to alter the CDB as a whole and other clauses to alter a specific pluggable database (PDB).

> **See Also:**
>
> *Oracle Database Administrator's Guide* for complete information on using the `ALTER SYSTEM` statement in a CDB

Prerequisites

To specify the `RELOCATE CLIENT` clause, you must be authenticated `AS SYSASM`.

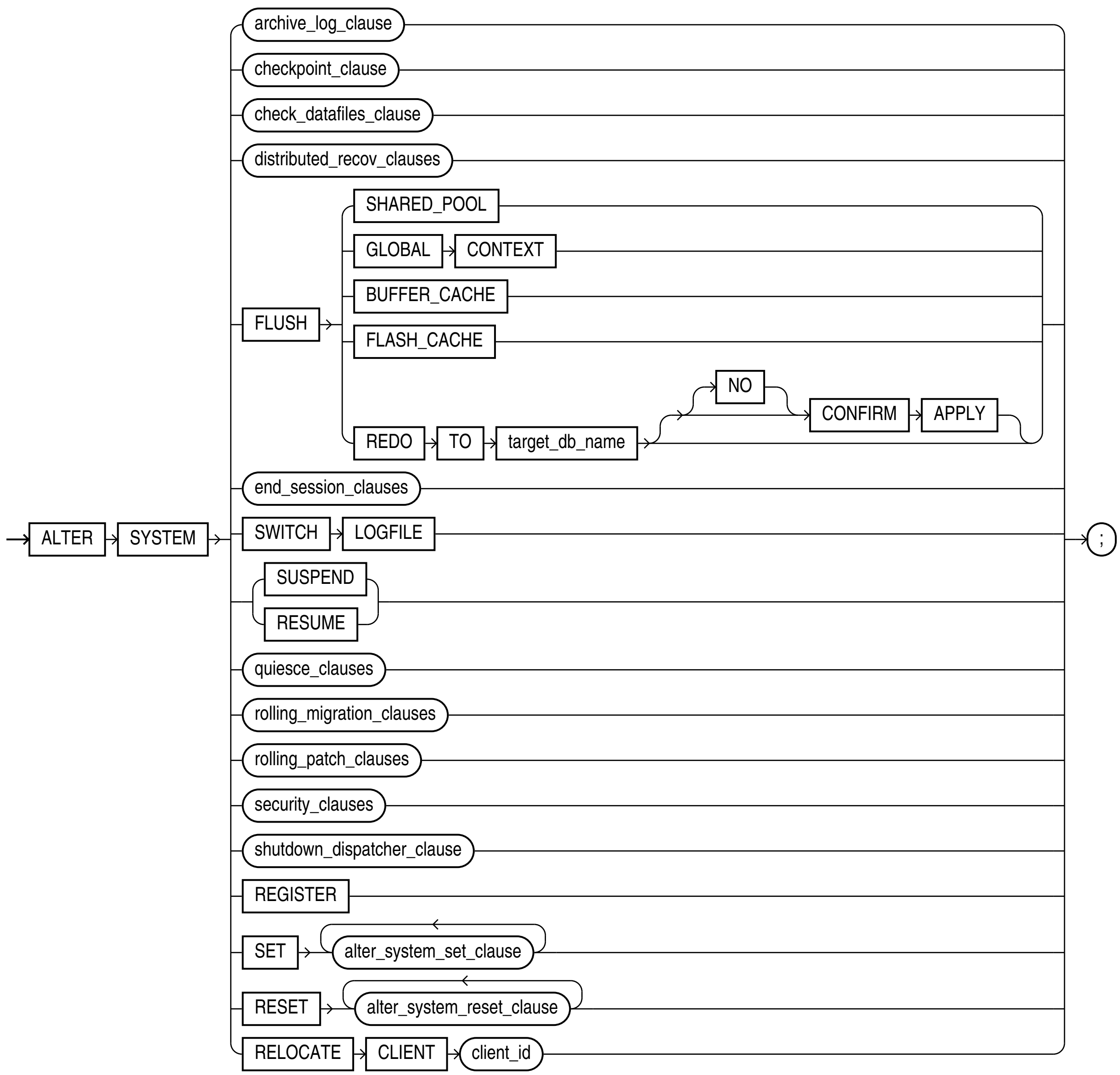
To specify all other clauses, you must have the `ALTER SYSTEM` system privilege.

If you are connected to a CDB:

- To alter the CDB as a whole, the current container must be the root and you must have the commonly granted `ALTER SYSTEM` privilege.
- To alter a PDB, the current container must be the PDB and you must have the `ALTER SYSTEM` privilege, either granted commonly or granted locally in the PDB.

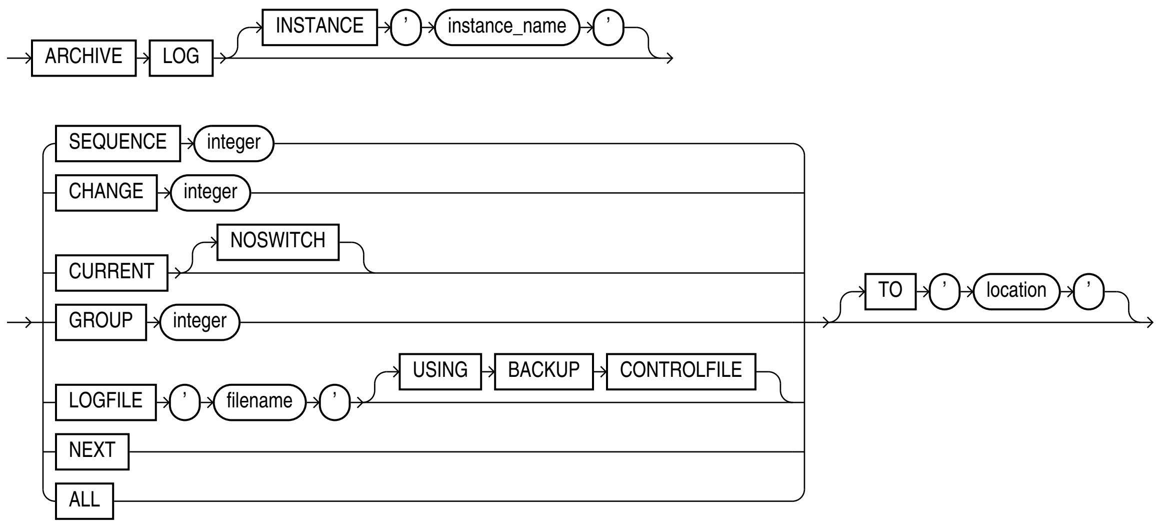
Syntax

***alter_system*::=**

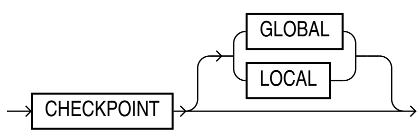


(*archive_log_clause*::=, *checkpoint_clause*::=, *check_datafiles_clause*::=, *distributed_recov_clauses*::=, *end_session_clauses*::=, *quiesce_clauses*::=, *rolling_migration_clauses*::=, *rolling_patch_clauses*::=, *security_clauses*::=, *shutdown_dispatcher_clause*::=, *alter_system_set_clause*::=, *alter_system_reset_clause*::=)

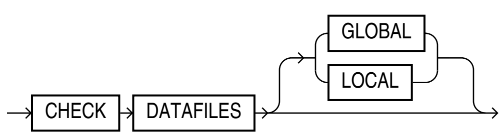
***archive_log_clause*::=**



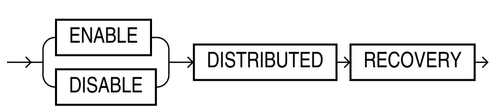
***checkpoint_clause*::=**



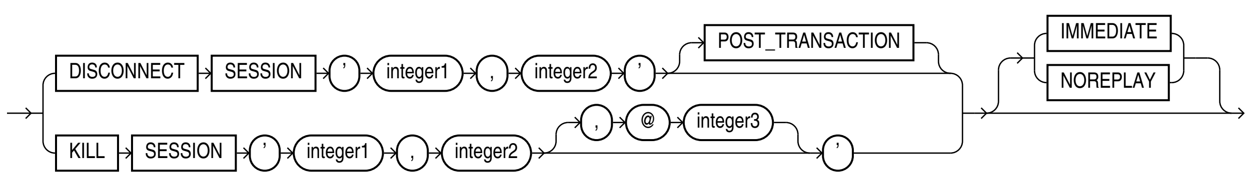
***check_datafiles_clause*::=**



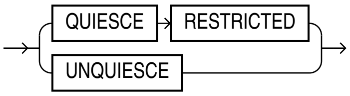
***distributed_recov_clauses*::=**



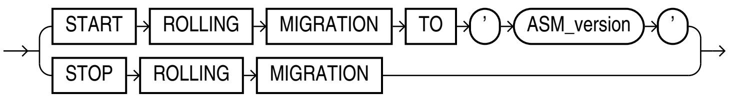
***end_session_clauses*::=**



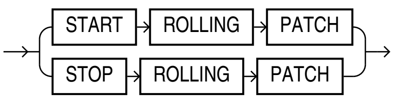
***quiesce_clauses*::=**



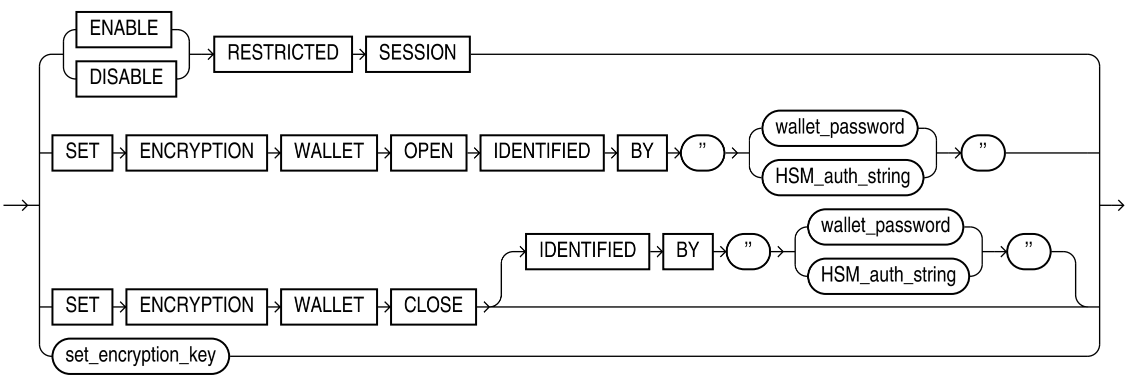
***rolling_migration_clauses*::=**



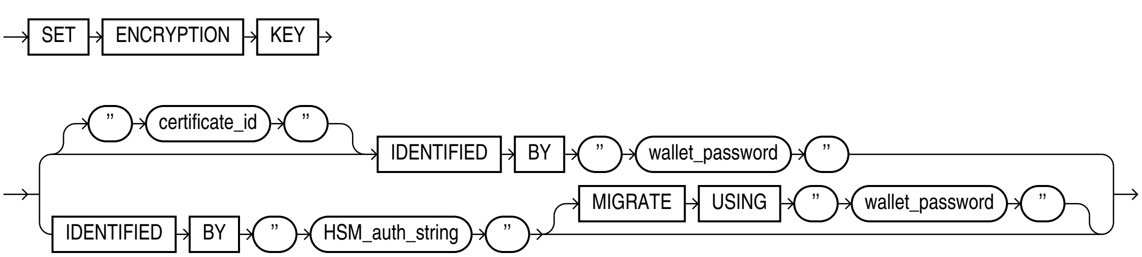
***rolling_patch_clauses*::=**



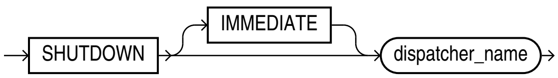
***security_clauses*::=**



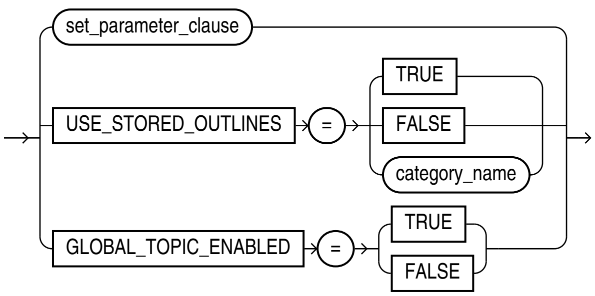
***set_encryption_key*::=**



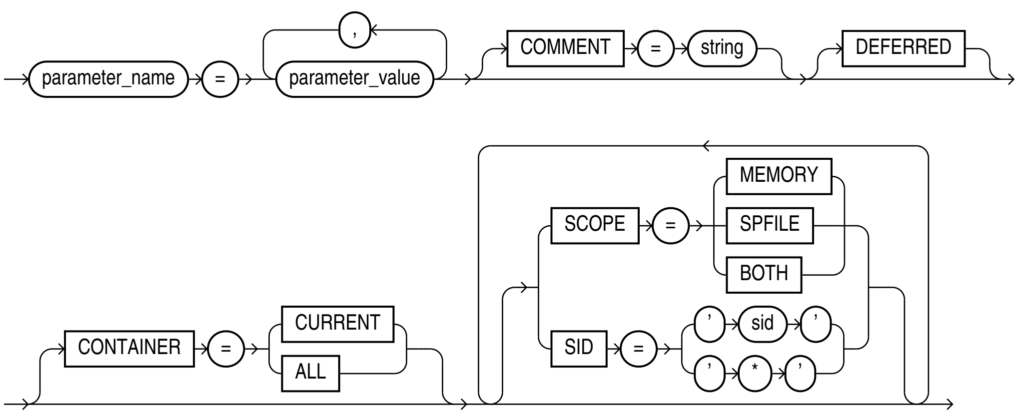
***shutdown_dispatcher_clause*::=**



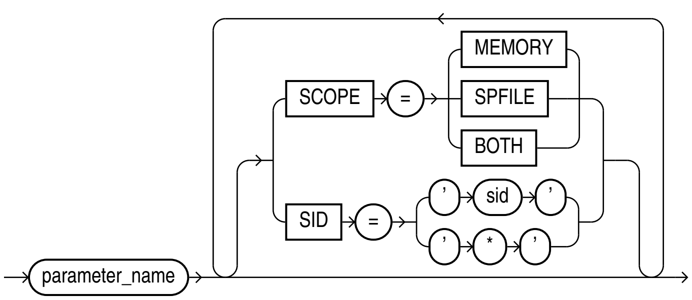
***alter_system_set_clause*::=**



***set_parameter_clause*::=**



***alter_system_reset_clause*::=**



Semantics

archive_log_clause

The *archive_log_clause* manually archives redo log files or enables or disables automatic archiving. To use this clause, your instance must have the database mounted. The database can be either open or closed unless otherwise noted.

INSTANCE Clause

This clause is relevant only if you are using Oracle Real Application Clusters (Oracle RAC). Specify the name of the instance for which you want the redo log file group to be archived. The instance name is a string of up to 80 characters. Oracle Database automatically determines the thread that is mapped to the specified instance and archives the corresponding redo log file group. If no thread is mapped to the specified instance, then Oracle Database returns an error.

SEQUENCE Clause

Specify `SEQUENCE` to manually archive the online redo log file group identified by the log sequence number *integer* in the specified thread. If you omit the `THREAD` parameter, then Oracle Database archives the specified group from the thread assigned to your instance.

CHANGE Clause

Specify `CHANGE` to manually archive the online redo log file group containing the redo log entry with the system change number (SCN) specified by *integer* in the specified thread. If the SCN is in the current redo log file group, then Oracle Database performs a log switch. If you omit the `THREAD` parameter, then Oracle Database archives the groups containing this SCN from all enabled threads.

You can use this clause only when your instance has the database open.

CURRENT Clause

Specify `CURRENT` to manually archive the current redo log file group of the specified thread, forcing a log switch. If you omit the `THREAD` parameter, then Oracle Database archives all redo log file groups from all enabled threads, including logs previous to current logs. You can specify `CURRENT` only when the database is open.

NOSWITCH

Specify `NOSWITCH` if you want to manually archive the current redo log file group without forcing a log switch. This setting is used primarily with standby databases to prevent data divergence when the primary database shuts down. Divergence implies the possibility of data loss in case of primary database failure.

You can use the `NOSWITCH` clause only when your instance has the database mounted but not open. If the database is open, then this operation closes the database automatically. You must then manually shut down the database before you can reopen it.

GROUP Clause

Specify `GROUP` to manually archive the online redo log file group with the `GROUP` value specified by *integer*. You can determine the `GROUP` value for a redo log file group by querying the dynamic performance view `V$LOG`. If you specify both the `THREAD` and

GROUP parameters, then the specified redo log file group must be in the specified thread.

LOGFILE Clause

Specify LOGFILE to manually archive the online redo log file group containing the redo log file member identified by '*filename*'. If you specify both the THREAD and LOGFILE parameters, then the specified redo log file group must be in the specified thread.

If the database was mounted with a backup control file, then specify USING BACKUP CONTROLFILE to permit archiving of all online logfiles, including the current logfile.

Restriction on the LOGFILE clause

You must archive redo log file groups in the order in which they are filled. If you specify a redo log file group for archiving with the LOGFILE parameter, and earlier redo log file groups are not yet archived, then Oracle Database returns an error.

NEXT Clause

Specify NEXT to manually archive the next online redo log file group from the specified thread that is full but has not yet been archived. If you omit the THREAD parameter, then Oracle Database archives the earliest unarchived redo log file group from any enabled thread.

ALL Clause

Specify ALL to manually archive all online redo log file groups from the specified thread that are full but have not been archived. If you omit the THREAD parameter, then Oracle Database archives all full unarchived redo log file groups from all enabled threads.

TO *location* Clause

Specify TO '*location*' to indicate the primary location to which the redo log file groups are archived. The value of this parameter must be a fully specified file location following the conventions of your operating system. If you omit this parameter, then Oracle Database archives the redo log file group to the location specified by the initialization parameters LOG_ARCHIVE_DEST or LOG_ARCHIVE_DEST_*n*.

checkpoint_clause

Specify CHECKPOINT to explicitly force Oracle Database to perform a checkpoint, ensuring that all changes made by committed transactions are written to data files on disk. You can specify this clause only when your instance has the database open. Oracle Database does not return control to you until the checkpoint is complete.

GLOBAL

In an Oracle Real Application Clusters (Oracle RAC) environment, this setting causes Oracle Database to perform a checkpoint for all instances that have opened the database. This is the default.

LOCAL

In an Oracle RAC environment, this setting causes Oracle Database to perform a checkpoint only for the thread of redo log file groups for the instance from which you issue the statement.

> **See Also:**
>
> "Forcing a Checkpoint: Example"

check_datafiles_clause

In a distributed database system, such as an Oracle RAC environment, this clause updates an instance's SGA from the database control file to reflect information on all online data files.

- Specify `GLOBAL` to perform this synchronization for all instances that have opened the database. This is the default.
- Specify `LOCAL` to perform this synchronization only for the local instance.

Your instance should have the database open.

distributed_recov_clauses

The `DISTRIBUTED RECOVERY` clause lets you enable or disable distributed recovery. To use this clause, your instance must have the database open.

ENABLE

Specify `ENABLE` to enable distributed recovery. In a single-process environment, you must use this clause to initiate distributed recovery.

You may need to issue the `ENABLE DISTRIBUTED RECOVERY` statement more than once to recover an in-doubt transaction if the remote node involved in the transaction is not accessible. In-doubt transactions appear in the data dictionary view `DBA_2PC_PENDING`.

> **See Also:**
>
> "Enabling Distributed Recovery: Example"

DISABLE

Specify `DISABLE` to disable distributed recovery.

FLUSH SHARED_POOL Clause

The `FLUSH SHARED_POOL` clause lets you clear data from the shared pool in the system global area (SGA). The shared pool stores:

- Cached data dictionary information and
- Shared SQL and PL/SQL areas for SQL statements, stored procedures, functions, packages, and triggers.

This statement does not clear global application context information, nor does it clear shared SQL and PL/SQL areas for items that are currently being executed. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

> **See Also:**
>
> "Clearing the Shared Pool: Example"

FLUSH GLOBAL CONTEXT Clause

The `FLUSH GLOBAL CONTEXT` clause lets you flush all global application context information from the shared pool in the system global area (SGA). You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

FLUSH BUFFER_CACHE Clause

The `FLUSH BUFFER_CACHE` clause lets you clear all data from the buffer cache in the system global area (SGA), including the `KEEP`, `RECYCLE`, and `DEFAULT` buffer pools.

> **Note:**
>
> This clause is intended for use only on a test database. Do not use this clause on a production database, because as a result of this statement, subsequent queries will have no hits, only misses.

This clause is useful if you need to measure the performance of rewritten queries or a suite of queries from identical starting points.

FLUSH FLASH_CACHE Clause

Use the `FLUSH FLASH_CACHE` clause to flush the Database Smart Flash Cache. This clause can be useful if you need to measure the performance of rewritten queries or a suite of queries from identical starting points, or if there might be corruption in the cache.

FLUSH REDO Clause

Use the `FLUSH REDO` clause to flush redo data from a primary database to a standby database and to optionally wait for the flushed redo data to be applied to a physical or logical standby database.

This clause can allow a failover to be performed on the target standby database without data loss, even if the primary database is not in a zero data loss data protection mode, provided that all redo data that has been generated by the primary database can be flushed to the standby database.

The `FLUSH REDO` clause must be issued on a mounted, but not open, primary database.

target_db_name

For *`target_db_name`*, specify the `DB_UNIQUE_NAME` of the standby database that is to receive the redo data flushed from the primary database.

The value of the `LOG_ARCHIVE_DEST_n` database initialization parameter that corresponds to the target standby database must contain the `DB_UNIQUE_NAME` attribute, and the value of that attribute must match the `DB_UNIQUE_NAME` of the target standby database.

NO CONFIRM APPLY

If you specify this clause, then the `ALTER SYSTEM` statement will not complete until the standby database has received all of the flushed redo data. You must specify this clause if the target standby database is a snapshot standby database.

CONFIRM APPLY

If you specify this clause, then the `ALTER SYSTEM` statement will not complete until the target standby database has received and applied all flushed redo data. This is the default behavior unless you specify `NO CONFIRM APPLY`. You cannot specify this clause if the target standby database is a snapshot standby database.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information about the `FLUSH REDO` clause and failovers

end_session_clauses

The *`end_session_clauses`* give you several ways to end the current session.

DISCONNECT SESSION Clause

Use the `DISCONNECT SESSION` clause to disconnect the current session by destroying the dedicated server process (or virtual circuit if the connection was made by way of a Shared Server). To use this clause, your instance must have the database open. You must identify the session with both of the following values from the `V$SESSION` view:

- For *`integer1`*, specify the value of the `SID` column.
- For *`integer2`*, specify the value of the `SERIAL#` column.

If system parameters are appropriately configured, then application failover will take effect.

- The `POST_TRANSACTION` setting allows ongoing transactions to complete before the session is disconnected. If the session has no ongoing transactions, then this clause has the same effect described for as `KILL SESSION`.
- The `IMMEDIATE` setting disconnects the session and recovers the entire session state immediately, without waiting for ongoing transactions to complete.
  - If you also specify `POST_TRANSACTION` and the session has ongoing transactions, then the `IMMEDIATE` keyword is ignored.
  - If you do not specify `POST_TRANSACTION`, or you specify `POST_TRANSACTION` but the session has no ongoing transactions, then this clause has the same effect as described for `KILL SESSION IMMEDIATE`.

  > **See Also:**
  >
  > "Disconnecting a Session: Example"

KILL SESSION Clause

The `KILL SESSION` clause lets you mark a session as terminated, roll back ongoing transactions, release all session locks, and partially recover session resources. To use this clause, your instance must have the database open. Your session and the session to be terminated must be on the same instance unless you specify *`integer3`*.You must identify the session with the following values from the `V$SESSION` view:

- For *`integer1`*, specify the value of the `SID` column.
- For *`integer2`*, specify the value of the `SERIAL#` column.
- For the optional *`integer3`*, specify the ID of the instance where the target session to be killed exists. You can find the instance ID by querying the GV$ tables.

If the session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, then Oracle Database waits for this activity to complete, marks the session as terminated, and then returns control to you. If the waiting lasts a minute, then Oracle Database marks the session to be terminated and returns control to you with a message that the session is marked to be terminated. The `PMON` background process then marks the session as terminated when the activity is complete.

Whether or not the session has an ongoing transaction, Oracle Database does not recover the entire session state until the session user issues a request to the session and receives a message that the session has been terminated.

> **See Also:**
>
> "Terminating a Session: Example"

IMMEDIATE

Specify `IMMEDIATE` to instruct Oracle Database to roll back ongoing transactions, release all session locks, recover the entire session state, and return control to you immediately.

NOREPLAY

This clause is valid if you are using Application Continuity. When connected to a service with Application Continuity enabled (that is, `FAILOVER_TYPE = TRANSACTION`), the session is recovered after the session fails or is killed. If you do not want to recover a session after it is terminated, then specify `NOREPLAY`.

SWITCH LOGFILE Clause

The `SWITCH LOGFILE` clause lets you explicitly force Oracle Database to begin writing to a new redo log file group, regardless of whether the files in the current redo log file group are full. When you force a log switch, Oracle Database begins to perform a checkpoint but returns control to you immediately rather than when the checkpoint is complete. To use this clause, your instance must have the database open.

> **See Also:**
>
> "Forcing a Log Switch: Example"

SUSPEND | RESUME

The SUSPEND clause lets you suspend all I/O (data file, control file, and file header) as well as queries, in all instances, enabling you to make copies of the database without having to handle ongoing transactions.

Restrictions on SUSPEND and RESUME

SUSPEND and RESUME are subject to the following restrictions:

- Do not use this clause unless you have put the database tablespaces in hot backup mode.
- Do not terminate the session that issued the ALTER SYSTEM SUSPEND statement. An attempt to reconnect while the system is suspended may fail because of recursive SQL that is running during the SYS login.
- If you start a new instance while the system is suspended, then that new instance will not be suspended.

The RESUME clause lets you make the database available once again for queries and I/O.

quiesce_clauses

Use the QUIESCE RESTRICTED and UNQUIESCE clauses to put the database in and take it out of the **quiesced state**. This state enables database administrators to perform administrative operations that cannot be safely performed in the presence of concurrent transactions, queries, or PL/SQL operations.

> **Note:**
>
> The QUIESCE RESTRICTED clause is valid only if the Database Resource Manager is installed and only if the Resource Manager has been on continuously since database startup in any instances that have opened the database.

If multiple QUIESCE RESTRICTED or UNQUIESCE statements issue at the same time from different sessions or instances, then all but one will receive an error.

QUIESCE RESTRICTED

Specify QUIESCE RESTRICTED to put the database in the quiesced state. For all instances with the database open, this clause has the following effect:

- Oracle Database instructs the Database Resource Manager in all instances to prevent all inactive sessions (other than SYS and SYSTEM) from becoming active. No user other than SYS and SYSTEM can start a new transaction, a new query, a new fetch, or a new PL/SQL operation.
- Oracle Database waits for all existing transactions in all instances that were initiated by a user other than SYS or SYSTEM to finish (either commit or abort). Oracle Database also waits for all running queries, fetches, and PL/SQL procedures in all instances that were initiated by users other than SYS or SYSTEM and that are not inside transactions to finish. If a query is carried out by multiple

successive OCI fetches, then Oracle Database does not wait for all fetches to finish. It waits for the current fetch to finish and then blocks the next fetch. Oracle Database also waits for all sessions (other than those of `SYS` or `SYSTEM`) that hold any shared resources (such as enqueues) to release those resources. After all these operations finish, Oracle Database places the database into quiesced state and finishes executing the `QUIESCE RESTRICTED` statement.

- If an instance is running in shared server mode, then Oracle Database instructs the Database Resource Manager to block logins (other than `SYS` or `SYSTEM`) on that instance. If an instance is running in non-shared-server mode, then Oracle Database does not impose any restrictions on user logins in that instance.

During the quiesced state, you cannot change the Resource Manager plan in any instance.

UNQUIESCE

Specify `UNQUIESCE` to take the database out of quiesced state. Doing so permits transactions, queries, fetches, and PL/SQL procedures that were initiated by users other than `SYS` or `SYSTEM` to be undertaken once again. The `UNQUIESCE` statement does not have to originate in the same session that issued the `QUIESCE RESTRICTED` statement.

rolling_migration_clauses

Use these clauses in a clustered Oracle Automatic Storage Management (Oracle ASM) environment to migrate one node at a time to a different Oracle ASM version without affecting the overall availability of the Oracle ASM cluster or the database clusters using Oracle ASM for storage.

START ROLLING MIGRATION

When starting rolling upgrade, for *`ASM_version`*, you must specify the following string:

```
'<version_num>, <release_num>, <update_num>,<port_release_num>,<port_update_num>'
```

`ASM_version` must be equal to or greater than 11.1.0.0.0. The surrounding single quotation marks are required. Oracle ASM first verifies that the current release is compatible for migration to the specified release, and then goes into limited functionality mode. Oracle ASM then determines whether any rebalance operations are under way anywhere in the cluster. If there are any such operations, then the statement fails and must be reissued after the rebalance operations are complete.

Rolling upgrade mode is a cluster-wide In-Memory persistent state. The cluster continues to be in this state until there is at least one Oracle ASM instance running in the cluster. Any new instance joining the cluster switches to migration mode immediately upon startup. If all the instances in the cluster terminate, then subsequent startup of any Oracle ASM instance will not be in rolling upgrade mode until you reissue this statement to restart rolling upgrade of the Oracle ASM instances.

STOP ROLLING MIGRATION

Use this clause to stop rolling upgrade and bring the cluster back into normal operation. Specify this clause only after all instances in the cluster have migrated to the same software version. The statement will fail if the cluster is not in rolling upgrade mode.

When you specify this clause, the Oracle ASM instance validates that all the members of the cluster are at the same software version, takes the instance out of rolling upgrade mode, and returns to full functionality of the Oracle ASM cluster. If any rebalance operations are pending because disks have gone offline, then those operations are restarted if the `ASM_POWER_LIMIT` parameter would not be violated by such a restart.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about rolling upgrade

rolling_patch_clauses

Use these clauses in a clustered Oracle Automatic Storage Management (Oracle ASM) environment to update one node at a time to the latest patch level without affecting the overall availability of the Oracle ASM cluster or the database clusters using Oracle ASM for storage.

START ROLLING PATCH

Use this clause to start the rolling patch operation. Oracle ASM first verifies that all live nodes in the cluster are at the same version, and then goes into rolling patch mode, which is a cluster-wide In-Memory persistent state. The cluster continues to be in this state until all live nodes have been patched to the latest patch level.

Any nodes that are down during this operation are not patched. This does not affect the success of the rolling patch operation. However, you must patch these nodes before they are started. Otherwise, they will not be allowed to join the cluster.

STOP ROLLING PATCH

use this clause to stop the rolling patch operation and bring the cluster back into normal operation. Specify this clause only after all live nodes in the cluster have been patched to the latest patch level. The statement will fail if the cluster is not in rolling patch mode.

When you specify this clause, the Oracle ASM instance validates that all members of the cluster are at the same patch level, takes the instance out of rolling patch mode, and returns full functionality of the Oracle ASM cluster. If any members of the cluster are not at the latest patch level, then this operation fails and the cluster goes into limited functionality mode.

The following queries display information about rolling patches. In order to run these queries, you must be connected to the Oracle ASM instance in the Grid home, and the Grid Infrastructure home must be configured with the Oracle Clusterware option for an Oracle RAC environment.

- You can determine whether a cluster is in rolling patch mode with the following query:

  ```
  SELECT SYS_CONTEXT('SYS_CLUSTER_PROPERTIES', 'CLUSTER_STATE') FROM DUAL;
  ```

- You can determine the patch level of a cluster with the following query:

  ```
  SELECT SYS_CONTEXT('SYS_CLUSTER_PROPERTIES', 'CURRENT_PATCHLVL') FROM DUAL;
  ```

- You can display a list of patches applied on the Oracle ASM instance, by querying the `V$PATCHES` dynamic performance view. Refer to *Oracle Database Reference* for more information.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about rolling patches

security_clauses

The `security_clauses` let you control access to the instance. They also allow you to enable or disable access to the encrypted data in the instance.

RESTRICTED SESSION

The `RESTRICTED SESSION` clause lets you restrict logon to Oracle Database. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

- Specify `ENABLE` to allow only users with `RESTRICTED SESSION` system privilege to log on to Oracle Database. Existing sessions are not terminated.

  This clause applies only to the current instance. Therefore, in an Oracle RAC environment, authorized users without the `RESTRICTED SESSION` system privilege can still access the database by way of other instances.
- Specify `DISABLE` to reverse the effect of the `ENABLE RESTRICTED SESSION` clause, allowing all users with `CREATE SESSION` system privilege to log on to Oracle Database. This is the default.

> **See Also:**
>
> "Restricting Sessions: Example"

SET ENCRYPTION WALLET Clause

Use this clause to manage database access to the Transparent Data Encryption (TDE) master encryption key. The TDE master encryption key is stored in an external security module, which can be an encryption wallet or Hardware Security Module (HSM).

Although this statement begins with the keyword `ALTER`, an `ALTER SYSTEM SET ENCRYPTION WALLET` statement is not a DDL clause. However, you cannot roll back such a statement.

Although this clause begins with the `SET` keyword, do not confuse it with the *alter_system_set_clause*, which allows you to use the `SET` keyword to set the value of initialization parameters. `ENCRYPTION WALLET` is not an initialization parameter.

OPEN

When you specify this clause, the database uses the specified password to open the wallet and load the TDE master key into database memory for the duration of the instance, or establish a connection to the HSM in order to send the encrypted table and tablespace keys to the HSM and receive then back decrypted.

- Specify *wallet_password* to retrieve the master encryption key from the encryption wallet. If the wallet is not available or is already open, then the database returns an error. The double quotation marks around *wallet_password* are required.
- Specify *HSM_auth_string* to make the HSM accessible. *HSM_auth_string* is of the form "*user_id*:*password*" where:
  – *user_id* is the user ID created for the database using the HSM management interface
  – *password* is the password created for the user ID using the HSM management interface

  The double quotation marks around *HSM_auth_string* are required

CLOSE

Use this clause to disable encryption and decryption in your database. The *wallet_password* is required to close an encryption wallet. *HSM_auth_string* is required to disable access to the HSM. Refer to OPEN for details on specifying *HSM_auth_string*.

A password is not required to close an auto-open wallet when only an auto-open wallet is present. The password is required to close an auto-open wallet when both an auto-open wallet and an encryption wallet are open. In this case, using `CLOSE` with a password will close the auto-open wallet and the encryption wallet.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for information on setting encryption wallets in an Oracle Real Application Clusters (Oracle RAC) environment

set_encryption_key

Use this clause to generate a new TDE master encryption key, if none exists. If there are existing master keys in the HSM or keystore, then this clause rekeys the existing table and tablespace keys, that is, it decrypts all table and tablespace keys with the old master key and reencrypts them with the new master key.

An `ALTER SYSTEM SET ENCRYPTION KEY` statement is a DDL statement and will automatically commit any pending transactions in the schema.

Although this clause begins with the `SET` keyword, do not confuse it with the *alter_system_set_clause*, which allows you to use the `SET` keyword to set the value of initialization parameters. `ENCRYPTION KEY` is not an initialization parameter.

IDENTIFIED BY *wallet_password*

This clause loads the TDE master encryption key from the encryption wallet into memory for access to encrypted data.

- The *certificate_id* is required if you are using PKI asymmetric key pairs as master encryption keys. Specify the integer that identifies the certificate. You can find this value by querying the `CERT_ID` column of the `V$WALLET` dynamic performance view. Do not specify *certificate_id* if you are using symmetric keys, which are the default.

- For *wallet_password*, specify the password used to connect to the security module.

If you specify an invalid *certificate_id* or *wallet_password*, then the database returns an error. The double quotation marks around *certificate_id* and *wallet_password* are required.

Restriction on IDENTIFIED BY *wallet_password*

PKI-based master keys, including unified master encryption keys, can only be used with TDE column encryption and an Oracle Wallet, not with HSM.

> **Note:**
>
> The use of PKI encryption with Transparent Data Encryption is deprecated. To configure Transparent Data Encryption, use the ADMINISTER KEY MANAGEMENT statement. See *Oracle Database Advanced Security Guide* for more information.

IDENTIFIED BY *HSM_auth_string*

This clause creates a master encryption key that will be stored inside the HSM. The master encryption key is used to encrypt or decrypt table keys inside the HSM.

HSM_auth_string is of the form "*user_id*:*password*" where:

- *user_id* is the user ID created for the database using the HSM management interface
- *password* is the password created for the user ID using the HSM management interface

The double quotation marks around *HSM_auth_string* are required.

If you are already using Transparent Data Encryption with an Oracle Wallet and you would like to migrate to an HSM, then specify the `MIGRATE USING` *wallet_password* clause. This decrypts the existing table and tablespace keys, and then reencrypts them with the newly created, HSM-based, master encryption key. Note that the wallet is still in use after you migrate to an HSM, because it may contain master encryption keys that were used for export files, RMAN backups, or encrypted data in temporary or undo tablespaces or redo log files. After migrating, perform one of the following steps:

- Change the wallet password to the *HSM_auth_string* using Oracle Wallet Manager or the `orapki` command-line tool.
- Create a local auto-open wallet from the encryption wallet and either rename the encryption wallet, or move it out of the directory specified in `ENCRYPTION_WALLET_LOCATION` in `sqlnet.ora`. Do not delete the encryption wallet and do not forget the wallet password.

See Also:

- *Oracle Database Advanced Security Guide* for more information on using Transparent Data Encryption
- The description of the `CREATE TABLE` "*encryption_spec* " for information on using that feature to encrypt table columns
- "Establishing a Wallet and Encryption Key: Examples"

shutdown_dispatcher_clause

The `SHUTDOWN` clause is relevant only if your system is using the shared server architecture of Oracle Database. It shuts down a dispatcher identified by *`dispatcher_name`*.

Note:

Do not confuse this clause with the SQL*Plus command `SHUTDOWN`, which is used to shut down the entire database.

The *`dispatcher_name`* must be a string of the form '`D`*`xxx`*', where *`xxx`* indicates the number of the dispatcher. For a listing of dispatcher names, query the `NAME` column of the `V$DISPATCHER` dynamic performance view.

- If you specify `IMMEDIATE`, then the dispatcher stops accepting new connections immediately and Oracle Database terminates all existing connections through that dispatcher. After all sessions are cleaned up, the dispatcher process shuts down.
- If you do not specify `IMMEDIATE`, then the dispatcher stops accepting new connections immediately but waits for all its users to disconnect and for all its database links to terminate. Then it shuts down.

REGISTER Clause

Specify `REGISTER` to instruct the `PMON` background process to register the instance with the listeners immediately. If you do not specify this clause, then registration of the instance does not occur until the next time `PMON` executes the discovery routine. As a result, clients may not be able to access the services for as long as 60 seconds after the listener is started.

See Also:

Oracle Database Concepts and *Oracle Database Net Services Administrator's Guide* for information on the `PMON` background process and listeners

alter_system_set_clause

This clause allows you to change parameter values. The *set_parameter_clause* allows you to change the value of a specified initialization parameter. The USE_STORED_OUTLINES and GLOBAL_TOPIC_ENABLED clauses allow you to change the value of those system parameters.

set_parameter_clause

You can change the value of many initialization parameters for the current instance, whether you have started the database with a traditional plain-text parameter file (pfile) or with a server parameter file (spfile). *Oracle Database Reference* indicates these parameters in the "Modifiable" category of each parameter description. If you are using a pfile, then the change will persist only for the duration of the instance. However, if you have started the database with an spfile, then you can change the value of the parameter in the spfile itself, so that the new value will occur in subsequent instances.

Oracle Database Reference documents all initialization parameters in full. The parameters fall into three categories:

- **Basic parameters:** Database administrators should be familiar with and consider the setting for all of the basic parameters.
- **Functional categories:** Oracle Database Reference also lists the initialization parameters by their functional category.
- **Alphabetical listing:** The Table of Contents of *Oracle Database Reference* contains all initialization parameters in alphabetical order.

The ability to change initialization parameter values depends on whether you have started up the database with a traditional plain-text initialization parameter file (pfile) or with a server parameter file (spfile). To determine whether you can change the value of a particular parameter, query the ISSYS_MODIFIABLE column of the V$PARAMETER dynamic performance view.

When setting a parameter value, you can specify additional settings as follows:

COMMENT

The COMMENT clause lets you associate a comment string with this change in the value of the parameter. The comment string cannot contain control characters or a line break. If you also specify SPFILE, then this comment will appear in the parameter file to indicate the most recent change made to this parameter.

DEFERRED

The DEFERRED keyword sets or modifies the value of the parameter for future sessions that connect to the database. Current sessions retain the old value.

You must specify DEFERRED if the value of the ISSYS_MODIFIABLE column of V$PARAMETER for this parameter is DEFERRED. If the value of that column is IMMEDIATE, then the DEFERRED keyword in this clause is optional. If the value of that column is FALSE, then you cannot specify DEFERRED in this ALTER SYSTEM statement.

> **See Also:**
>
> *Oracle Database Reference* for information on the `V$PARAMETER` dynamic performance view

CONTAINER

You can specify the `CONTAINER` clause when you set a parameter value in a CDB. A CDB uses an inheritance model for initialization parameters in which PDBs inherit initialization parameter values from the root. In this case, inheritance means that the value of a particular parameter in the root applies to a particular PDB.

A PDB can override the root's setting for some parameters, which means that a PDB has an inheritance property for each initialization parameter that is either true or false. The inheritance property is true for a parameter when the PDB inherits the root's value for the parameter. The inheritance property is false for a parameter when the PDB does not inherit the root's value for the parameter.

The inheritance property for some parameters must be true. For other parameters, you can change the inheritance property by running the `ALTER SYSTEM SET` statement to set the parameter when the current container is the PDB. If `ISPDB_MODIFIABLE` is `TRUE` for an initialization parameter in the `V$SYSTEM_PARAMETER` view, then the inheritance property can be false for the parameter.

- If you specify `CONTAINER = ALL`, then the parameter setting applies to all containers in the CDB, including the root and all of the PDBs. The current container must be the root.

  Specifying `ALL` sets the inheritance property to true for the parameter in all PDBs.
- If you specify `CONTAINER = CURRENT`, then the parameter setting applies only to the current container. When the current container is the root, the parameter setting applies to the root and to any PDB with an inheritance property of true for the parameter.

If you omit this clause, then `CONTAINER = CURRENT` is the default.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on modifying parameters in a CDB

SCOPE

The `SCOPE` clause lets you specify when the change takes effect. The behavior of this clause depends on whether you are connected to a non-CDB, a CDB root, or a PDB.

When you issue the ALTER SYSTEM statement while connected to a non-CDB or a CDB root, the scope depends on whether you started up the database using a traditional plain-text parameter file (pfile) or server parameter file (spfile).

- `MEMORY` indicates that the change is made in memory, takes effect immediately, and persists until the database is shut down. If you started up the database using a parameter file (pfile), then this is the only scope you can specify.
- `SPFILE` indicates that the change is made in the server parameter file. The new setting takes effect when the database is next shut down and started up again. You must specify `SPFILE` when changing the value of a static parameter that is described as not modifiable in *Oracle Database Reference*.
- `BOTH` indicates that the change is made in memory and in the server parameter file. The new setting takes effect immediately and persists after the database is shut down and started up again.

If a server parameter file was used to start up the database, then `BOTH` is the default. If a parameter file was used to start up the database, then `MEMORY` is the default, as well as the only scope you can specify.

When you issue the ALTER SYSTEM statement while connected to a PDB, you can modify only initialization parameters for which the `ISPDB_MODIFIABLE` column is `TRUE` in the `V$SYSTEM_PARAMETER` view. The initialization parameter value takes effect only for the PDB. For any initialization parameter that is not set explicitly for a PDB, the PDB inherits the CDB root's parameter value.

- `MEMORY` indicates that the change is made in memory and takes effect immediately in the PDB. The setting reverts to the value set in the CDB root in the any of the following cases:
  - An `ALTER SYSTEM SET` statement sets the value of the parameter in the root with `SCOPE` equal to `BOTH` or `MEMORY`, and the PDB is closed and reopened. The parameter value in the PDB is not changed if `SCOPE` is equal to `SPFILE`, and the PDB is closed and reopened.
  - The PDB is closed and reopened.
  - The CDB is shut down and reopened.
- `SPFILE` indicates that the change is made for the PDB and stored persistently. The new setting affects only the PDB and takes effect in either of the following cases:
  - The PDB is closed and reopened.
  - The CDB is shut down and reopened.
- `BOTH` indicates that the change is made in memory, made for the PDB, and stored persistently. The new setting takes effect immediately in the PDB and persists after the PDB is closed and reopened or the CDB is shut down and reopened. The new setting affects only the PDB.

When a PDB is unplugged from a CDB, the values of the initialization parameters that were specified for the PDB with `SCOPE=BOTH` or `SCOPE=SPFILE` are added to the PDB's XML metadata file. These values are restored for the PDB when it is plugged in to a CDB.

SID

The `SID` clause lets you specify the SID of the instance where the value will take effect.

- Specify `SID = '*'` if you want Oracle Database to change the value of the parameter for all instances that do not already have an explicit setting for this parameter.

- Specify SID = *'sid'* if you want Oracle Database to change the value of the parameter only for the instance *sid*. This setting takes precedence over previous and subsequent ALTER SYSTEM SET statements that specify SID = '*'.

If you do not specify this clause, then:

- If the instance was started up with a pfile (traditional plain-text initialization parameter file), then Oracle Database assumes the SID of the current instance.
- If the instance was started up with an spfile (server parameter file), then Oracle Database assumes SID = '*'.

If you specify an instance other than the current instance, then Oracle Database sends a message to that instance to change the parameter value in the memory of that instance.

USE_STORED_OUTLINES Clause

> **Note:**
>
> Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. Refer to *Oracle Database SQL Tuning Guide* for more information about SQL plan management.

USE_STORED_OUTLINES is a system parameter, not an initialization parameter. You cannot set it in a pfile or spfile, but you can set it with an ALTER SYSTEM statement. This parameter determines whether the optimizer will use stored public outlines to generate execution plans.

- TRUE causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- FALSE specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

GLOBAL_TOPIC_ENABLED

GLOBAL_TOPIC_ENABLED is a system parameter, not an initialization parameter. You cannot set it in a pfile or spfile, but you can set it with an ALTER SYSTEM statement. This parameter determines whether all queues and topics created in Oracle Streams AQ are automatically registered with the LDAP server. If GLOBAL_TOPIC_ENABLED = TRUE when a queue table is created, altered, or dropped, then the corresponding Lightweight Directory Access Protocol (LDAP) entry is also created, altered or dropped.

The parameter works the same way for the Java Message Service (JMS). If a database has been configured to use LDAP and the GLOBAL_TOPIC_ENABLED parameter has been set to TRUE, then all JMS queues and topics are automatically registered with the LDAP server when they are created. The administrator can also create aliases to the queues and topics registered in LDAP. Queues and topics that are registered in LDAP can be looked up through JNDI using the name or alias of the queue or topic.

Shared Server Parameters

When you start your instance, Oracle Database creates shared server processes and dispatcher processes for the shared server architecture based on the values of the SHARED_SERVERS and DISPATCHERS initialization parameters. You can also set the SHARED_SERVERS and DISPATCHERS parameters with ALTER SYSTEM to perform one of the following operations while the instance is running:

- Create additional shared server processes by increasing the minimum number of shared server processes.
- Terminate existing shared server processes after their current calls finish processing.
- Create more dispatcher processes for a specific protocol, up to a maximum across all protocols specified by the initialization parameter MAX_DISPATCHERS.
- Terminate existing dispatcher processes for a specific protocol after their current user processes disconnect from the instance.

> **See Also:**
>
> - *Oracle Real Application Clusters Administration and Deployment Guide* for information on setting parameter values for an individual instance in an Oracle Real Application Clusters environment
> - The following examples of using the ALTER SYSTEM statement: "Changing Licensing Parameters: Examples", "Enabling Query Rewrite: Example", "Enabling Resource Limits: Example", "Shared Server Parameters", and "Changing Shared Server Settings: Examples"

alter_system_reset_clause

This clause lets you reset an initialization parameter.

The semantics of this clause are similar to the *set_parameter_clause*, except instead of changing the value of an initialization parameter, this clause removes the setting of an initialization parameter. Refer to the set_parameter_clause to learn about the parameters you can reset, and for the full semantics of the SCOPE and SID clauses.

RELOCATE CLIENT

This clause is valid only if you are using Oracle Flex ASM. You must issue this clause from within an Oracle ASM instance, not from a normal database instance.

Use this clause to relocate the specified client to the least loaded Oracle ASM instance. When you issue this clause, the connection to the client is terminated and the client fails over to the least loaded instance. If the client is currently connected to the least loaded instance, then the connection to the client is terminated and the client fails over to that same instance.

For *client_id*, specify a string of the following form enclosed in single quotation marks:

```
instance_name:db_name
```

where *instance_name* is the identifier for the client and *db_name* is the database name for the client. You can find these values by querying the INSTANCE_NAME and DB_NAME columns of the V$ASM_CLIENT dynamic performance view.

> **See Also:**
>
> - *Oracle Automatic Storage Management Administrator's Guide* for more information on managing Oracle Flex ASM
> - *Oracle Database Reference* for more information on the `V$ASM_CLIENT` dynamic performance view

Examples

Archiving Redo Logs Manually: Examples

The following statement manually archives the redo log file group containing the redo log entry with the SCN 9356083:

```
ALTER SYSTEM ARCHIVE LOG CHANGE 9356083;
```

The following statement manually archives the redo log file group containing a member named '`diskl:log6.log`' to an archived redo log file in the location '`diska:[arch$]`':

```
ALTER SYSTEM ARCHIVE LOG
    LOGFILE 'diskl:log6.log'
    TO 'diska:[arch$]';
```

Enabling Query Rewrite: Example

This statement enables query rewrite in all sessions for all materialized views for which query rewrite has not been explicitly disabled:

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE;
```

Restricting Sessions: Example

You might want to restrict sessions if you are performing application maintenance and you want only application developers with `RESTRICTED SESSION` system privilege to log on. To restrict sessions, issue the following statement:

```
ALTER SYSTEM
   ENABLE RESTRICTED SESSION;
```

You can then terminate any existing sessions using the `KILL SESSION` clause of the `ALTER SYSTEM` statement.

After performing maintenance on your application, issue the following statement to allow any user with `CREATE SESSION` system privilege to log on:

```
ALTER SYSTEM
   DISABLE RESTRICTED SESSION;
```

Establishing a Wallet and Encryption Key: Examples

The following statements load information from the server wallet into memory and set the Transparent Data Encryption master key:

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN IDENTIFIED BY "password";
ALTER SYSTEM SET ENCRYPTION KEY IDENTIFIED BY "password";
```

These statements assume that you have initialized the security module and created a wallet with *password*.

Closing a Wallet: Examples

The following statement removes password-based wallet information from memory:

```
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE IDENTIFIED BY "password";
```

The following statement removes password-based wallet information and auto-login information, if present, from memory:

```
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE;
```

Clearing the Shared Pool: Example

You might want to clear the shared pool before beginning performance analysis. To clear the shared pool, issue the following statement:

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

Forcing a Checkpoint: Example

The following statement forces a checkpoint:

```
ALTER SYSTEM CHECKPOINT;
```

Enabling Resource Limits: Example

This `ALTER SYSTEM` statement dynamically enables resource limits:

```
ALTER SYSTEM SET RESOURCE_LIMIT = TRUE;
```

Changing Shared Server Settings: Examples

The following statement changes the minimum number of shared server processes to 25:

```
ALTER SYSTEM SET SHARED_SERVERS = 25;
```

If there are currently fewer than 25 shared server processes, then Oracle Database creates more. If there are currently more than 25, then Oracle Database terminates some of them when they are finished processing their current calls if the load could be managed by the remaining 25.

The following statement dynamically changes the number of dispatcher processes for the TCP/IP protocol to 5 and the number of dispatcher processes for the ipc protocol to 10:

```
ALTER SYSTEM
   SET DISPATCHERS =
      '(INDEX=0)(PROTOCOL=TCP)(DISPATCHERS=5)',
      '(INDEX=1)(PROTOCOL=ipc)(DISPATCHERS=10)';
```

If there are currently fewer than 5 dispatcher processes for TCP, then Oracle Database creates new ones. If there are currently more than 5, then Oracle Database terminates some of them after the connected users disconnect.

If there are currently fewer than 10 dispatcher processes for ipc, then Oracle Database creates new ones. If there are currently more than 10, then Oracle Database terminates some of them after the connected users disconnect.

If there are currently existing dispatchers for another protocol, then the preceding statement does not affect the number of dispatchers for that protocol.

Changing Licensing Parameters: Examples

The following statement dynamically changes the limit on sessions for your instance to 64 and the warning threshold for sessions on your instance to 54:

```
ALTER SYSTEM
   SET LICENSE_MAX_SESSIONS = 64
   LICENSE_SESSIONS_WARNING = 54;
```

If the number of sessions reaches 54, then Oracle Database writes a warning message to the `ALERT` file for each subsequent session. Also, users with `RESTRICTED SESSION` system privilege receive warning messages when they begin subsequent sessions.

If the number of sessions reaches 64, then only users with `RESTRICTED SESSION` system privilege can begin new sessions until the number of sessions falls below 64 again.

The following statement dynamically disables the limit for sessions on your instance. After you issue this statement, Oracle Database no longer limits the number of sessions on your instance.

```
ALTER SYSTEM SET LICENSE_MAX_SESSIONS = 0;
```

The following statement dynamically changes the limit on the number of users in the database to 200. After you issue the preceding statement, Oracle Database prevents the number of users in the database from exceeding 200.

```
ALTER SYSTEM SET LICENSE_MAX_USERS = 200;
```

Forcing a Log Switch: Example

You might want to force a log switch to drop or rename the current redo log file group or one of its members, because you cannot drop or rename a file while Oracle Database is writing to it. The forced log switch affects only the redo log thread of your instance. The following statement forces a log switch:

```
ALTER SYSTEM SWITCH LOGFILE;
```

Enabling Distributed Recovery: Example

The following statement enables distributed recovery:

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

You might want to disable distributed recovery for demonstration or testing purposes. You can disable distributed recovery in both single-process and multiprocess mode with the following statement:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

When your demonstration or testing is complete, you can then enable distributed recovery again by issuing an `ALTER SYSTEM` statement with the `ENABLE DISTRIBUTED RECOVERY` clause.

Terminating a Session: Example

You might want to terminate the session of a user that is holding resources needed by other users. The user receives an error message indicating that the session has been terminated. That user can no longer make calls to the database without beginning a new session. Consider this data from the V$SESSION dynamic performance table, when the users SYS and oe both have open sessions:

```
SELECT sid, serial#, username
   FROM V$SESSION;

       SID    SERIAL# USERNAME
---------- ---------- ------------------------------
        29         85 SYS
        33          1
        35          8
        39         23 OE
        40          1
. . .
```

The following statement terminates the session of the user scott using the SID and SERIAL# values from V$SESSION:

```
ALTER SYSTEM KILL SESSION '39, 23';
```

Disconnecting a Session: Example

The following statement disconnects user scott's session, using the SID and SERIAL# values from V$SESSION:

```
ALTER SYSTEM DISCONNECT SESSION '13, 8' POST_TRANSACTION;
```

ALTER TABLE

Purpose

Use the ALTER TABLE statement to alter the definition of a nonpartitioned table, a partitioned table, a table partition, or a table subpartition. For object tables or relational tables with object columns, use ALTER TABLE to convert the table to the latest definition of its referenced type after the type has been altered.

> **Note:**
>
> Oracle recommends that you use the ALTER MATERIALIZED VIEW LOG statement, rather than ALTER TABLE, whenever possible for operations on materialized view log tables.

> **See Also:**
>
> - CREATE TABLE for information on creating tables
> - *Oracle Text Reference* for information on ALTER TABLE statements in conjunction with Oracle Text

Prerequisites

The table must be in your own schema, or you must have `ALTER` object privilege on the table, or you must have `ALTER ANY TABLE` system privilege.

Additional Prerequisites for Partitioning Operations

If you are not the owner of the table, then you need the `DROP ANY TABLE` privilege in order to use the *`drop_table_partition`* or *`truncate_table_partition`* clause.

You must also have space quota in the tablespace in which space is to be acquired in order to use the *`add_table_partition`*, *`modify_table_partition`*, *`move_table_partition`*, and *`split_table_partition`* clauses.

When a partitioning operation cascades to reference-partitioned child tables, privileges are not required on the reference-partitioned child tables.

When using the *`exchange_partition_subpart`* clause, if the table data being exchanged contains an identity column and you are not the owner of both tables involved in the exchange, then you must have the `ALTER ANY SEQUENCE` system privilege.

You cannot partition a non-partitioned table that has an object type.

Additional Prerequisites for Constraints and Triggers

To enable a unique or primary key constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle Database creates an index on the columns of the unique or primary key in the schema containing the table.

To enable or disable triggers, the triggers must be in your schema or you must have the `ALTER ANY TRIGGER` system privilege.

> **See Also:**
>
> CREATE INDEX for information on the privileges needed to create indexes

Additional Prerequisites When Using Object Types

To use an object type in a column definition when modifying a table, either that object must belong to the same schema as the table being altered, or you must have either the `EXECUTE ANY TYPE` system privilege or the `EXECUTE` object privilege for the object type.

Additional Prerequisites for Flashback Data Archive Operations

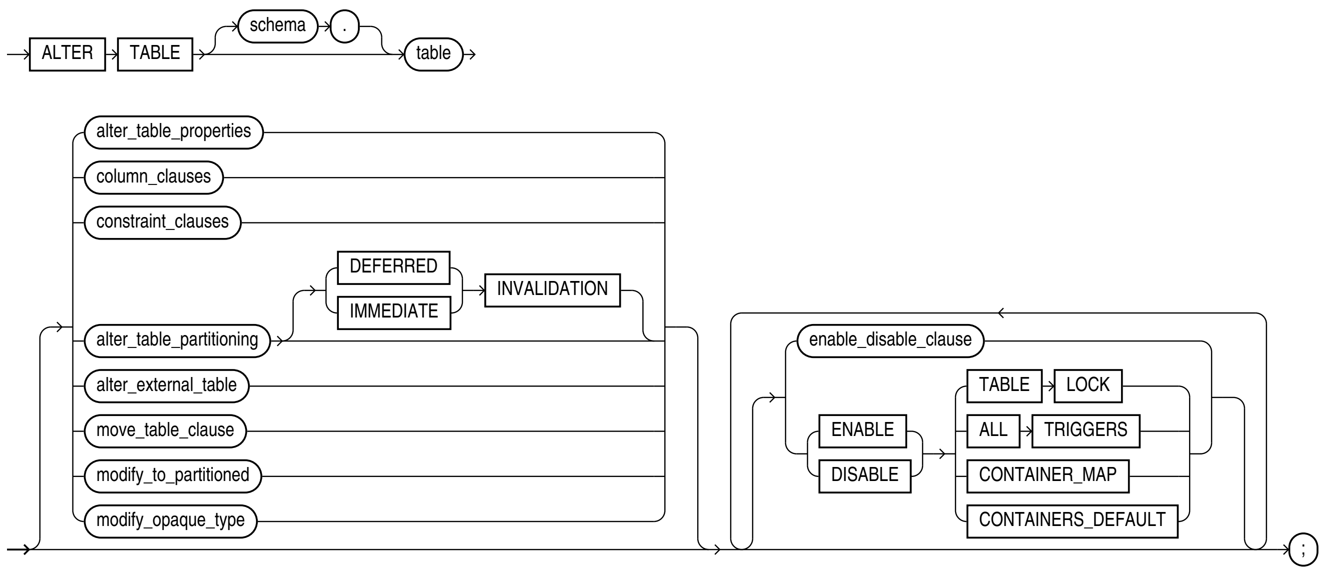
To use the *`flashback_archive_clause`* to *enable* historical tracking for the table, you must have the `FLASHBACK ARCHIVE` object privilege on the flashback data archive that will contain the historical data. To use the *`flashback_archive_clause`* to *disable* historical tracking for the table, you must have the `FLASHBACK ARCHIVE ADMINSTER` system privilege or you must be logged in as `SYSDBA`.

Additional Prerequisite for Referring to Editioned Objects

To specify an edition in the *evaluation_edition_clause* or the *unusable_editions_clause*, you must have the USE privilege on the edition.

Syntax

***alter_table*::=**



Note:

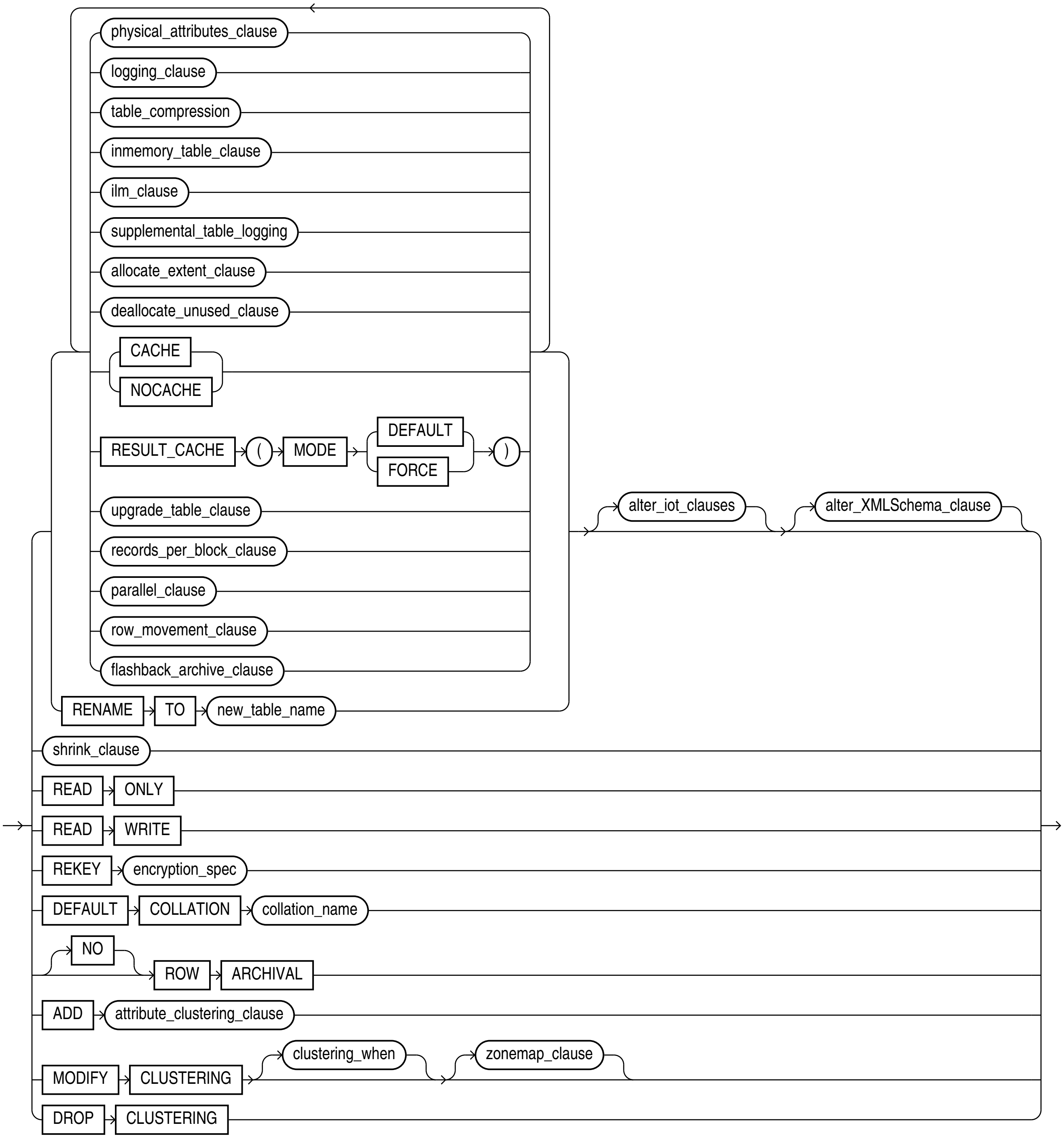
You must specify some clause after *table*. None of the clauses after *table* are required, but you must specify at least one of them.

Groups of ALTER TABLE syntax:

- ***alter_table_properties*::=**
- ***column_clauses*::=**
- ***constraint_clauses*::=**
- ***alter_table_partitioning*::=**
- ***alter_external_table*::=**
- ***move_table_clause*::=**
- ***modify_to_partitioned*::=**
- ***modify_opaque_type*::=**
- ***enable_disable_clause*::=**

After each clause you will find links to its component subclauses.

***alter_table_properties*::=**

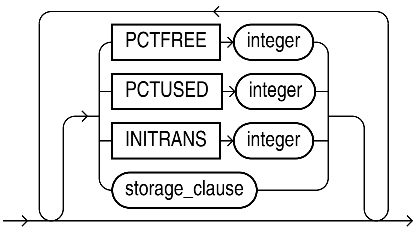


Note:

If you specify the `MODIFY CLUSTERING` clause, then you must specify at least one of the clauses *`clustering_when`* or *`zonemap_clause`*.

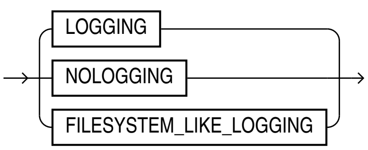
(*physical_attributes_clause*::=, *logging_clause*::=, *table_compression*::=, *inmemory_table_clause*::=, *ilm_clause*::=, *supplemental_table_logging*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::= , *upgrade_table_clause*::=, *records_per_block_clause*::=, *parallel_clause*::=, *row_movement_clause*::=, *flashback_archive_clause*::=, *shrink_clause*::=, *attribute_clustering_clause*::=, *clustering_when*::=, *zonemap_clause*::=, *alter_iot_clauses*::=, *alter_XMLSchema_clause*::=)

***physical_attributes_clause*::=**

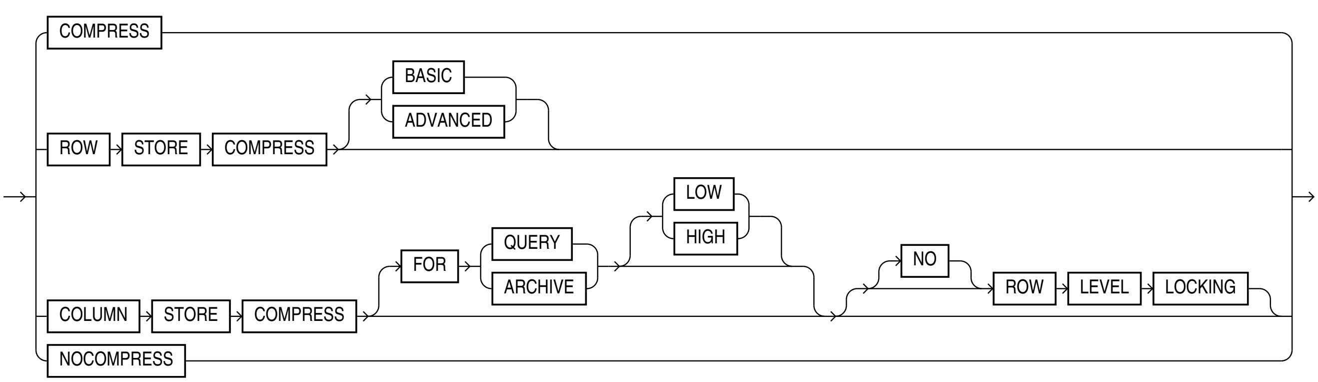


(*storage_clause*::=)

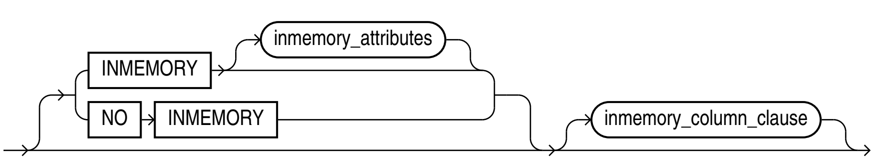
***logging_clause*::=**



***table_compression*::=**

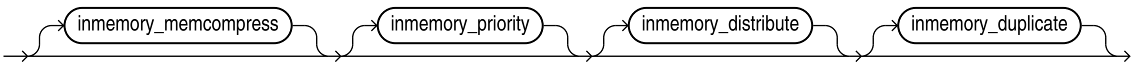


***inmemory_table_clause*::=**



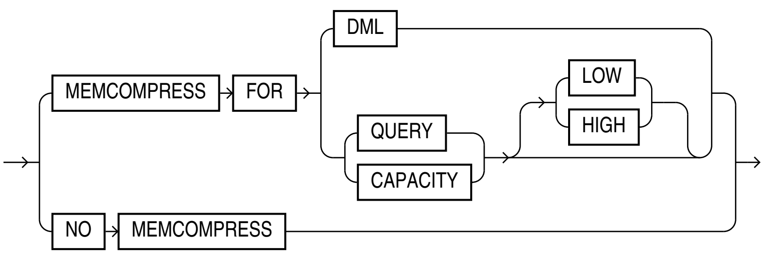
(*inmemory_attributes*::=, *inmemory_column_clause*::=)

***inmemory_attributes*::=**

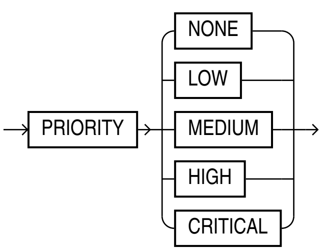


(*inmemory_memcompress*::=, *inmemory_priority*::=, *inmemory_distribute*::=, *inmemory_duplicate*::=)

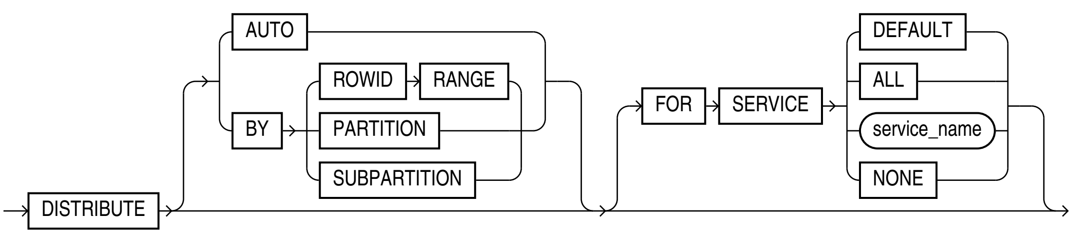
***inmemory_memcompress*::=**



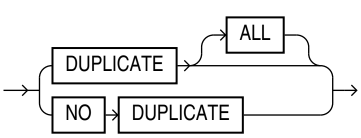
***inmemory_priority*::=**



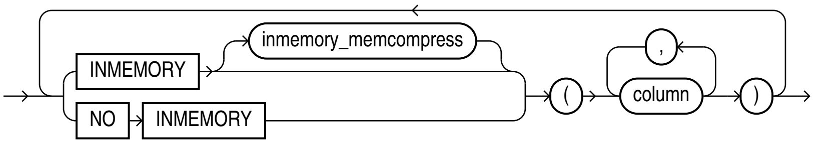
***inmemory_distribute*::=**



***inmemory_duplicate*::=**

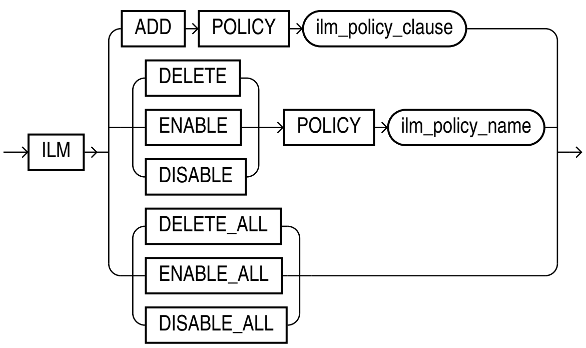


***inmemory_column_clause*::=**

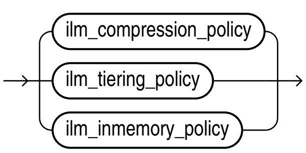


(*inmemory_memcompress::=*)

***ilm_clause*::=**

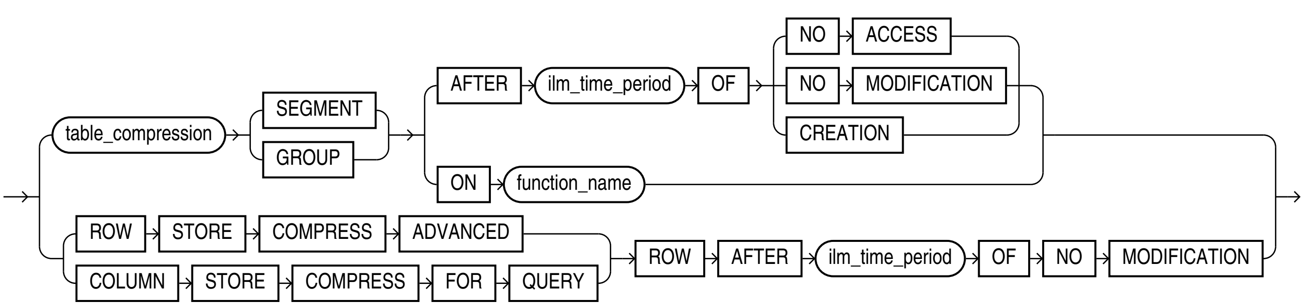


***ilm_policy_clause*::=**



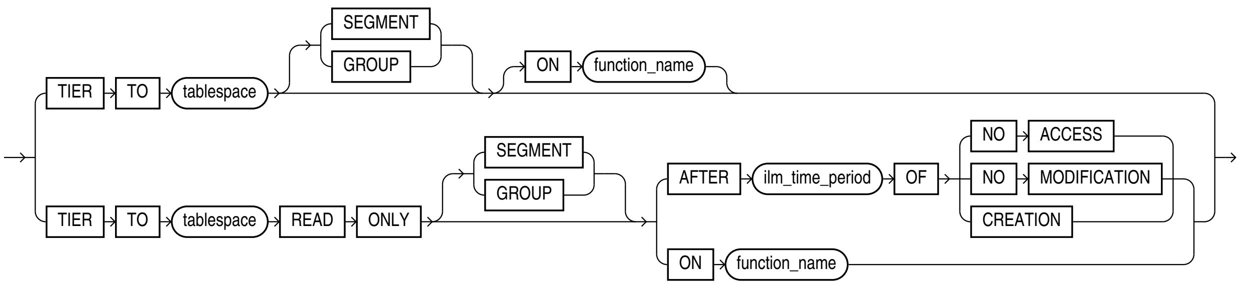
(*ilm_compression_policy::=*, *ilm_tiering_policy::=*, *ilm_inmemory_policy::=*)

***ilm_compression_policy*::=**



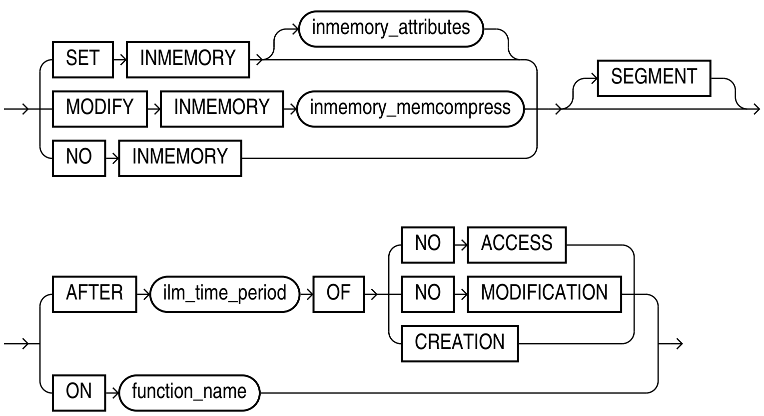
(*table_compression::=*, *ilm_time_period::=*)

***ilm_tiering_policy*::=**

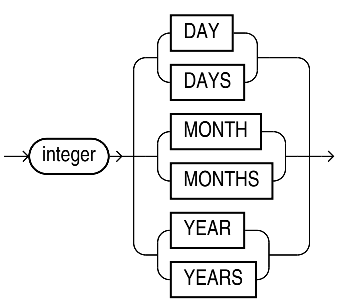


(*ilm_time_period::=*)

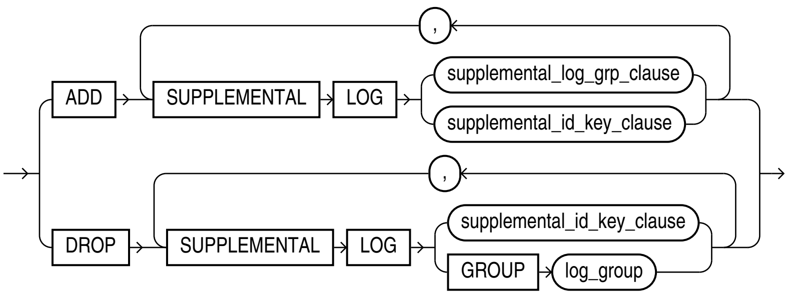
***ilm_inmemory_policy*::=**



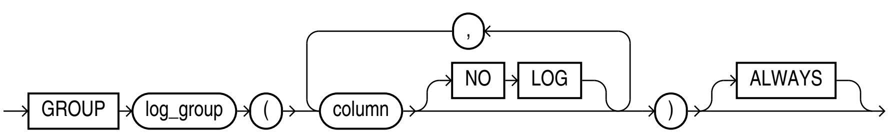
***ilm_time_period*::=**



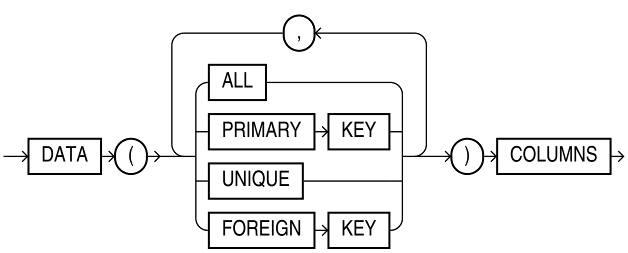
***supplemental_table_logging*::=**



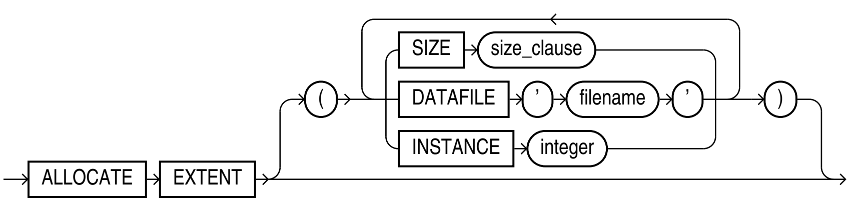
***supplemental_log_grp_clause*::=**



***supplemental_id_key_clause*::=**

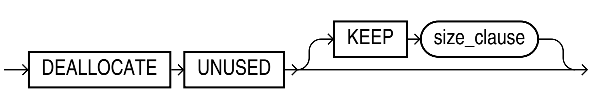


***allocate_extent_clause*::=**



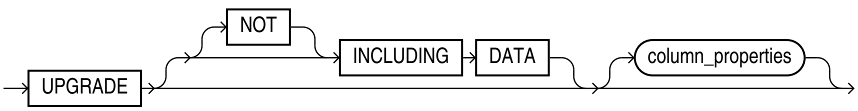
(*size_clause*::=)

***deallocate_unused_clause*::=**



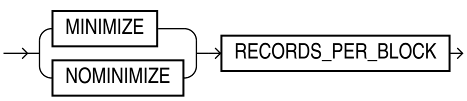
(*size_clause::=*)

***upgrade_table_clause*::=**

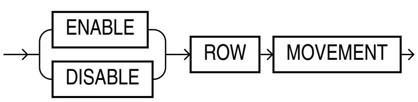


(*column_properties::=*)

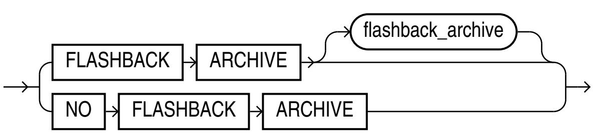
***records_per_block_clause*::=**



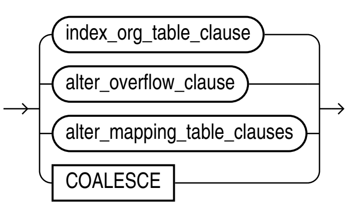
***row_movement_clause*::=**



***flashback_archive_clause*::=**

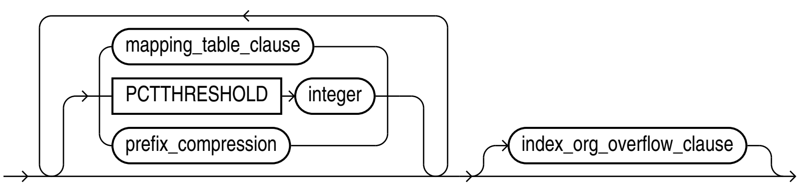


***alter_iot_clauses*::=**



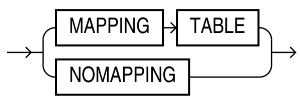
(*alter_overflow_clause::=*, *alter_mapping_table_clauses::=*)

***index_org_table_clause*::=**

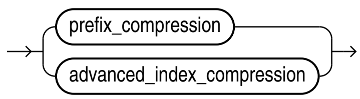


(*mapping_table_clauses*::=, *prefix_compression*::=, *index_org_overflow_clause*::=)

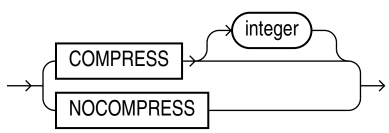
***mapping_table_clauses*::=**



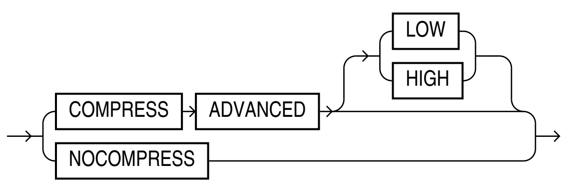
***index_compression*::=**



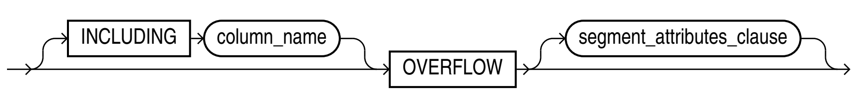
***prefix_compression*::=**



***advanced_index_compression*::=**

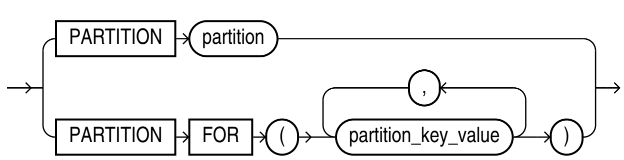


***index_org_overflow_clause*::=**

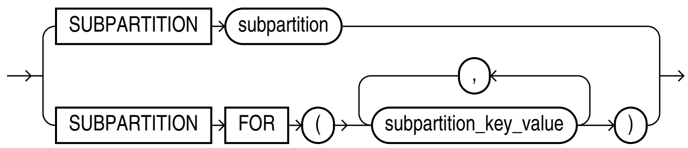


(*segment_attributes_clause*::=)

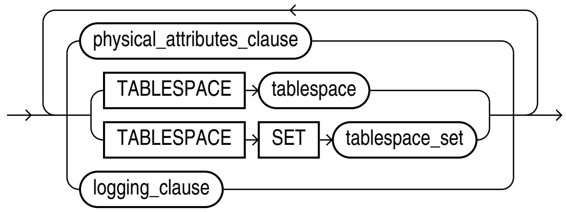
***partition_extended_name*::=**



***subpartition_extended_name*::=**

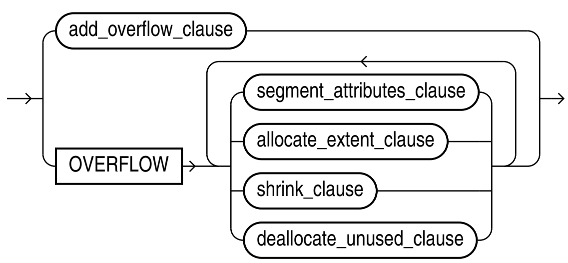


***segment_attributes_clause*::=**



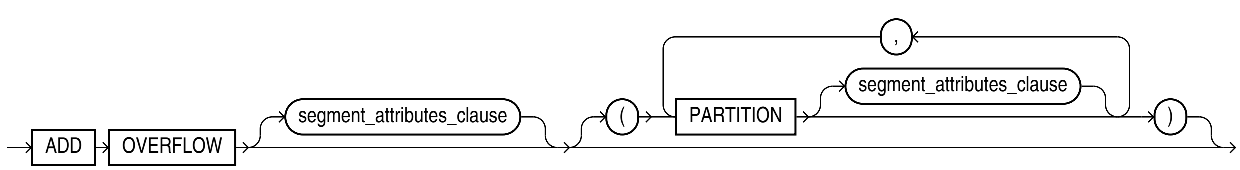
(*physical_attributes_clause::=*, `TABLESPACE SET`: not supported with `ALTER TABLE`, *logging_clause::=*)

***alter_overflow_clause*::=**



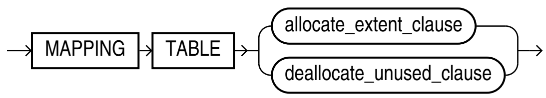
(*segment_attributes_clause::=*, *allocate_extent_clause::=*, *shrink_clause::=*, *deallocate_unused_clause::=*)

***add_overflow_clause*::=**



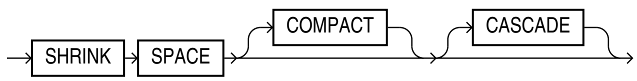
(*segment_attributes_clause::=*)

***alter_mapping_table_clauses*::=**

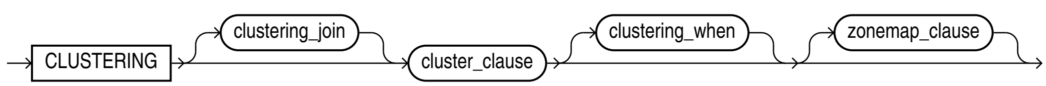


(*allocate_extent_clause::=*, *deallocate_unused_clause::=*)

***shrink_clause*::=**

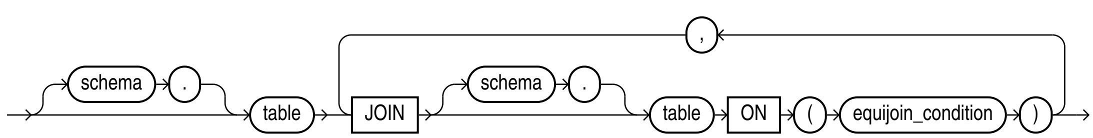


***attribute_clustering_clause*::=**

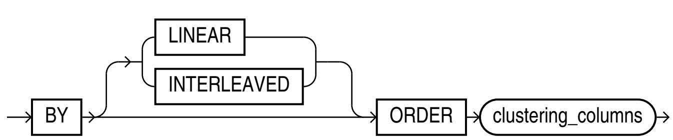


(*clustering_join::=*, *cluster_clause::=*, *clustering_when::=*, *zonemap_clause::=*)

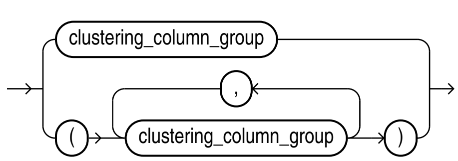
***clustering_join*::=**



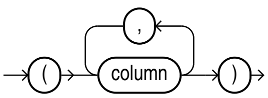
***cluster_clause*::=**



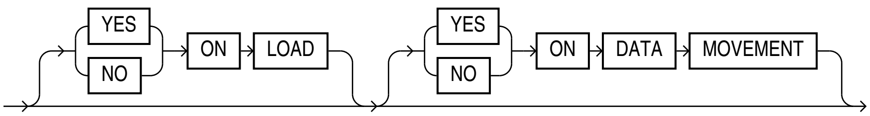
***clustering_columns*::=**



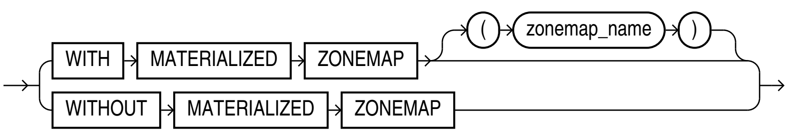
***clustering_column_group*::=**



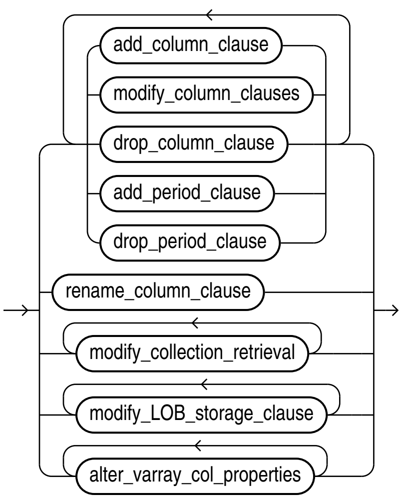
***clustering_when*::=**



***zonemap_clause*::=**

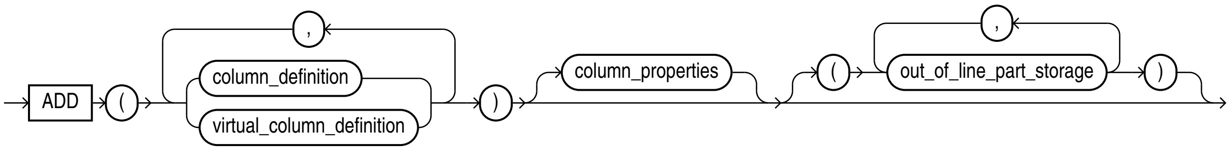


***column_clauses*::=**



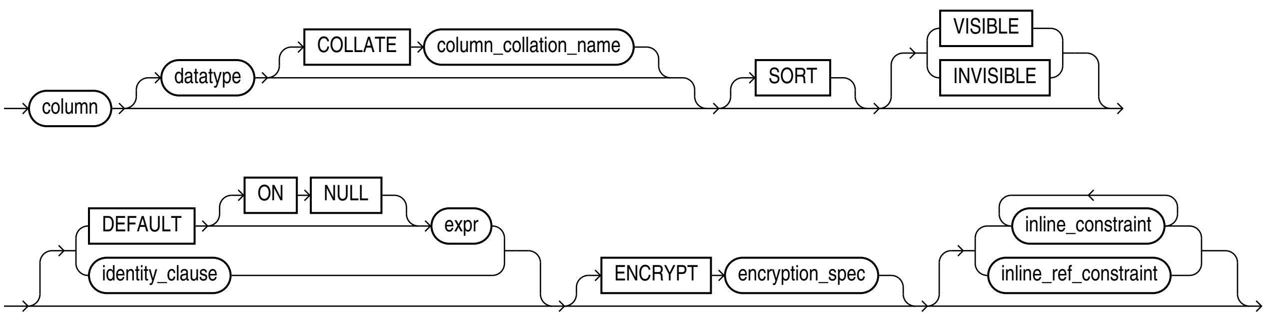
(*add_column_clause*::=, *modify_column_clauses*::=, *drop_column_clause*::=, *add_period_clause*::=, *drop_period_clause*::=, *rename_column_clause*::=, *modify_collection_retrieval*::=, *modify_LOB_storage_clause*::=, *alter_varray_col_properties*::=)

add_column_clause::=



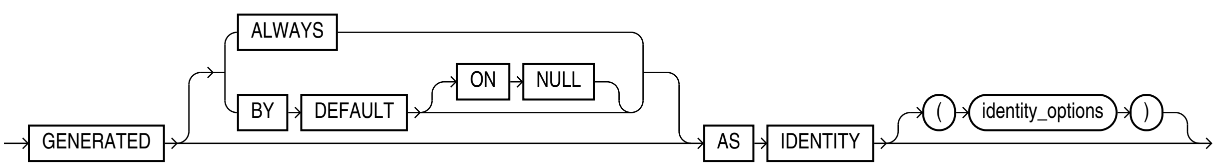
(*column_definition*::=, *virtual_column_definition*::=, *column_properties*::=, *out_of_line_part_storage*::=)

column_definition::=

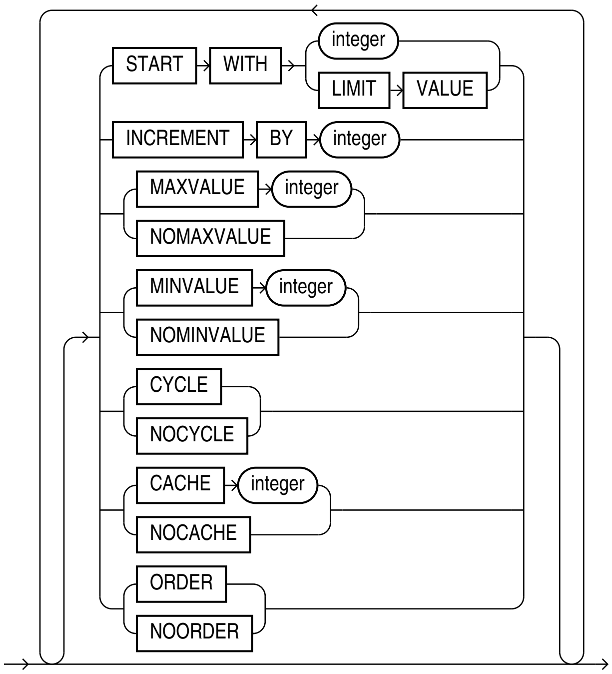


(*identity_clause*::=, *encryption_spec*::=, `inline_constraint` and `inline_ref_constraint`: *constraint*::=)

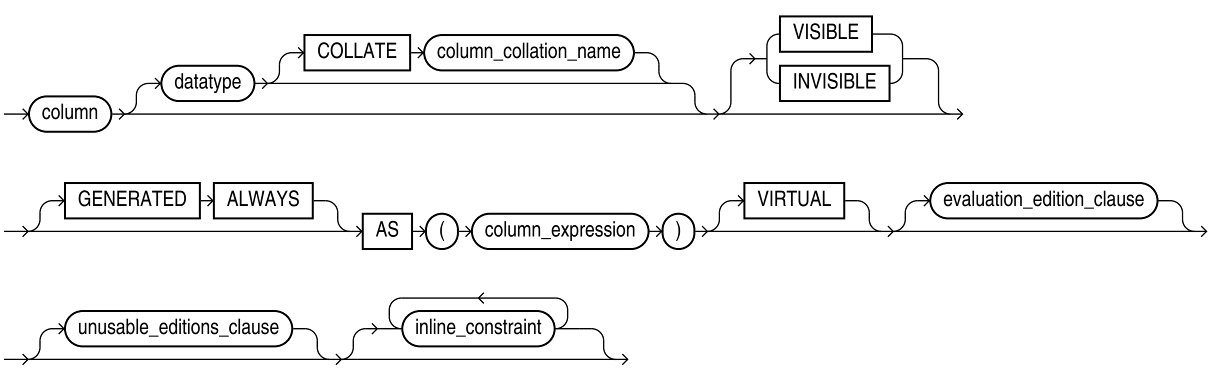
identity_clause::=



***identity_options*::=**

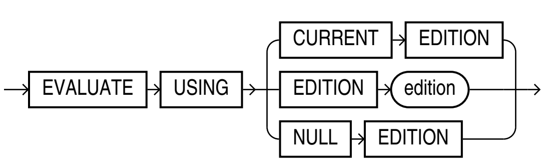


***virtual_column_definition*::=**

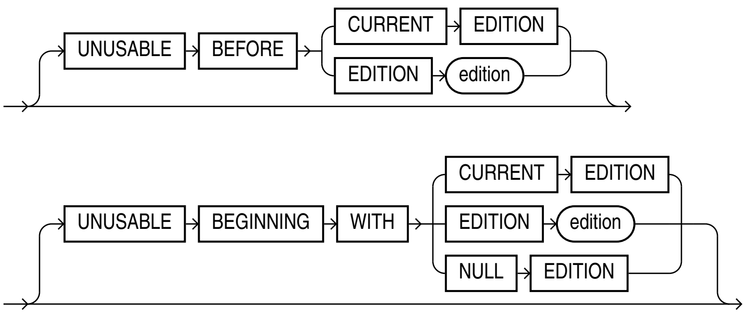


(*evaluation_edition_clause*::=, *unusable_editions_clause*::=, *constraint*::=)

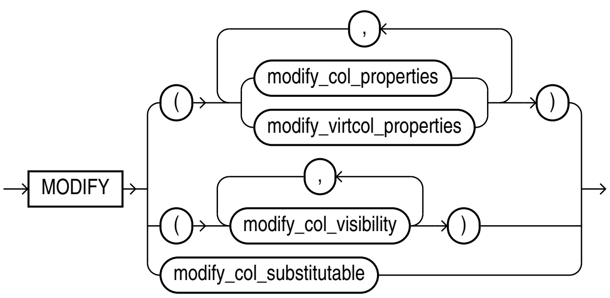
***evaluation_edition_clause*::=**



***unusable_editions_clause*::=**

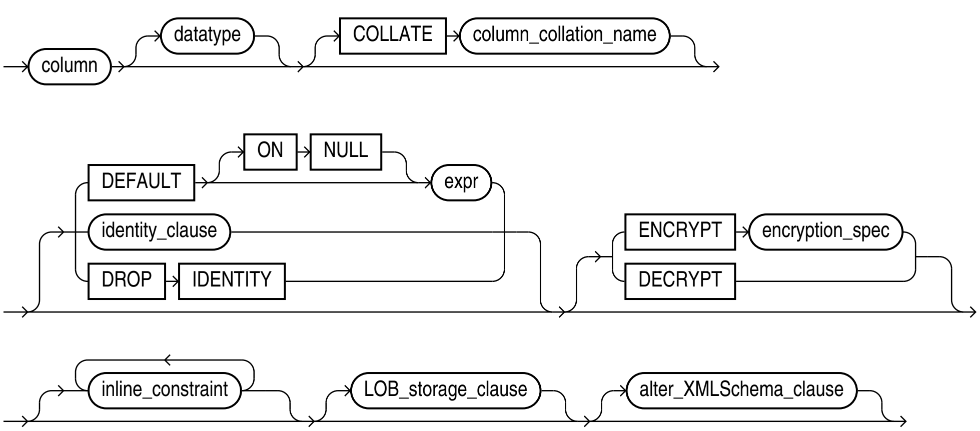


***modify_column_clauses*::=**



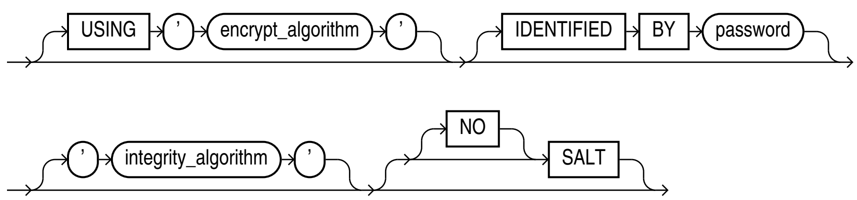
(*modify_col_properties*::=, *modify_virtcol_properties*::=, *modify_col_visibility*::=, *modify_col_substitutable*::=)

***modify_col_properties*::=**

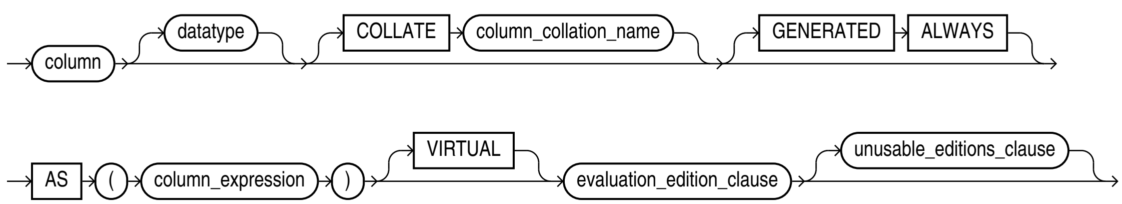


(*identity_clause*::=, *encryption_spec*::=, `inline_constraint`: *constraint*::=, *LOB_storage_clause*::=, *alter_XMLSchema_clause*::=)

***encryption_spec*::=**

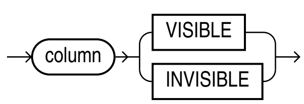


***modify_virtcol_properties*::=**



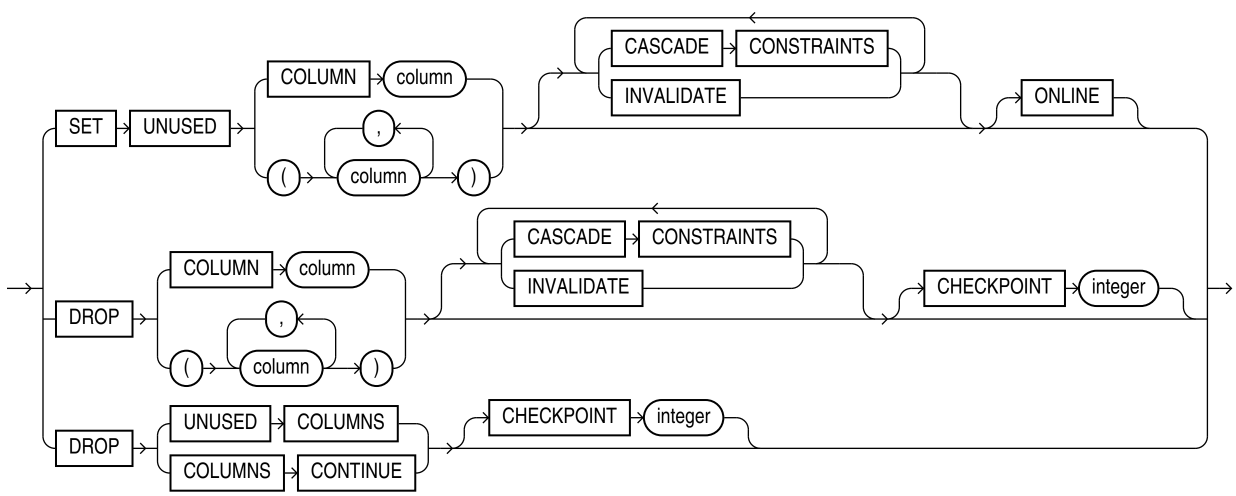
(*evaluation_edition_clause*::=, *unusable_editions_clause*::=)

***modify_col_visibility*::=**

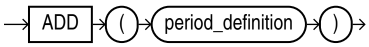


***modify_col_substitutable*::=**

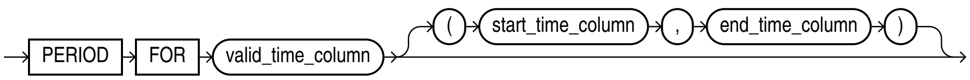
***drop_column_clause*::=**



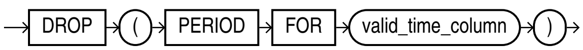
add_period_clause::=



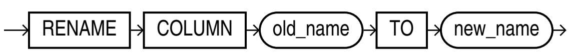
period_definition::=



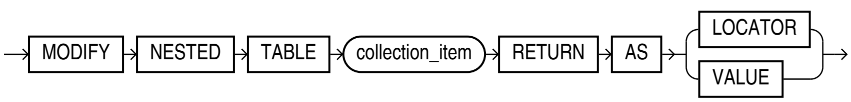
drop_period_clause::=



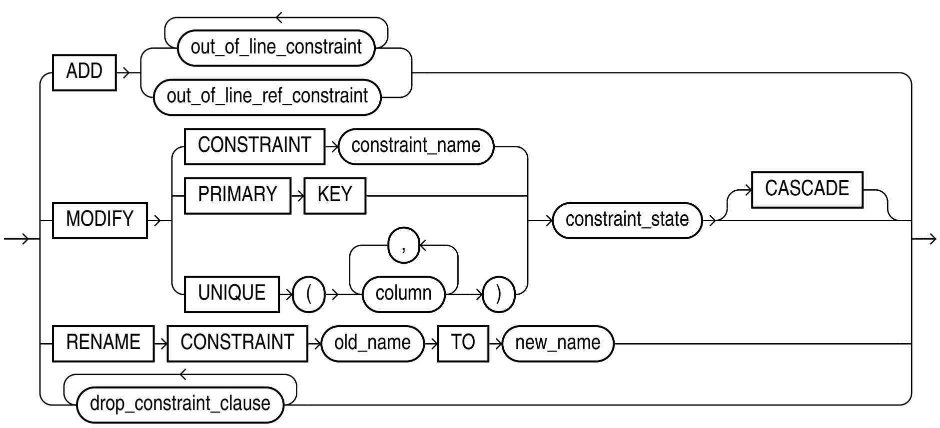
rename_column_clause::=



modify_collection_retrieval::=

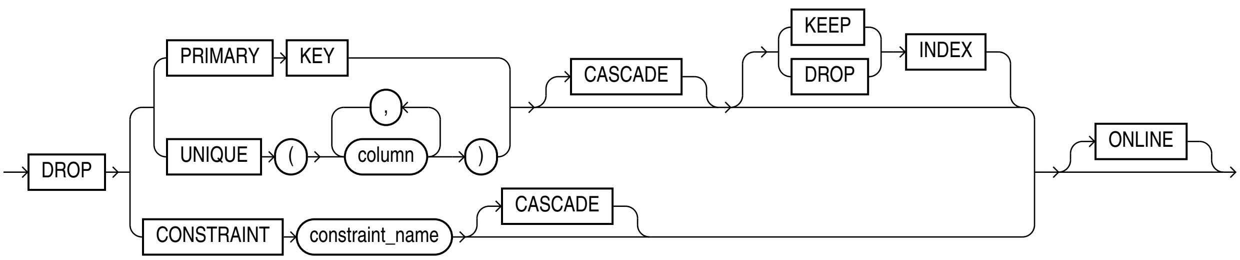


constraint_clauses::=

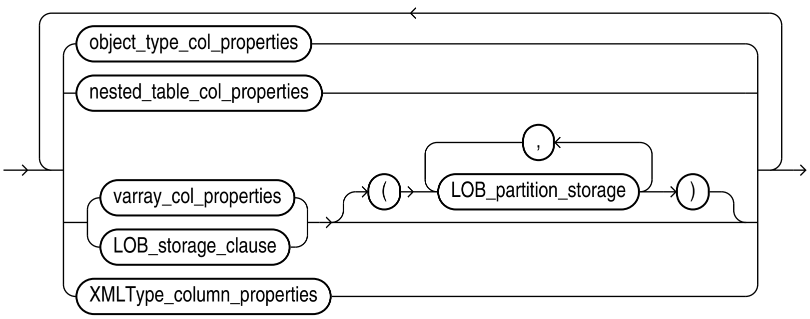


(*out_of_line_constraint*::=, *out_of_line_ref_constraint*::=, *constraint_state*::=)

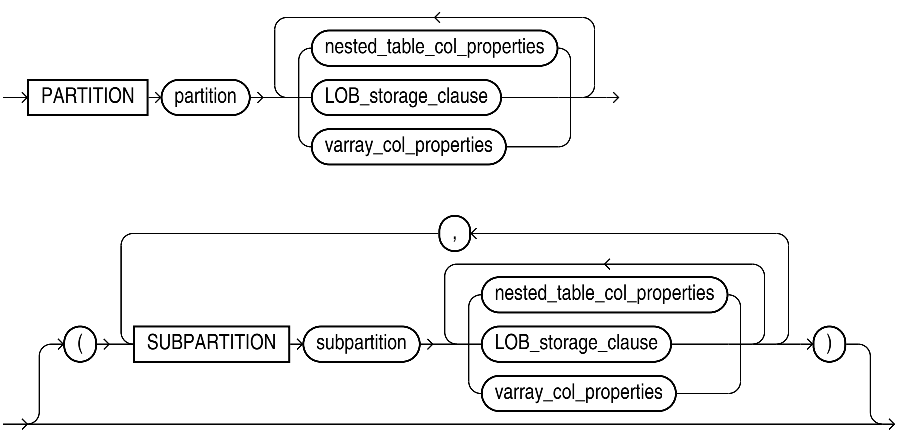
***drop_constraint_clause*::=**



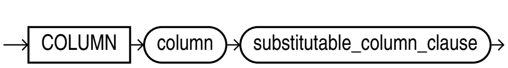
***column_properties*::=**



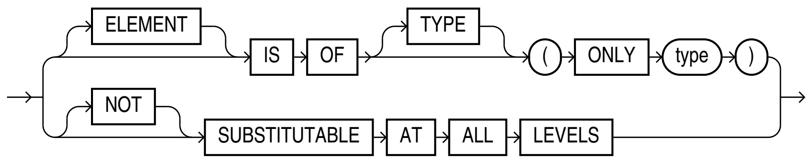
***out_of_line_part_storage*::=**



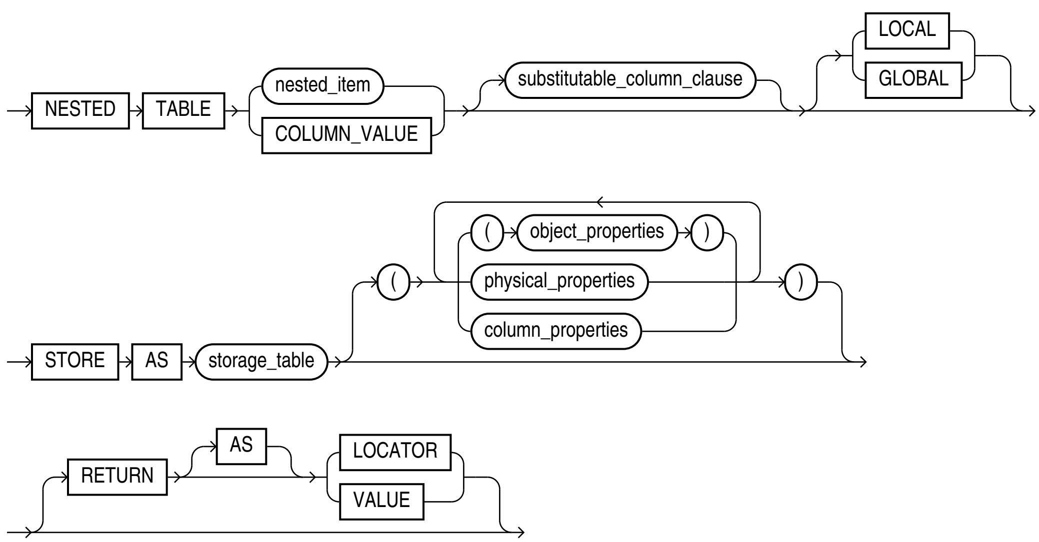
***object_type_col_properties*::=**



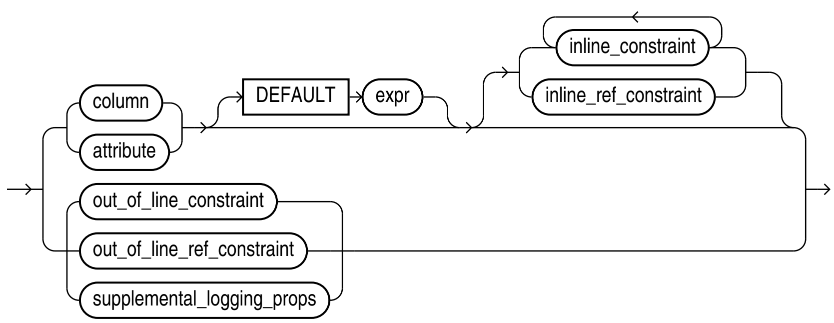
***substitutable_column_clause*::=**



***nested_table_col_properties*::=**

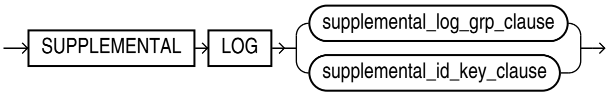


***object_properties*::=**



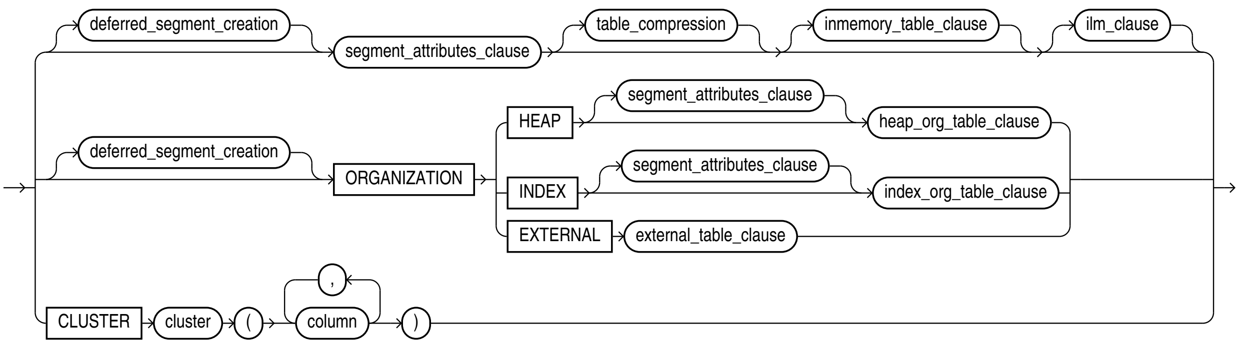
(`inline_constraint`, `inline_ref_constraint`, `out_of_line_constraint`, `out_of_line_ref_constraint`: constraint::=)

***supplemental_logging_props*::=**



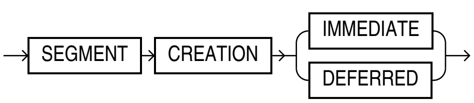
(*supplemental_log_grp_clause*::=, *supplemental_id_key_clause*::=)

***physical_properties*::=**

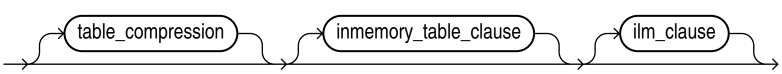


(*deferred_segment_creation*::= , *segment_attributes_clause*::=, *table_compression*::=, *inmemory_table_clause*::=—part of `CREATE TABLE` syntax, *ilm_clause*::=, *heap_org_table_clause*::=, *index_org_table_clause*::=, *external_table_clause*::=—part of `CREATE TABLE` syntax)

***deferred_segment_creation*::=**

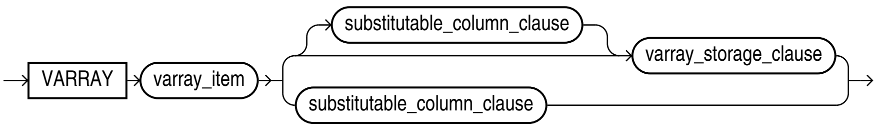


***heap_org_table_clause*::=**



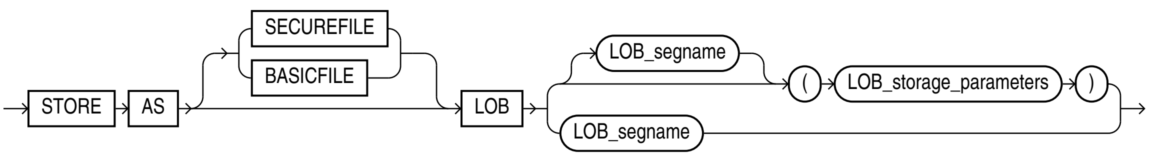
(*table_compression*::=, *inmemory_table_clause*::=—part of `CREATE TABLE` syntax, *ilm_clause*::=)

***varray_col_properties*::=**



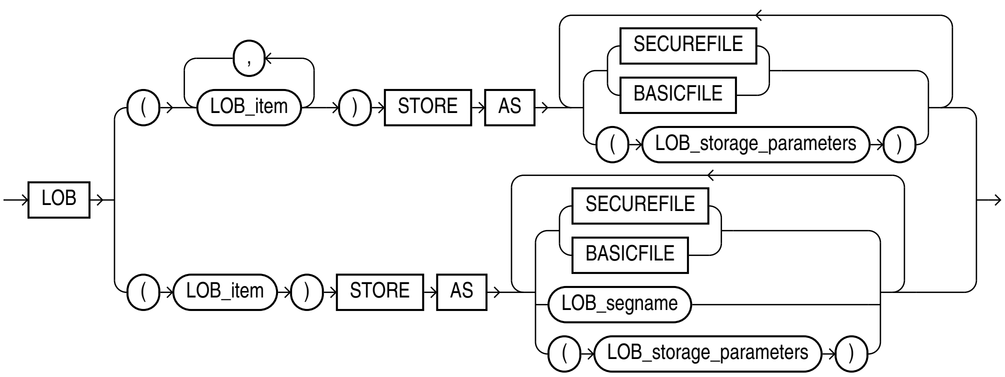
(*substitutable_column_clause::=*, *varray_storage_clause::=*)

***varray_storage_clause*::=**



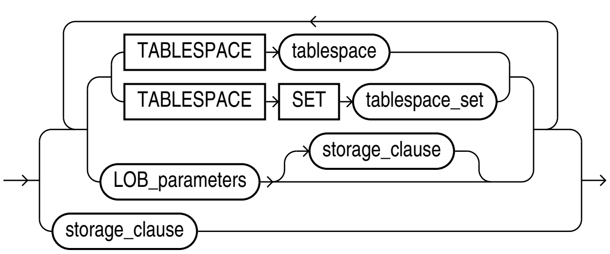
(*LOB_parameters::=*)

***LOB_storage_clause*::=**



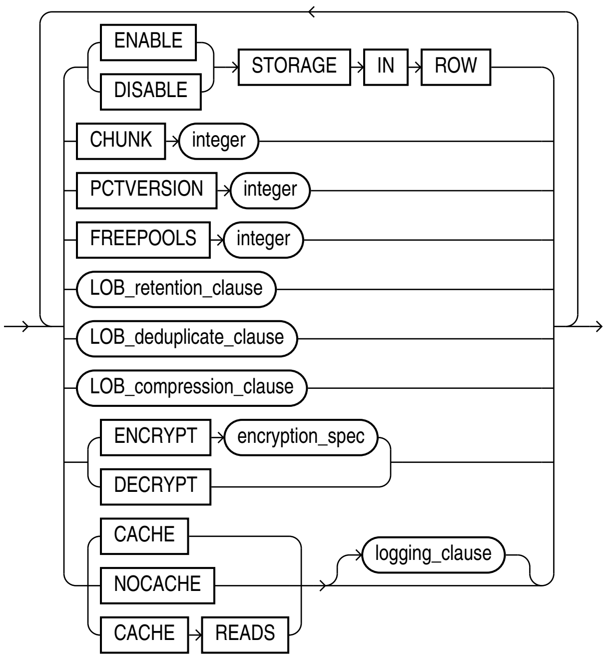
(*LOB_storage_parameters::=*)

***LOB_storage_parameters*::=**



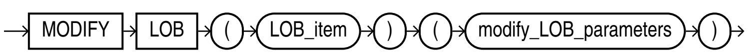
(TABLESPACE SET: not supported with ALTER TABLE, *LOB_parameters*::=, *storage_clause*::=)

***LOB_parameters*::=**

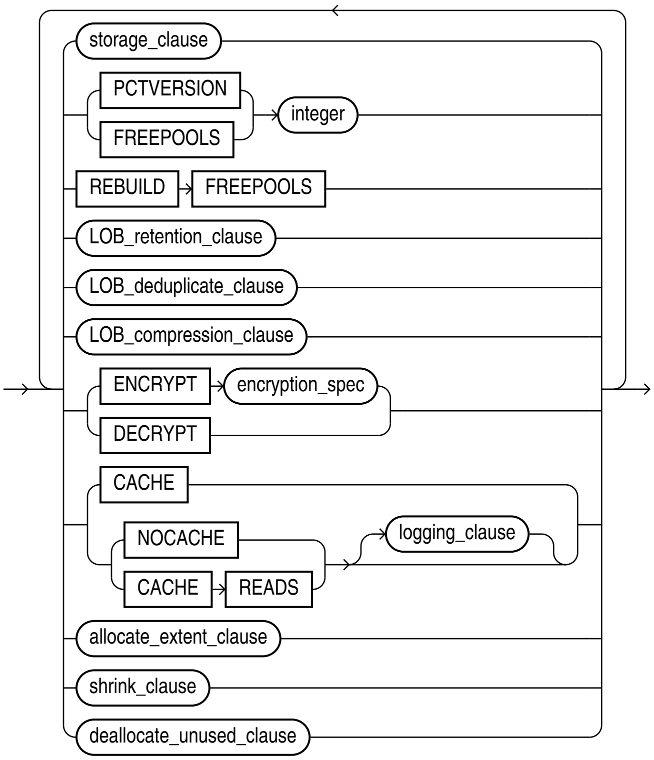


(*LOB_retention_clause*::=, *LOB_deduplicate_clause*::=, *LOB_compression_clause*::=, *encryption_spec*::=, *logging_clause*::=)

***modify_LOB_storage_clause*::=**

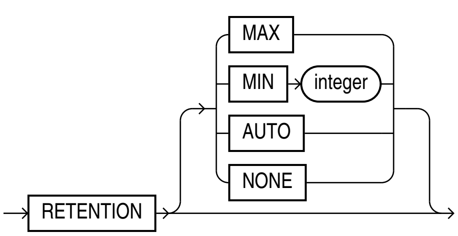


***modify_LOB_parameters*::=**

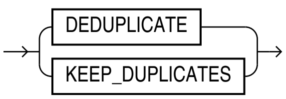


(*storage_clause*::=, *LOB_retention_clause*::=, *LOB_compression_clause*::=, *encryption_spec*::=, *logging_clause*::=, *allocate_extent_clause*::=, *shrink_clause*::=, *deallocate_unused_clause*::=)

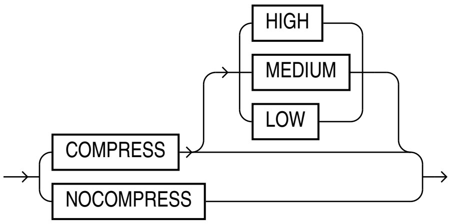
***LOB_retention_clause*::=**



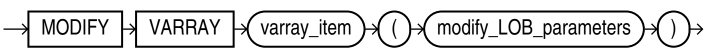
***LOB_deduplicate_clause*::=**



***LOB_compression_clause*::=**

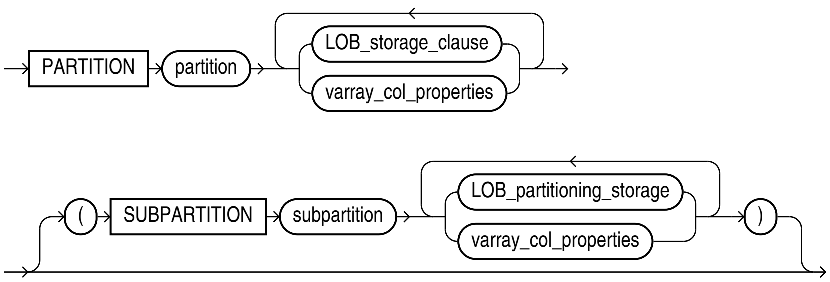


***alter_varray_col_properties*::=**



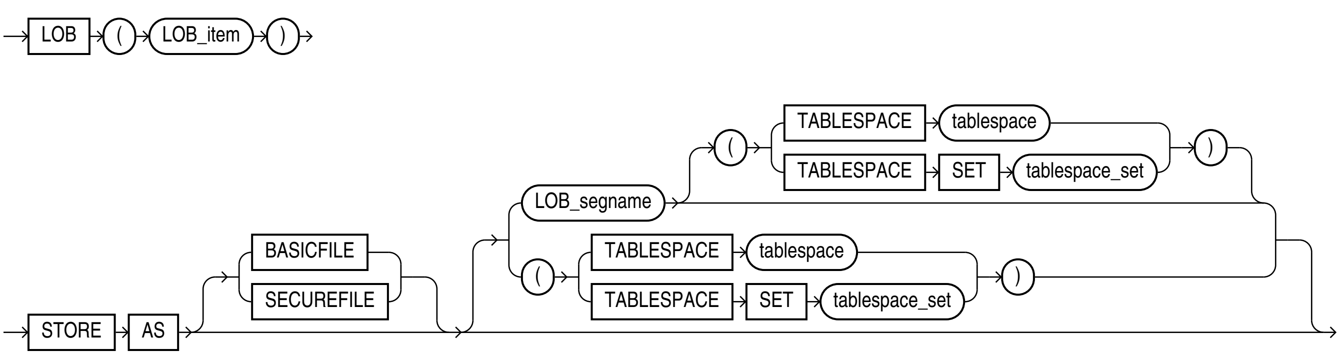
(*modify_LOB_parameters::=*)

***LOB_partition_storage*::=**



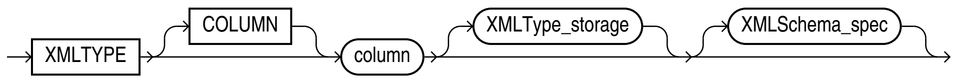
(*LOB_storage_clause::=*, *varray_col_properties::=*, *LOB_partitioning_storage::=*)

***LOB_partitioning_storage*::=**

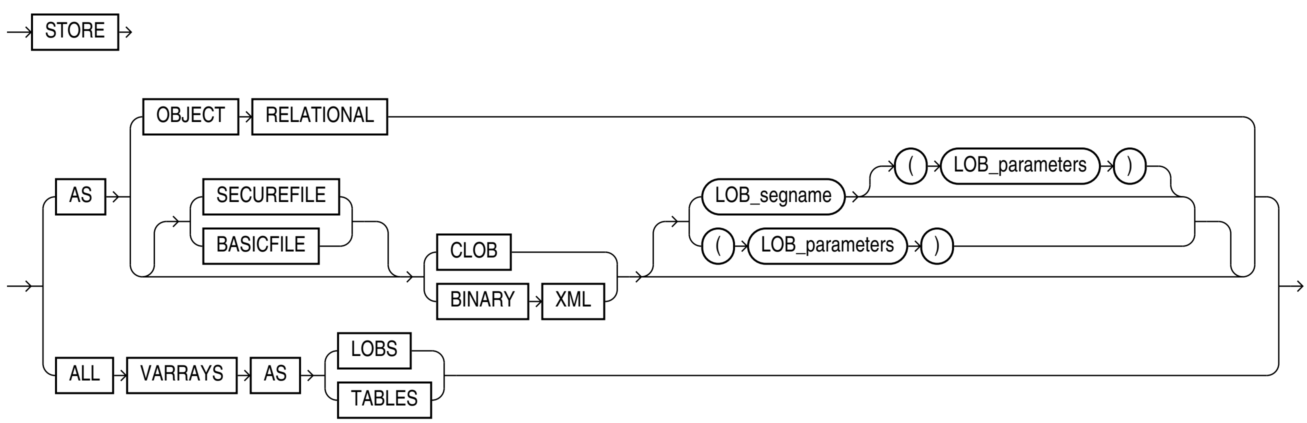


(`TABLESPACE SET`: not supported with `ALTER TABLE`)

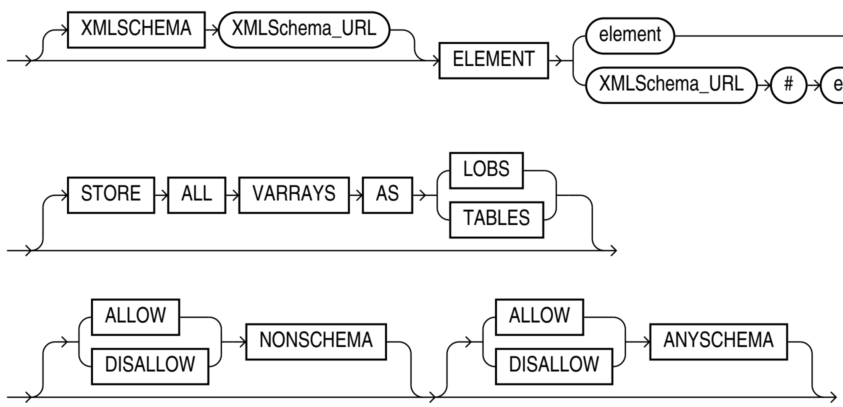
***XMLType_column_properties*::=**



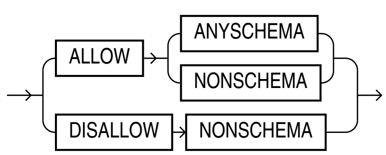
***XMLType_storage*::=**



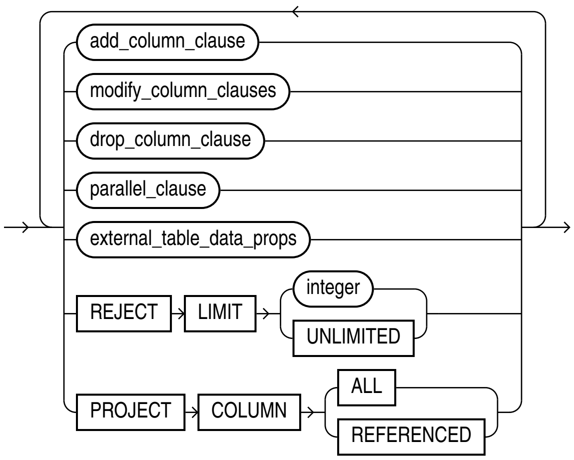
***XMLSchema_spec*::=**



***alter_XMLSchema_clause*::=**

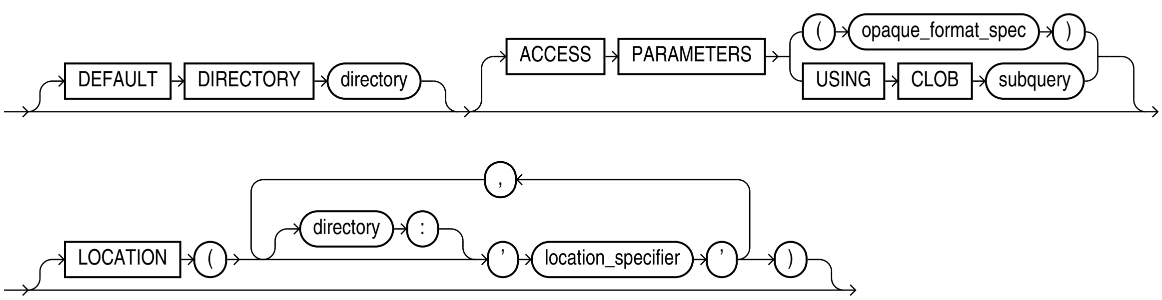


***alter_external_table*::=**

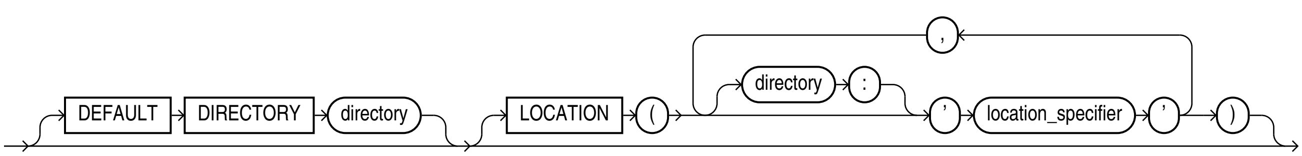


(*add_column_clause*::=, *modify_column_clauses*::=, *drop_column_clause*::=, *parallel_clause*::=, *external_table_data_props*::=)

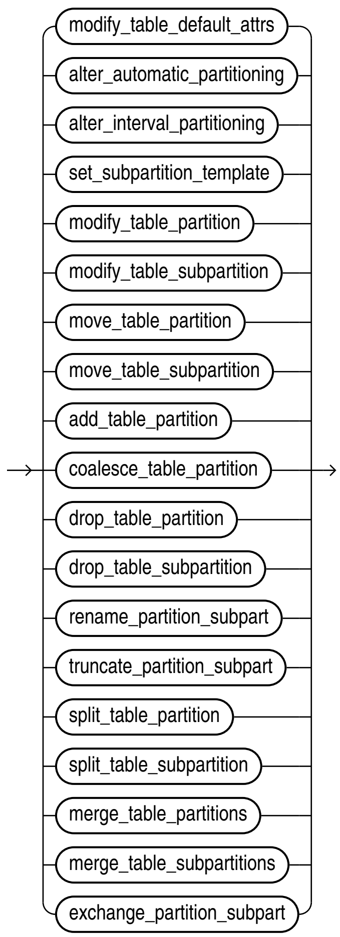
***external_table_data_props*::=**



***external_part_subpart_data_props*::=**

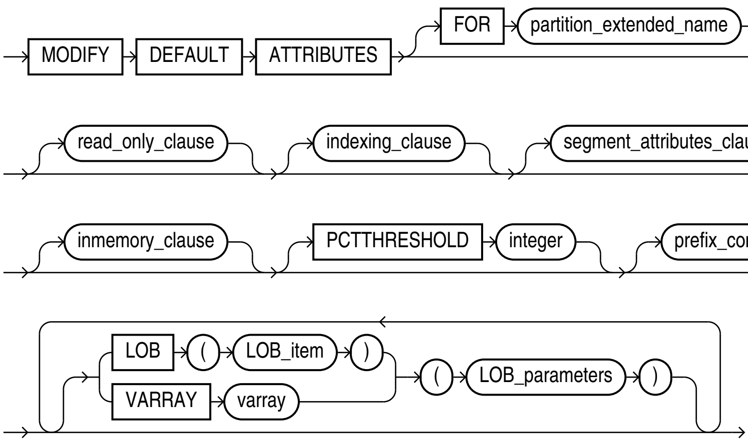


***alter_table_partitioning*::=**



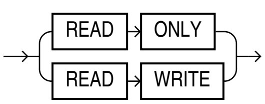
(*modify_table_default_attrs*::=, *alter_automatic_partitioning*::=, *alter_interval_partitioning*::=, *set_subpartition_template*::=, *modify_table_partition*::=, *modify_table_subpartition*::=, *move_table_partition*::=, *move_table_subpartition*::=, *add_table_partition*::=, *coalesce_table_partition*::=, *drop_table_partition*::=, *drop_table_subpartition*::=, *rename_partition_subpart*::=, *truncate_partition_subpart*::=, *split_table_partition*::=, *split_table_subpartition*::=, *merge_table_partitions*::=, *merge_table_subpartitions*::=, *exchange_partition_subpart*::=

***modify_table_default_attrs*::=**

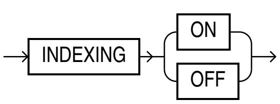


(*partition_extended_name*::=, *deferred_segment_creation*::= , *read_only_clause*::=, *indexing_clause*::=, *segment_attributes_clause*::=, *table_compression*::=, *inmemory_clause*::=, *prefix_compression*::=, *alter_overflow_clause*::=, *LOB_parameters*::=)

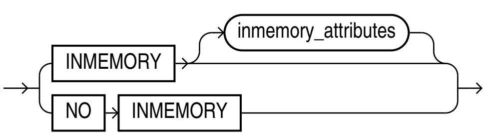
***read_only_clause*::=**



***indexing_clause*::=**

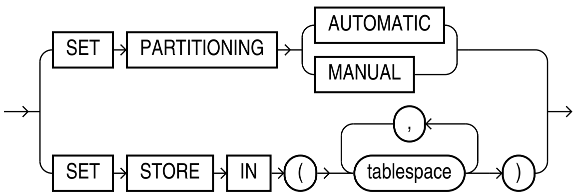


***inmemory_clause*::=**

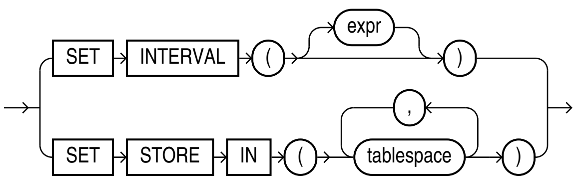


(*inmemory_attributes*::=)

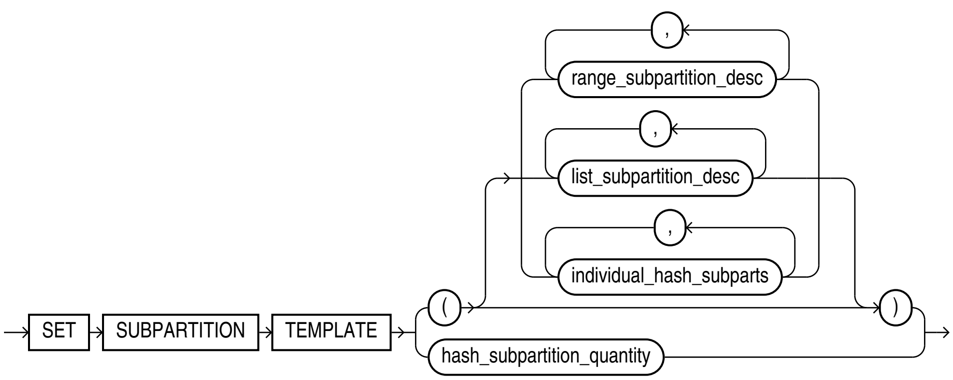
***alter_automatic_partitioning*::=**



***alter_interval_partitioning*::=**

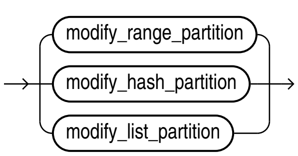


***set_subpartition_template*::=**



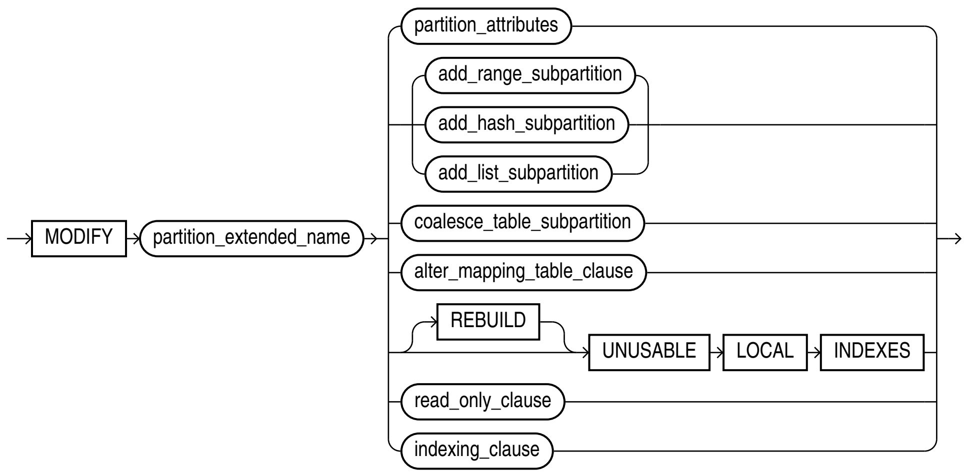
(*range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=)

***modify_table_partition*::=**



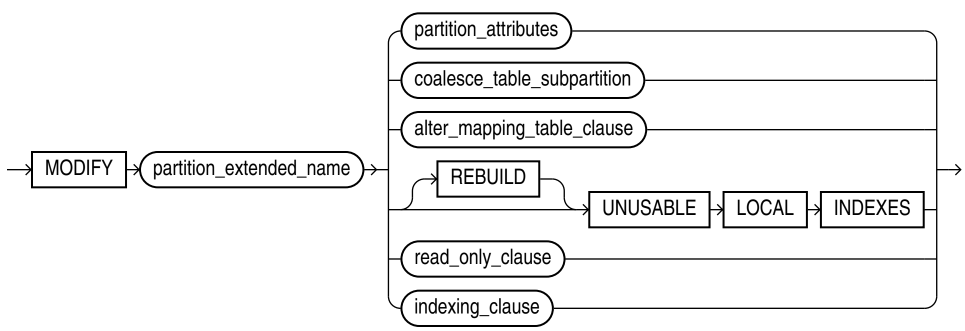
(*modify_range_partition*::=, *modify_hash_partition*::=, *modify_list_partition*::=)

***modify_range_partition*::=**



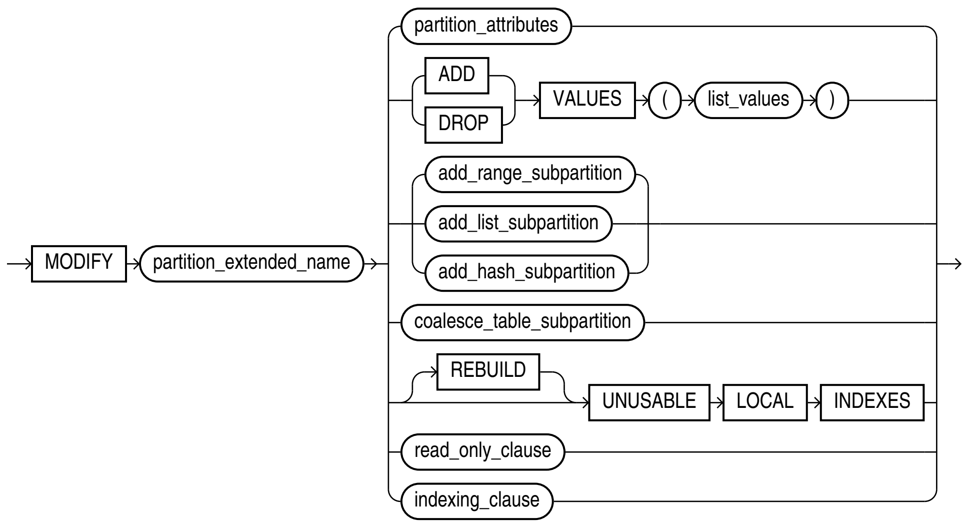
(*partition_extended_name*::=, *partition_attributes*::=, *add_range_subpartition*::=, *add_hash_subpartition*::=, *add_list_subpartition*::=, *coalesce_table_subpartition*::=, *alter_mapping_table_clauses*::=, *read_only_clause*::=, *indexing_clause*::=)

***modify_hash_partition*::=**



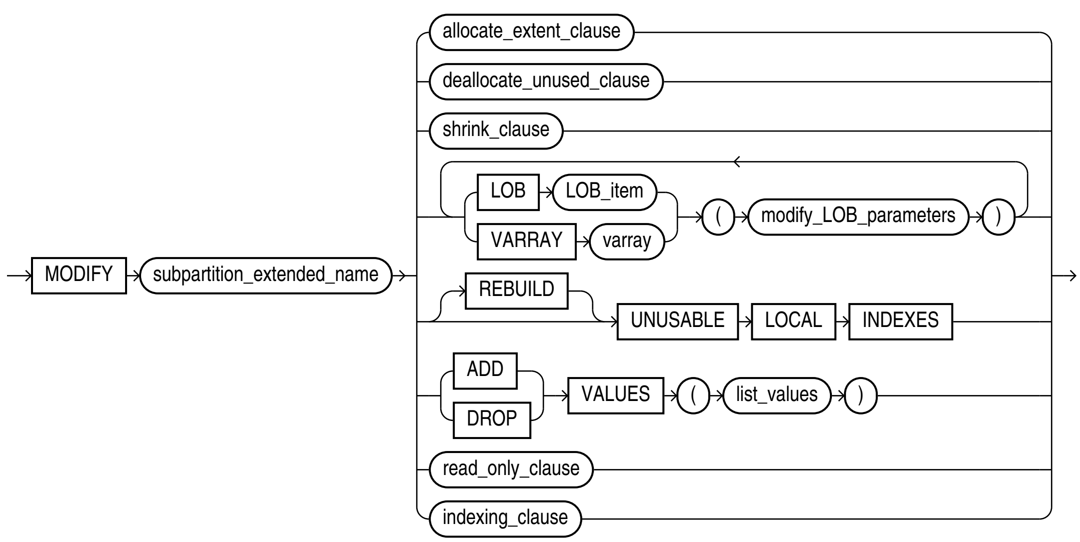
(*partition_extended_name*::=, *coalesce_table_subpartition*::=, *partition_attributes*::=, *alter_mapping_table_clauses*::=, *read_only_clause*::=, *indexing_clause*::=)

***modify_list_partition*::=**



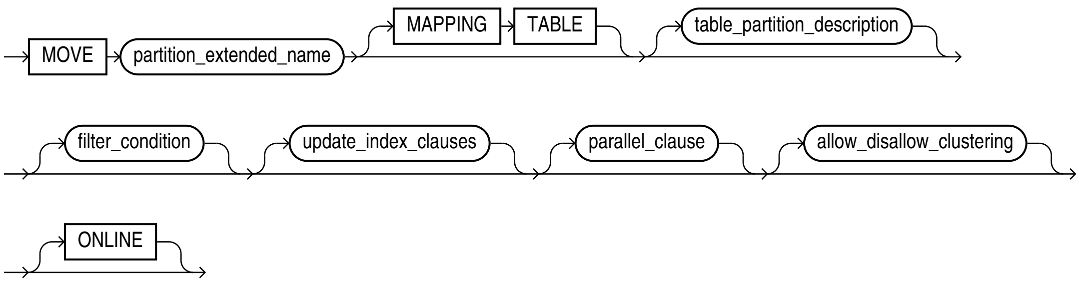
(*partition_extended_name*::=, *partition_attributes*::=, *list_values*::=, *add_range_subpartition*::=, *add_list_subpartition*::=, *add_hash_subpartition*::=, *coalesce_table_subpartition*::=, *read_only_clause*::=, *indexing_clause*::=)

***modify_table_subpartition*::=**



(*subpartition_extended_name*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::=, *shrink_clause*::=, *modify_LOB_parameters*::=, *list_values*::=, *read_only_clause*::=, *indexing_clause*::=)

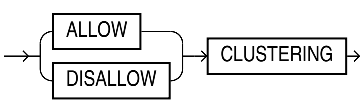
***move_table_partition*::=**



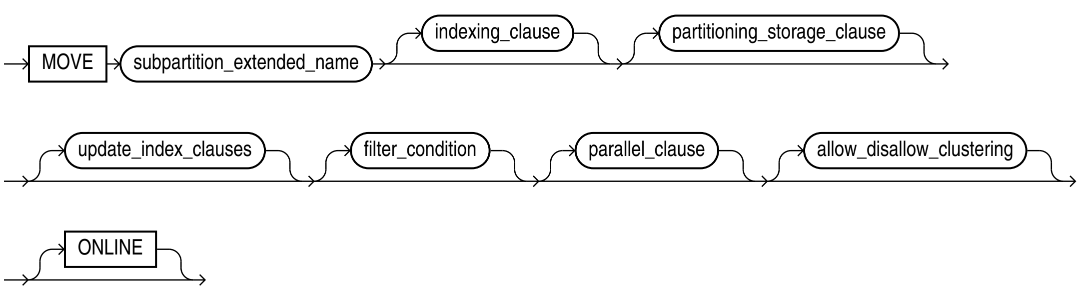
(*partition_extended_name*::=, *table_partition_description*::=, *filter_condition*::=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

***filter_condition*::=**

***allow_disallow_clustering*::=**

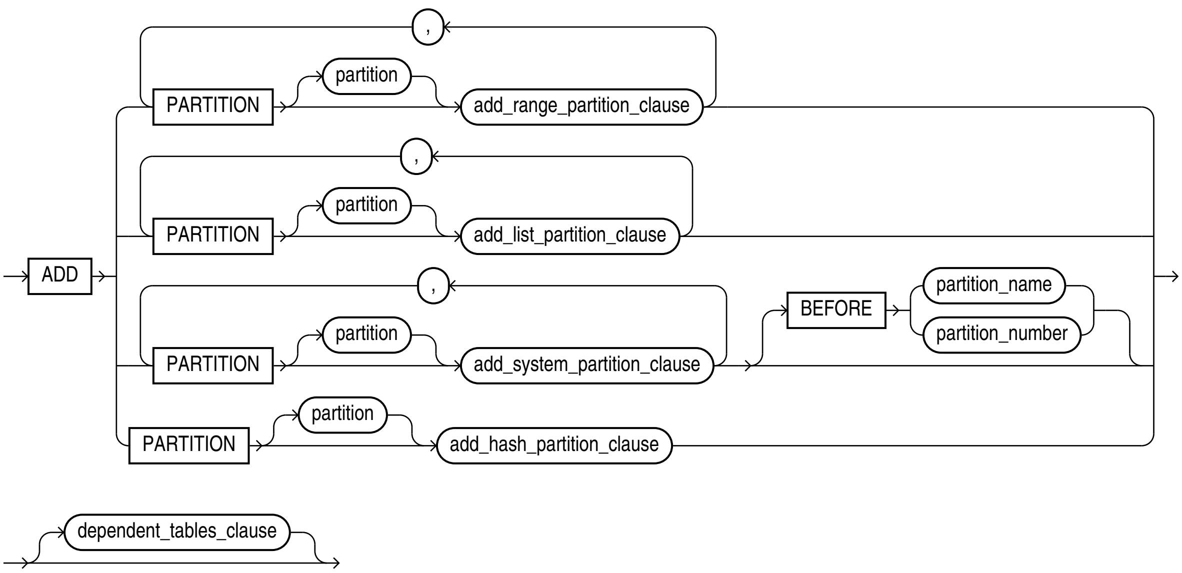


***move_table_subpartition*::=**



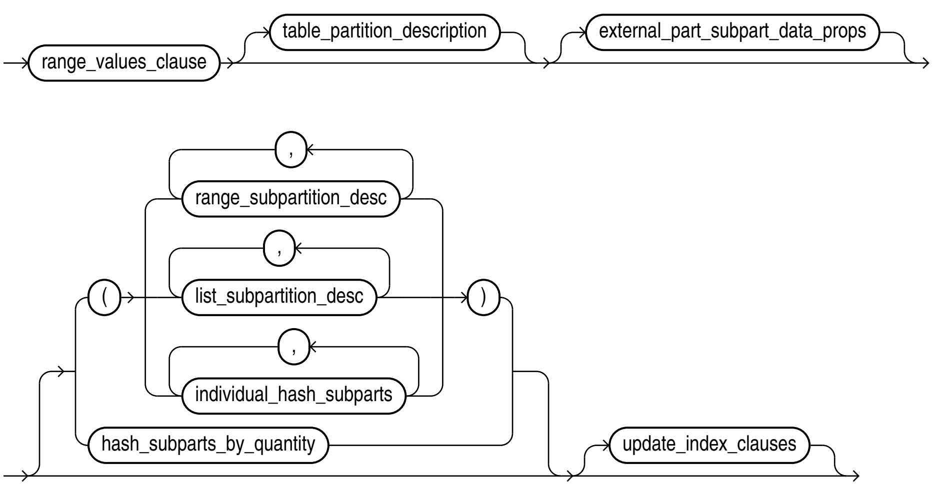
(*subpartition_extended_name*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *update_index_clauses*::=, *filter_condition*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

***add_table_partition*::=**



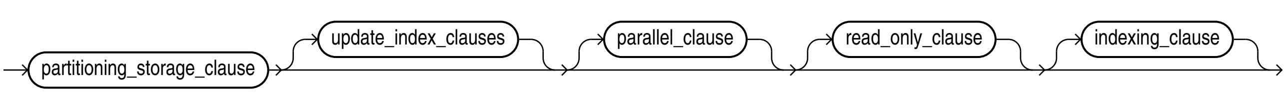
(*add_range_partition_clause*::=, *add_list_partition_clause*::=, *add_system_partition_clause*::=, *add_hash_partition_clause*::=, *dependent_tables_clause*:=)

***add_range_partition_clause*::=**



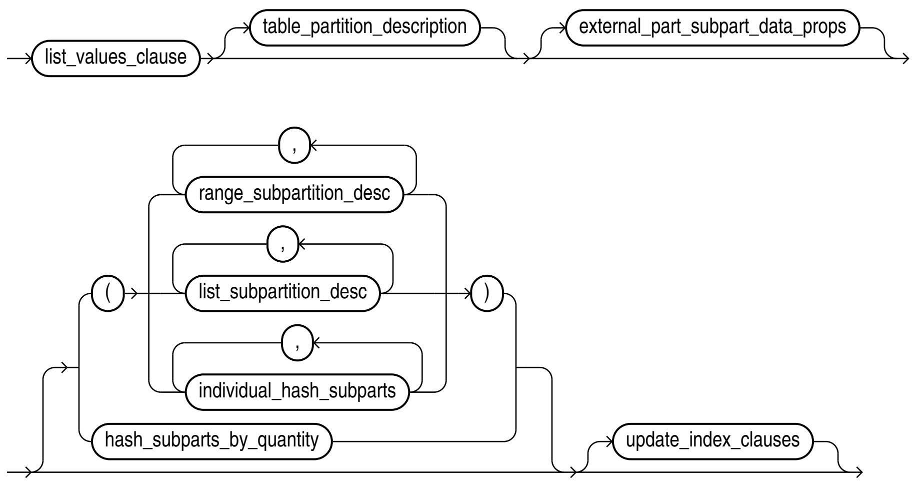
(*range_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=, *update_index_clauses*::=)

***add_hash_partition_clause*::=**



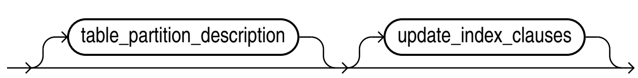
(*partitioning_storage_clause*::=, *update_index_clauses*::=, *parallel_clause*::=, *read_only_clause*::=, *indexing_clause*::=)

***add_list_partition_clause*::=**



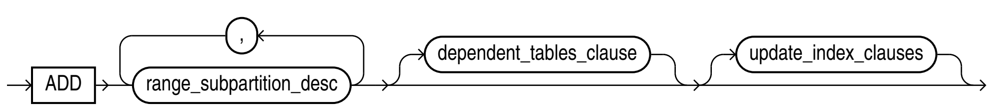
(*list_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=, *update_index_clauses*::=)

***add_system_partition_clause*::=**



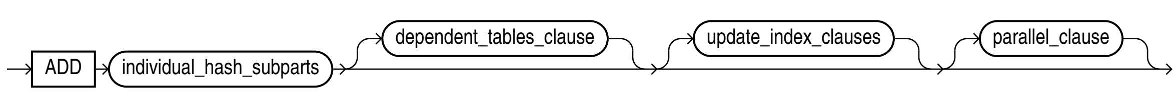
(*table_partition_description*::=, *update_index_clauses*::=)

***add_range_subpartition*::=**



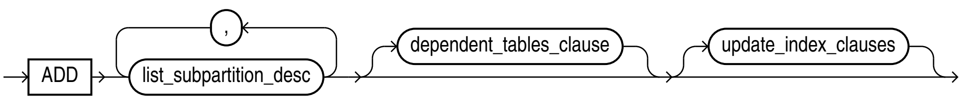
(*range_subpartition_desc*::=, *dependent_tables_clause*:=, *update_index_clauses*::=)

***add_hash_subpartition*::=**



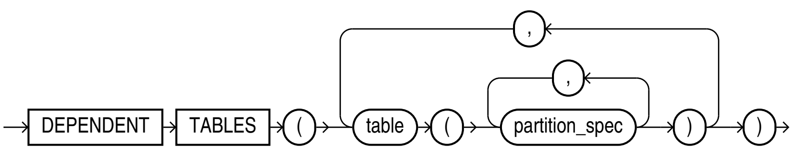
(*individual_hash_subparts*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=)

***add_list_subpartition*::=**



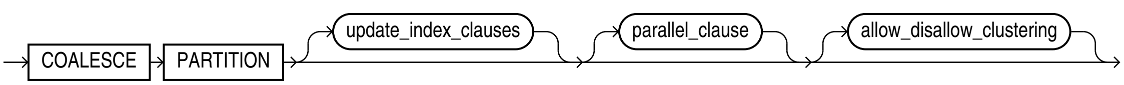
(*list_subpartition_desc*::=, *dependent_tables_clause*:=, *update_index_clauses*::=)

***dependent_tables_clause*:=**



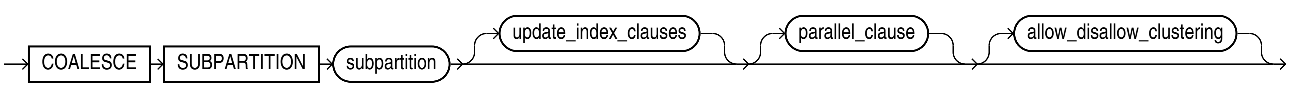
(*partition_spec*::=)

***coalesce_table_partition*::=**



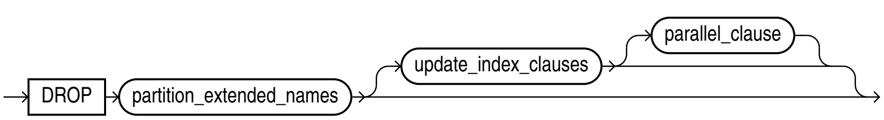
(*update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

***coalesce_table_subpartition*::=**



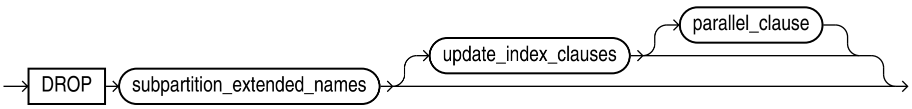
(*update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

***drop_table_partition*::=**



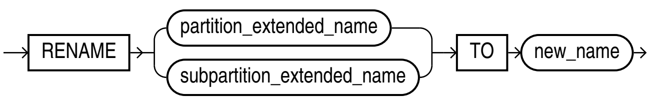
(*partition_extended_names*::=, *update_index_clauses*::=, *parallel_clause*::=)

***drop_table_subpartition*::=**



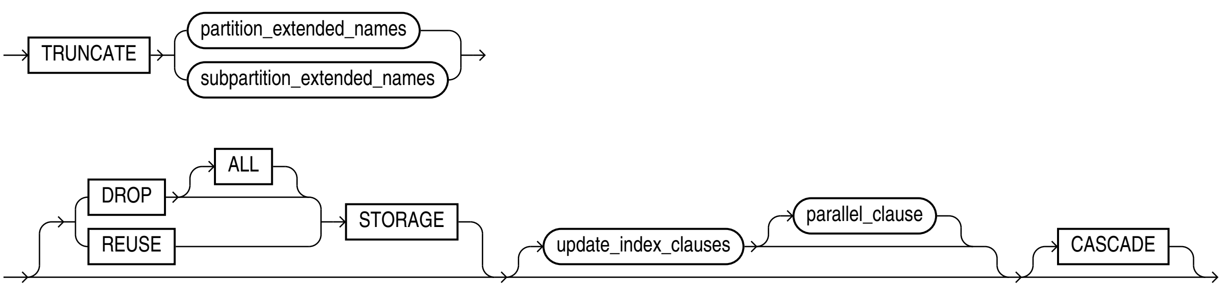
(*subpartition_extended_names*::=, *update_index_clauses*::=, *parallel_clause*::=)

***rename_partition_subpart*::=**



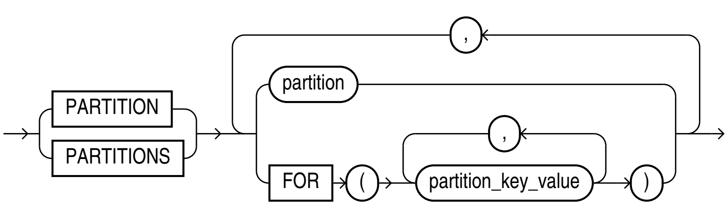
(*partition_extended_name*::=, *subpartition_extended_name*::=)

***truncate_partition_subpart*::=**

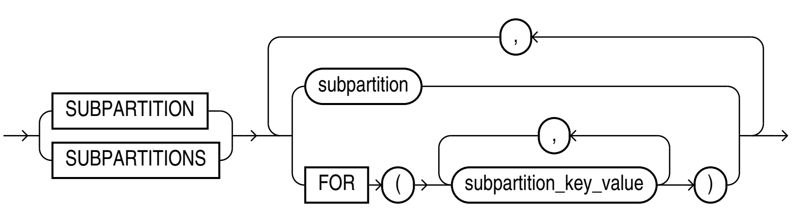


(*partition_extended_names*::=, *subpartition_extended_names*::=, *update_index_clauses*::=, *parallel_clause*::=)

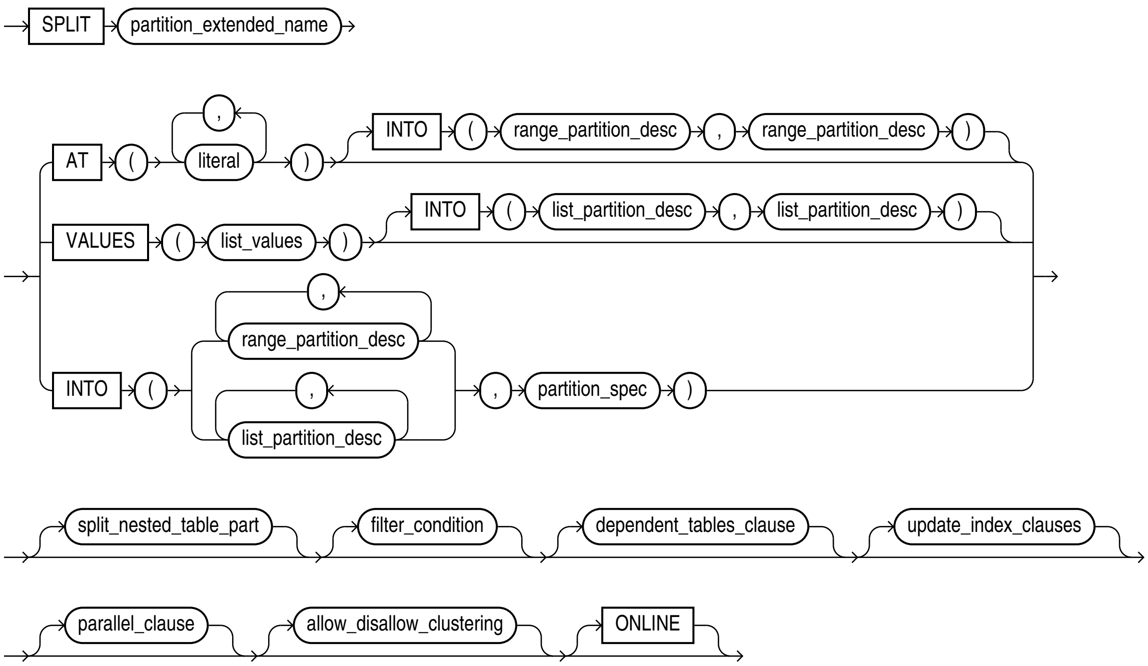
***partition_extended_names*::=**



***subpartition_extended_names*::=**

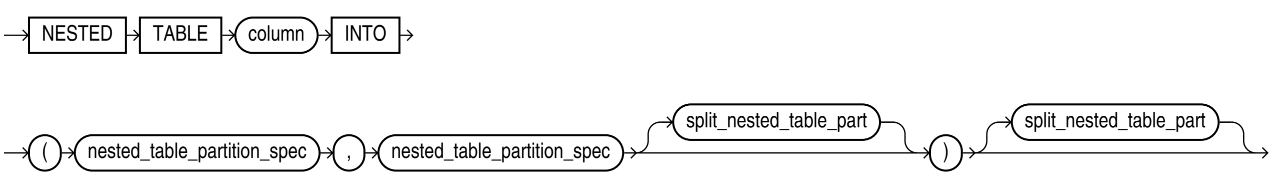


***split_table_partition*::=**

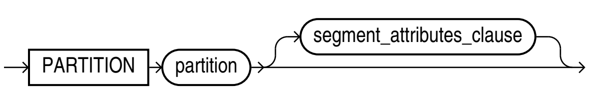


(*partition_extended_name*::=, *range_partition_desc*::=, *list_values*::=, *list_partition_desc*::=, *partition_spec*::=, *split_nested_table_part*::=, *filter_condition*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=)

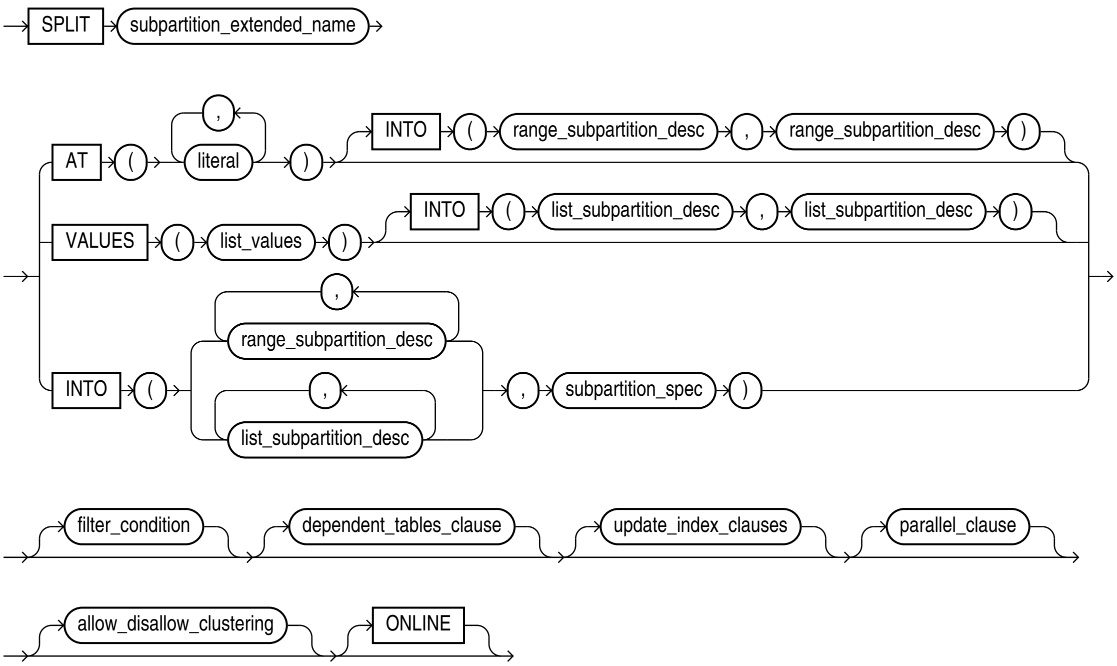
***split_nested_table_part*::=**



***nested_table_partition_spec*::=**

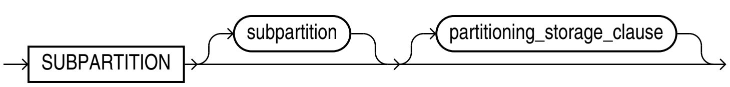


***split_table_subpartition*::=**

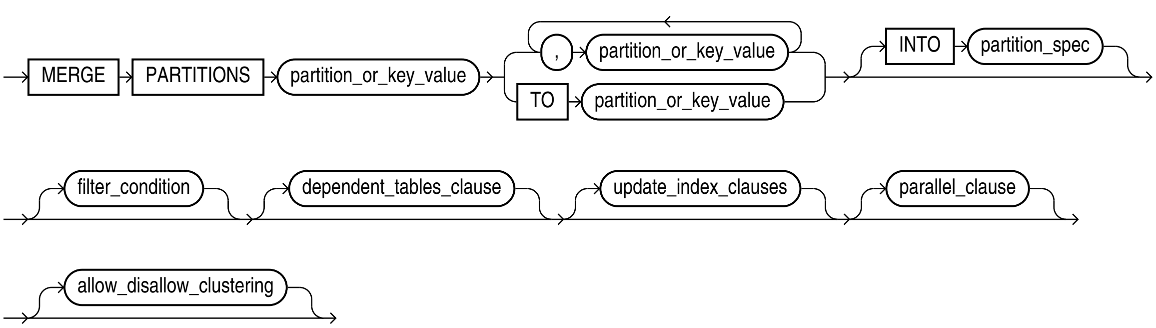


(*subpartition_extended_name*::=, *range_subpartition_desc*::=, *list_values*::=, *list_subpartition_desc*::=, *subpartition_spec*::=, *filter_condition*::=, *dependent_tables_clause*:=, *update_index_clauses*::=, *parallel_clause*::=, *allow_disallow_clustering*::=

***subpartition_spec*::=**

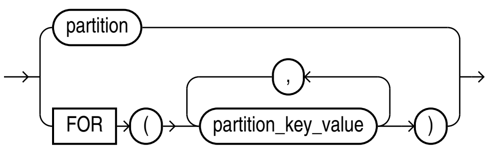


***merge_table_partitions*::=**

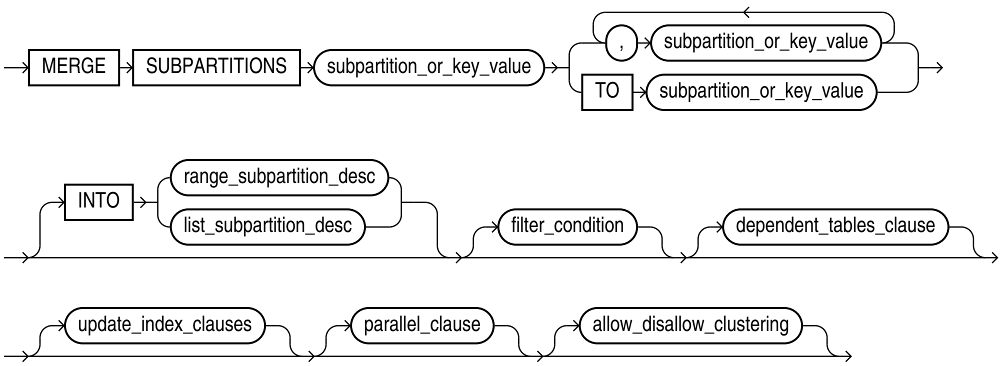


(*partition_or_key_value::=*, *partition_spec::=*, *filter_condition::=*, *dependent_tables_clause:=*, *update_index_clauses::=*, *parallel_clause::=*, *allow_disallow_clustering::=*)

***partition_or_key_value*::=**

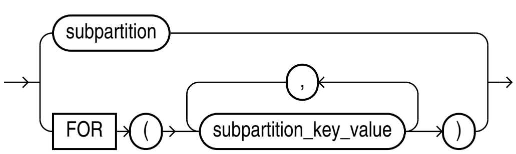


***merge_table_subpartitions*::=**

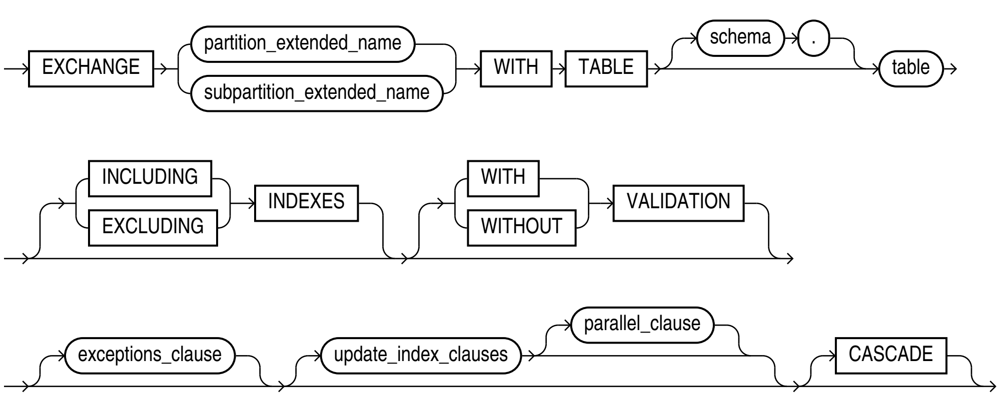


(*subpartition_or_key_value::=*, *range_subpartition_desc::=*, *list_subpartition_desc::=*, *filter_condition::=*, *dependent_tables_clause:=*, *update_index_clauses::=*, *parallel_clause::=*, *allow_disallow_clustering::=*)

***subpartition_or_key_value*::=**

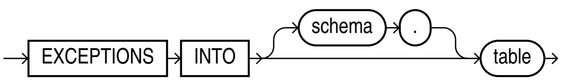


***exchange_partition_subpart*::=**

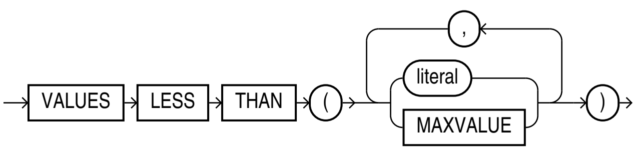


(*partition_extended_name*::=, *subpartition_extended_name*::=, *exceptions_clause*::=, *update_index_clauses*::=, *parallel_clause*::=)

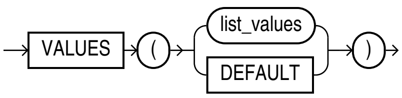
***exceptions_clause*::=**



***range_values_clause*::=**

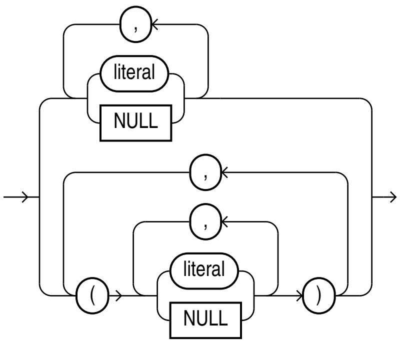


***list_values_clause*::=**

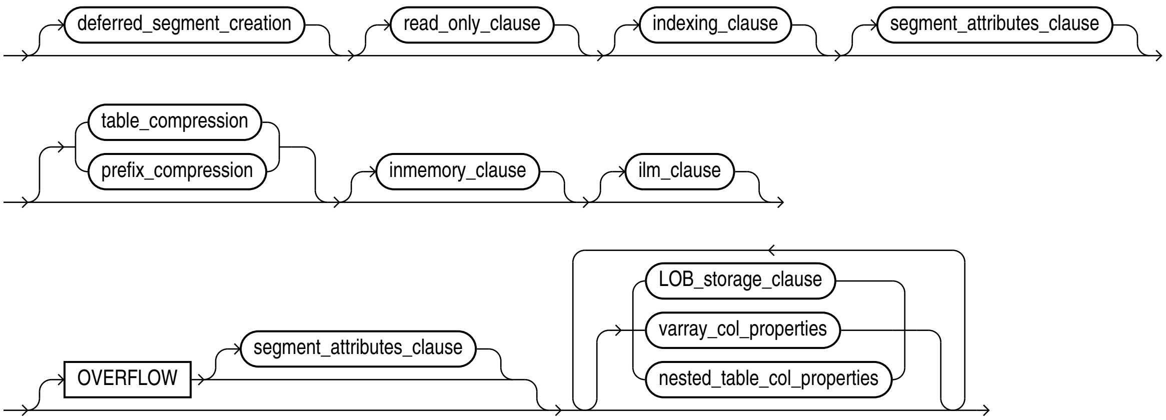


(*list_values*::=)

***list_values*::=**

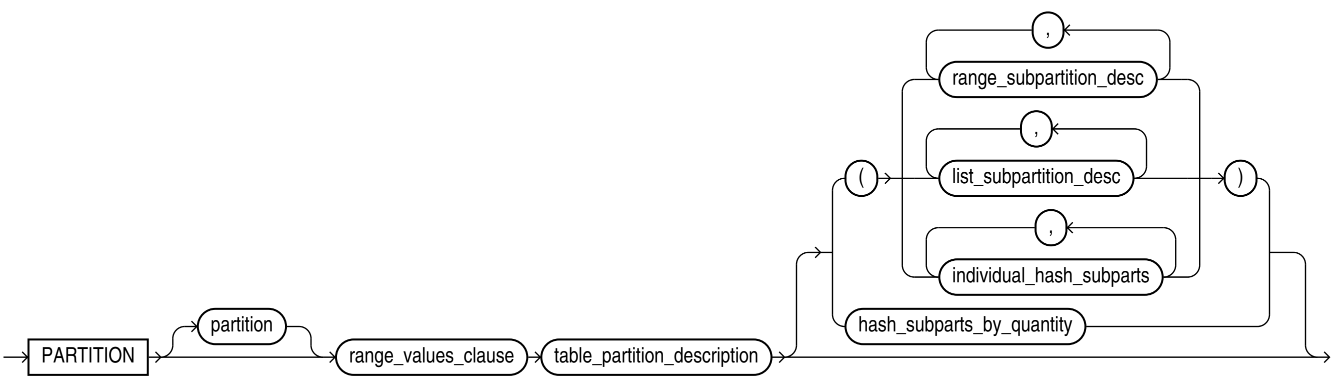


***table_partition_description*::=**



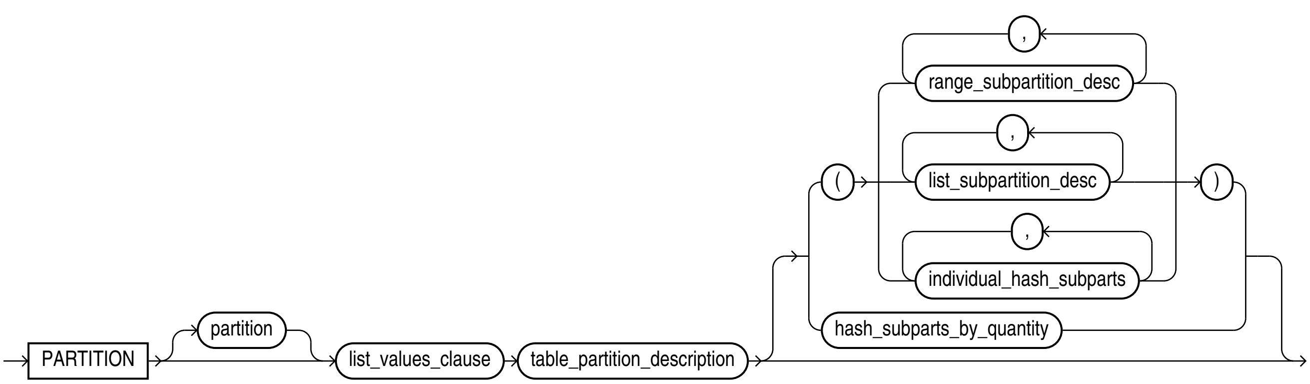
(*deferred_segment_creation*::= , *read_only_clause*::=, *indexing_clause*::=, *segment_attributes_clause*::=, *table_compression*::=, *prefix_compression*::=, *inmemory_clause*::=, *LOB_storage_clause*::=, *varray_col_properties*::=)

***range_partition_desc*::=**



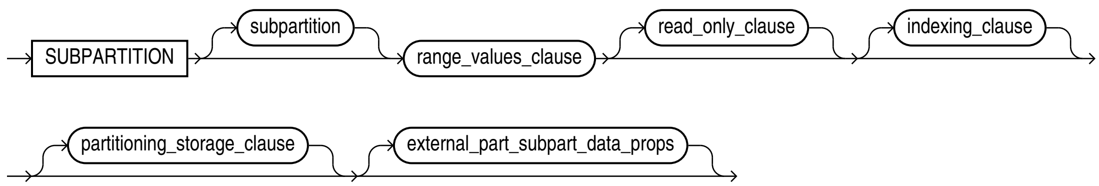
(*range_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=)

***list_partition_desc*::=**



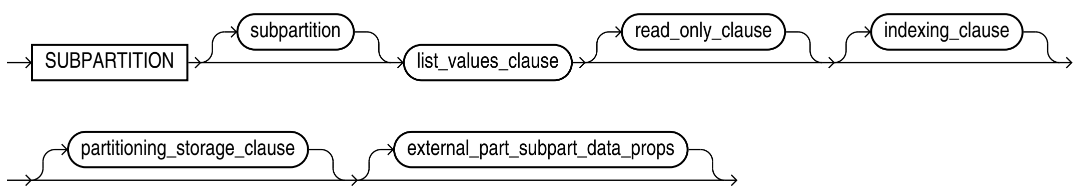
(*list_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=)

***range_subpartition_desc*::=**



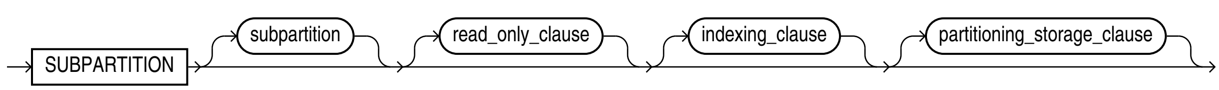
(*range_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***list_subpartition_desc*::=**



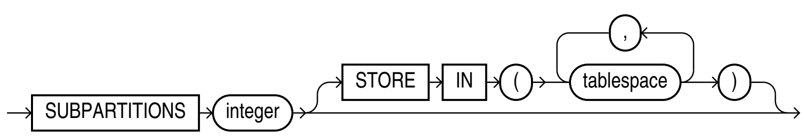
(*list_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***individual_hash_subparts*::=**

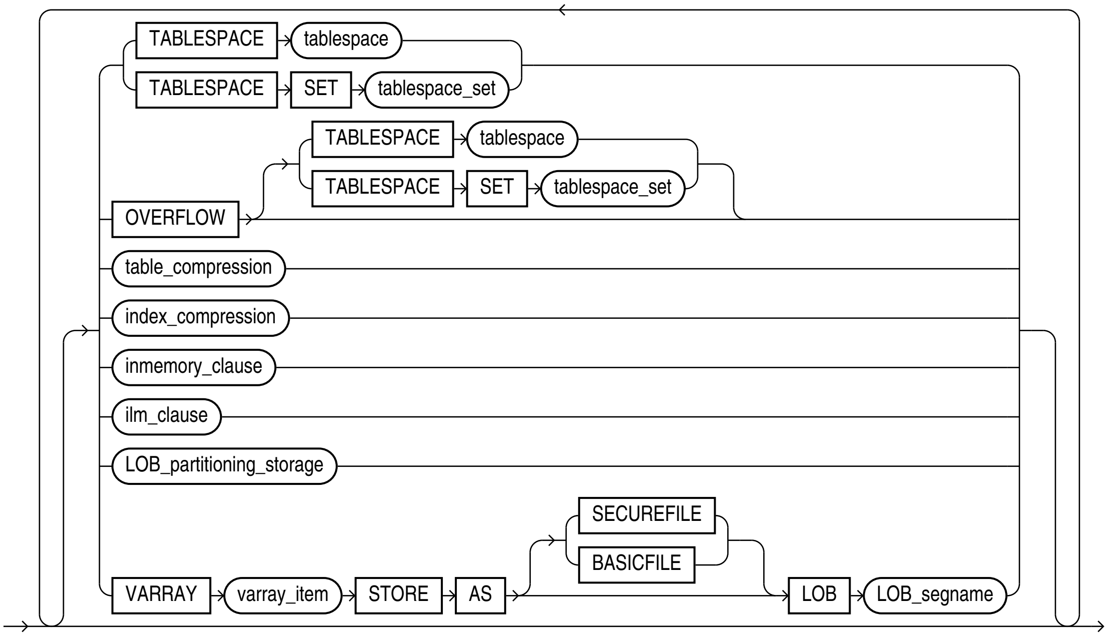


(*read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=)

***hash_subparts_by_quantity*::=**

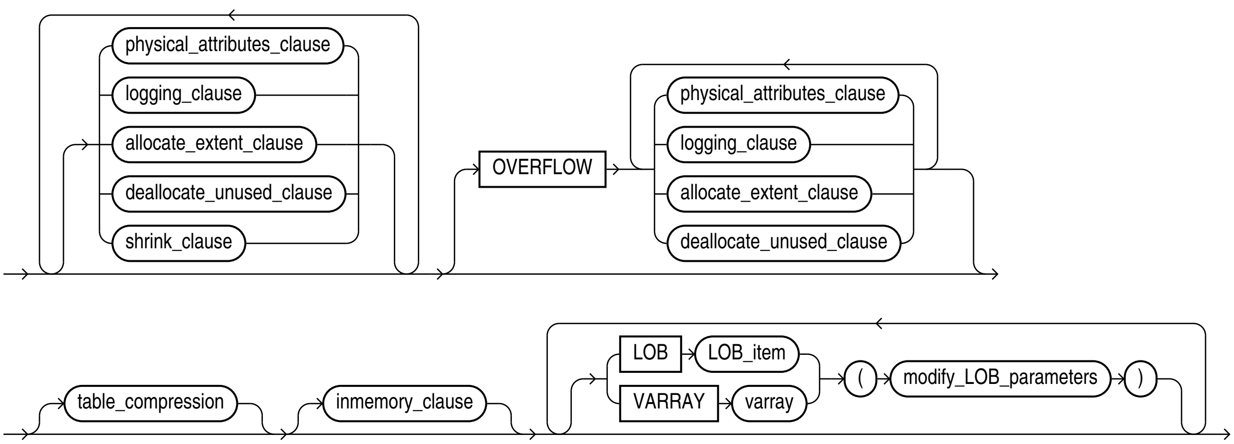


***partitioning_storage_clause*::=**



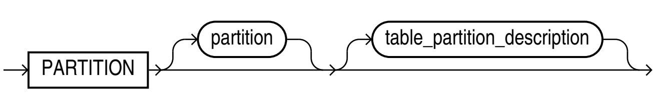
(`TABLESPACE SET`: not supported with `ALTER TABLE`, *table_compression*::=, *index_compression*::=, *inmemory_clause*::=, *LOB_partitioning_storage*::=)

***partition_attributes*::=**



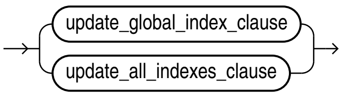
(*physical_attributes_clause*::=, *logging_clause*::=, *allocate_extent_clause*::=, *deallocate_unused_clause*::=, *shrink_clause*::=, *table_compression*::=, *inmemory_clause*::=, *modify_LOB_parameters*::=)

***partition_spec*::=**



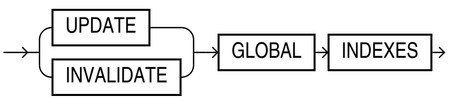
(table_partition_description::=)

***update_index_clauses*::=**

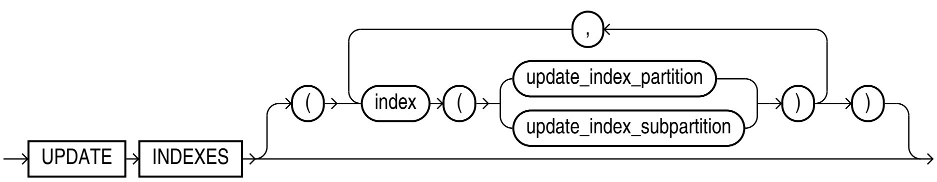


(update_global_index_clause::=, update_all_indexes_clause::=)

***update_global_index_clause*::=**

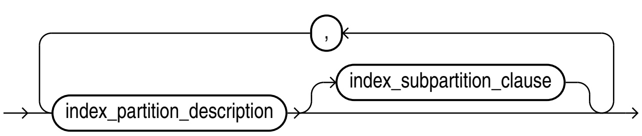


***update_all_indexes_clause*::=**



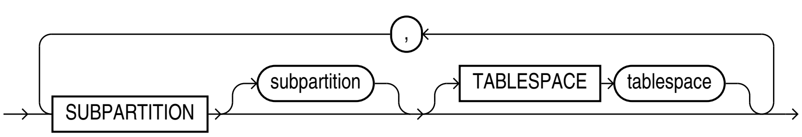
(update_index_partition::=, update_index_subpartition::=)

***update_index_partition*::=**

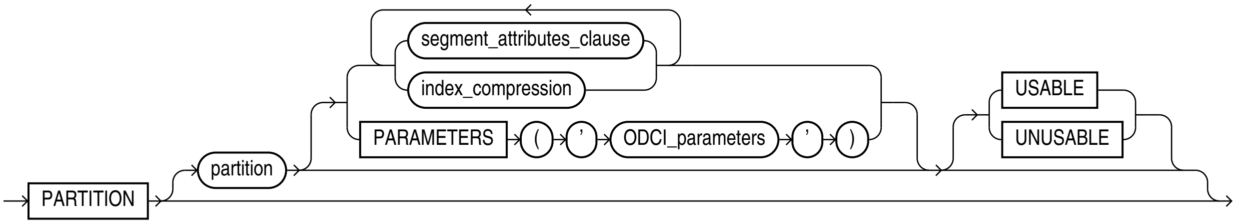


(index_partition_description::=, index_subpartition_clause::=)

***update_index_subpartition*::=**

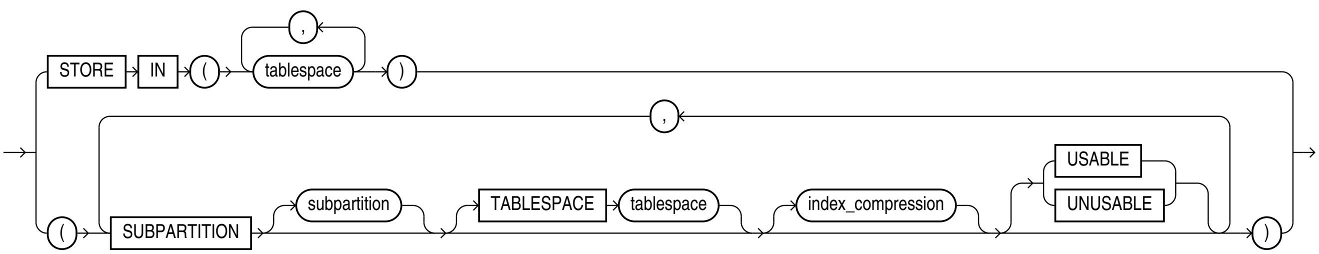


***index_partition_description*::=**



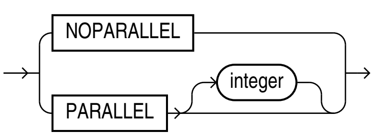
(*segment_attributes_clause::=*, *index_compression::=*)

***index_subpartition_clause*::=**

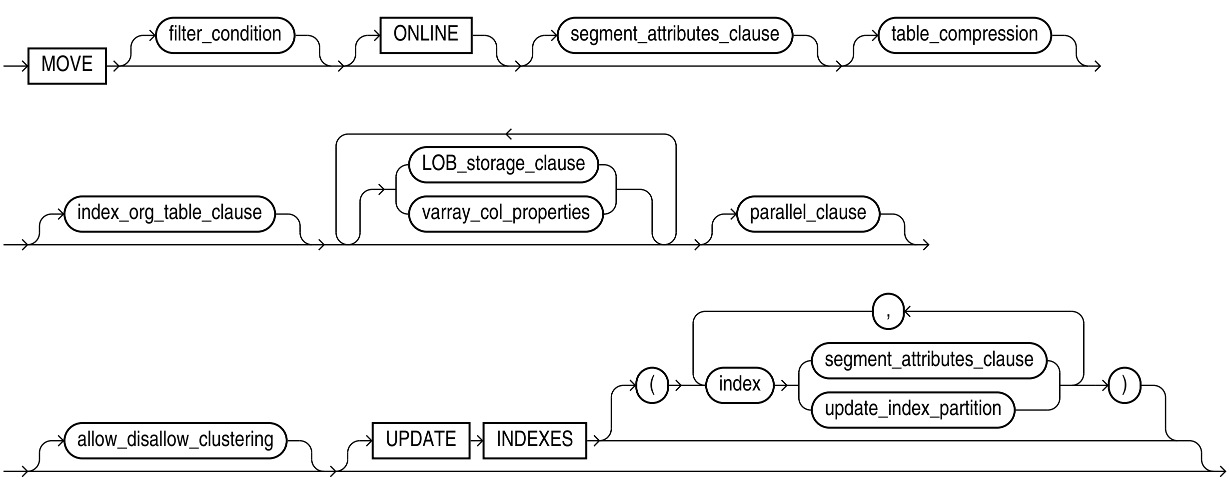


(*index_compression::=*)

***parallel_clause*::=**

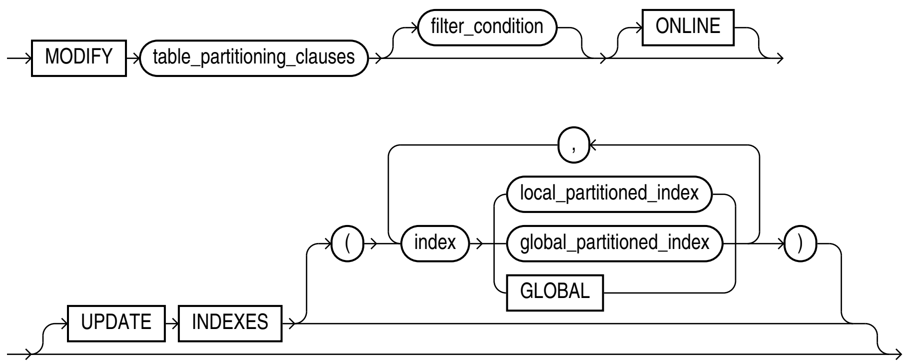


***move_table_clause*::=**

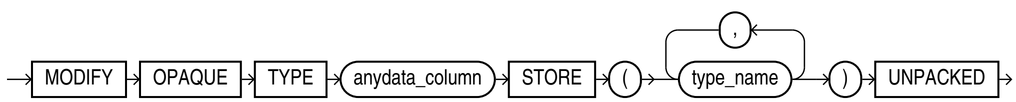


(*filter_condition*::=, *segment_attributes_clause*::=, *table_compression*::=, *index_org_table_clause*::=, *LOB_storage_clause*::=, *varray_col_properties*::=, *parallel_clause*::=, *allow_disallow_clustering*::=, *update_index_partition*::=)

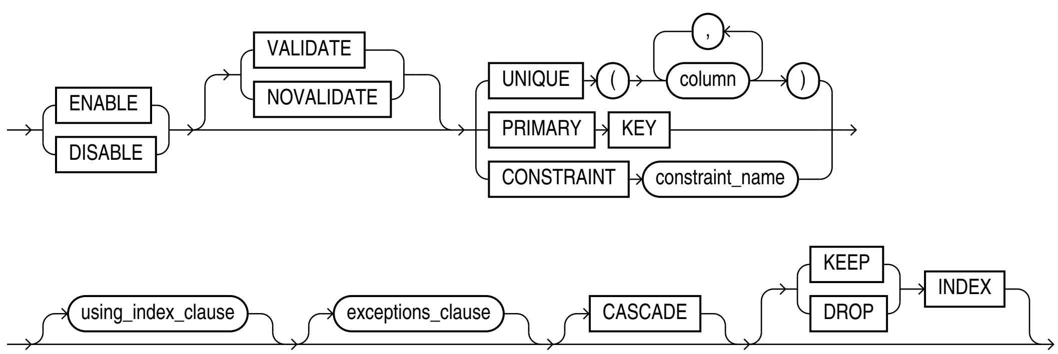
***modify_to_partitioned*::=**



***modify_opaque_type*::=**

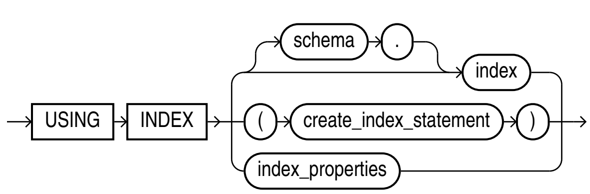


***enable_disable_clause*::=**



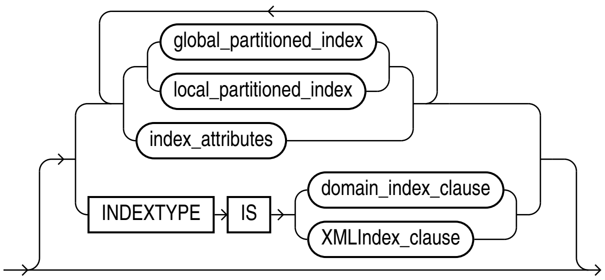
(*using_index_clause*::=, *exceptions_clause*::=,)

***using_index_clause*::=**



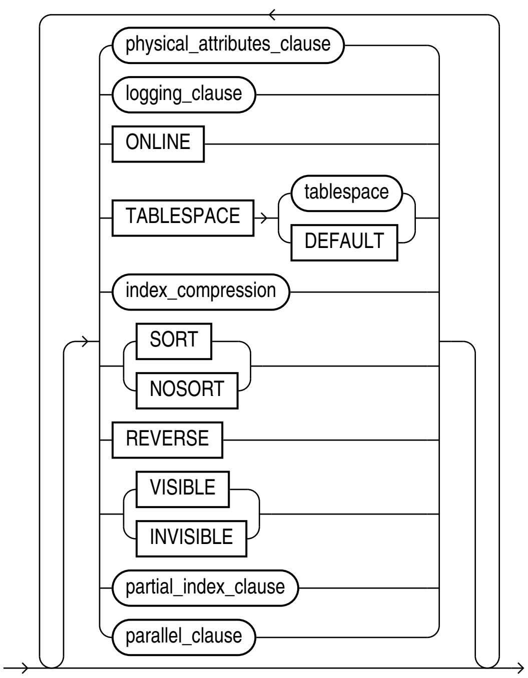
(*create_index*::=, *index_properties*::=)

***index_properties*::=**



(*global_partitioned_index*::=, *local_partitioned_index*::=—part of `CREATE INDEX`, *index_attributes*::=, *`domain_index_clause`*: not supported in *`using_index_clause`*)

***index_attributes*::=**



(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, *`partial_index_clause`* and *`parallel_clause`*: not supported in *`using_index_clause`*)

Semantics

Many clauses of the `ALTER TABLE` statement have the same functionality they have in a `CREATE TABLE` statement. For more information on such clauses, see CREATE TABLE.

> **Note:**
>
> Operations performed by the `ALTER TABLE` statement can cause Oracle Database to invalidate procedures and stored functions that access the table. For information on how and when the database invalidates such objects, see *Oracle Database Development Guide*.

schema

Specify the schema containing the table. If you omit *`schema`*, then Oracle Database assumes the table is in your own schema.

table

Specify the name of the table to be altered.

> **Note:**
>
> If you alter a table that is a master table for one or more materialized views, then Oracle Database marks the materialized views `INVALID`. Invalid materialized views cannot be used by query rewrite and cannot be refreshed. For information on revalidating a materialized view, see ALTER MATERIALIZED VIEW .

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on materialized views in general

Restrictions on Altering Temporary Tables

You can modify, drop columns from, or rename a temporary table. However, for a temporary table you cannot:

- Add columns of nested table type. You can add columns of other types.
- Specify referential integrity (foreign key) constraints for an added or modified column.
- Specify the following clauses of the *`LOB_storage_clause`* for an added or modified LOB column: `TABLESPACE`, *`storage_clause`*, *`logging_clause`*, *`allocate_extent_clause`*, or *`deallocate_unused_clause`*.
- Specify the *`physical_attributes_clause`*, *`nested_table_col_properties`*, *`parallel_clause`*, *`allocate_extent_clause`*, *`deallocate_unused_clause`*, or any of the index-organized table clauses.
- Exchange partitions between a partition and a temporary table.
- Specify the *`logging_clause`*.

- Specify MOVE.
- Add an INVISIBLE column or modify an existing column to be INVISIBLE.

Restrictions on Altering External Tables

You can add, drop, or modify the columns of an external table. However, for an external table you cannot:

- Add a LONG, LOB, or object type column or change the data type of an external table column to any of these data types.
- Modify the storage parameters of an external table.
- Specify the *logging_clause*.
- Specify MOVE.
- Add an INVISIBLE column or modify an existing column to be INVISIBLE.

alter_table_properties

Use the *alter_table_clauses* to modify a database table.

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of the PCTFREE, PCTUSED, and INITRANS parameters and storage characteristics. Refer to *physical_attributes_clause* and storage_clause for a full description of these parameters and characteristics.

Restrictions on Altering Table Physical Attributes

Altering physical attributes is subject to the following restrictions:

- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
- If you attempt to alter the storage attributes of tables in locally managed tablespaces, then Oracle Database raises an error. However, if some segments of a partitioned table reside in a locally managed tablespace and other segments reside in a dictionary-managed tablespace, then the database alters the storage attributes of the segments in the dictionary-managed tablespace but does not alter the attributes of the segments in the locally managed tablespace, and does not raise an error.
- For segments with automatic segment-space management, the database ignores attempts to change the PCTUSED setting. If you alter the PCTFREE setting, then you must subsequently run the DBMS_REPAIR.SEGMENT_FIX_STATUS procedure to implement the new setting on blocks already allocated to the segment.

Cautions on Altering Tables Physical Attributes

The values you specify in this clause affect the table as follows:

- For a nonpartitioned table, the values you specify override any values specified for the table at create time.
- For a range-, list-, or hash-partitioned table, the values you specify are the default values for the table and the actual values for every existing partition, overriding any values already set for the partitions. To change default table attributes without overriding existing partition values, use the *modify_table_default_attrs* clause.
- For a composite-partitioned table, the values you specify are the default values for the table and all partitions of the table and the actual values for all subpartitions of the table,

overriding any values already set for the subpartitions. To change default partition attributes without overriding existing subpartition values, use the *modify_table_default_attrs* clause with the FOR PARTITION clause.

logging_clause

Use the *logging_clause* to change the logging attribute of the table. The *logging_clause* specifies whether subsequent ALTER TABLE ... MOVE and ALTER TABLE ... SPLIT operations will be logged or not logged.

When used with the *modify_table_default_attrs* clause, this clause affects the logging attribute of a partitioned table.

> **See Also:**
>
> - *logging_clause* for a full description of this clause
> - *Oracle Database VLDB and Partitioning Guide* for more information about the *logging_clause* and parallel DML

table_compression

The *table_compression* clause is valid only for heap-organized tables. Use this clause to instruct Oracle Database whether to compress data segments to reduce disk and memory use. Refer to the CREATE TABLE *table_compression* for the full semantics of this clause and for information on creating objects with table compression.

> **Note:**
>
> The first time a table is altered in such a way that compressed data will be added, all bitmap indexes and bitmap index partitions on that table must be marked UNUSABLE.

inmemory_table_clause

Use this clause to enable or disable a table or table column for the In-Memory Column Store (IM column store), or to change the In-Memory attributes for a table or table column.

- Specify INMEMORY to enable a table for the IM column store, or to change the *inmemory_attributes* for a table that is already enabled for the IM column store.
- Specify NO INMEMORY to disable a table for the IM column store.
- Specify the *inmemory_column_clause* to enable or disable a table column for the IM column store, or to change the *inmemory_memcompress* setting for a table column. If you specify this clause when the table or partition is disabled for the IM column store, then the column settings will take effect when the table or partition is subsequently enabled for the IM column store. Regardless of whether the table or partition is enabled or disabled for the IM column store, when you specify NO INMEMORY for a column, any previously specified *inmemory_memcompress* setting for

the column is lost. Refer to the *inmemory_column_clause* of CREATE TABLE for the full semantics of this clause.

This *inmemory_table_clause* has the same semantics as the *inmemory_table_clause* of CREATE TABLE, with the following additions:

- When you specify the *inmemory_memcompress* clause to change the data compression method for a table that is already enabled for the IM column store, any columns that were previously assigned a specific data compression method will retain that data compression method. Refer to the *inmemory_memcompress* clause of CREATE TABLE for more information on this clause.
- When you specify the *inmemory_distribute* clause, if you omit one subclause, then its setting remains unchanged. That is, if you specify only the AUTO or BY clause, then the FOR SERVICE setting for the table remains unchanged, and if you specify only the FOR SERVICE clause, then the AUTO or BY setting for the table remains unchanged. If you omit both subclauses and specify only the DISTRIBUTE keyword, then the table is assigned the DISTRIBUTE AUTO setting and its FOR SERVICE setting remains unchanged. Refer to the *inmemory_distribute* clause of CREATE TABLE for more information on this clause.
- When you specify NO INMEMORY to disable a partitioned or nonpartitioned table for the IM column store, any column-level In-Memory settings are lost. If you subsequently enable the table for the IM column store, then all columns will use the In-Memory settings for the table, unless you specify otherwise when enabling the table.
- When you specify NO INMEMORY to disable a partition for the IM column store, the column-level In-Memory settings are retained, even if all partitions in the table are disabled. If you subsequently enable the table or a partition for the IM column store, then the column-level In-Memory settings will go into effect, unless you specify otherwise when enabling the table or partition.
- If a table is currently populated in the IM column store and you change any *inmemory_attribute* of the table other than PRIORITY, then the database evicts the table from the IM column store. The repopulation behavior depends on the PRIORITY setting.

inmemory_clause

Use this clause to enable or disable a table partition for the IM column store, or to change the In-Memory parameters for a table partition. This clause has the same semantics in CREATE TABLE and ALTER TABLE. Refer to the *inmemory_clause* in the documentation on CREATE TABLE for the full semantics of this clause.

ilm_clause

Use this clause to add, delete, enable, or disable Automatic Data Optimization policies for the table.

ADD POLICY

Specify this clause to add a policy for the table.

Use *ilm_policy_clause* to specify the policy. Refer to the *ilm_policy_clause* for the full semantics of this clause

Oracle Database assigns a name to the policy of the form P*n* where *n* is an integer value

{ DELETE | ENABLE | DISABLE } POLICY

Specify these clauses to delete a policy for the table, enable a policy for the table, or disable a policy for the table, respectively.

For *ilm_policy_name*, specify the name of the policy. You can view policy names by querying the POLICY_NAME column of the DBA_ILMPOLICIES view.

{ DELETE_ALL, ENABLE_ALL, DISABLE_ALL }

Specify these clauses to delete all policies for the table, enable all policies for the table, or disable all policies for the table, respectively.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on managing policies for Automatic Data Optimization

ilm_policy_clause

This clause lets you specify an Automatic Data Optimization policy. You can use the *ilm_compression_policy* clause to specify a compression policy, the *ilm_tiering_policy* clause to specify a storage tiering policy, or the *ilm_inmemory_policy* clause to specify an In-Memory Column Store policy.

ilm_compression_policy

Use this clause to specify a compression policy. This type of policy instructs the database to compress data when a specified condition is met. Use the SEGMENT, GROUP, or ROW clause to specify a segment-level, group-level, or row-level compression policy.

table_compression

Use the *table_compression* clause to specify the compression type. This clause applies to segment-level and group-level compression policies.

You must specify a compression type that is higher than the current compression type. The order of compression types from lowest to highest is:

```
NOCOMPRESS
ROW STORE COMPRESS BASIC
ROW STORE COMPRESS ADVANCED
COLUMN STORE COMPRESS FOR QUERY LOW
COLUMN STORE COMPRESS FOR QUERY HIGH
COLUMN STORE COMPRESS FOR ARCHIVE LOW
COLUMN STORE COMPRESS FOR ARCHIVE HIGH
```

Refer to *table_compression* for the full semantics of this clause.

SEGMENT

Specify SEGMENT to create a segment-level compression policy. This type of policy instructs the database to compress table segments when the condition specified in the AFTER clause is met or when the PL/SQL function specified in the ON clause returns TRUE.

GROUP

Specify GROUP to create a group-level compression policy. This type of policy instructs the database to compress the table and its dependent objects, such as indexes and SecureFiles LOBs, when the condition specified in the AFTER clause is met or when the PL/SQL function specified in the ON clause returns TRUE.

ROW

Specify ROW to create a row-level compression policy. This type of policy instructs the database to compress database blocks in which all the rows have not been modified for a specified period of time. When creating a row-level policy, you must specify ROW STORE COMPRESS ADVANCED or COLUMN STORE COMPRESS FOR QUERY compression, and you must specify AFTER *ilm_time_period* OF NO MODIFICATION. Refer to *table_compression* for the full semantics of the ROW STORE COMPRESS ADVANCED and COLUMN STORE COMPRESS FOR QUERY clauses.

AFTER

Use this clause to describe the condition that must be met in order for the policy to take effect. The condition consists of a length of time, specified with the *ilm_time_period* clause, and one of the following condition types:

- OF NO ACCESS: The policy will take effect after table has not been accessed for the specified length of time.
- OF NO MODIFICATION: The policy will take effect after table has not been modified for the specified length of time.
- OF CREATION: The policy will take effect when the specified length of time has passed since table was created.

ilm_time_period

Specify a length of time in days, months, or years after which the condition must be met. For *integer*, specify a positive integer. The DAY and DAYS keywords can be used interchangeably and are provided for semantic clarity. This is also the case for the MONTH and MONTHS keywords, and the YEAR and YEARS keywords.

ON

Use this clause to specify a PL/SQL function that returns a boolean value. For *function_name*, specify the name of the function. The policy will take effect when the function returns TRUE.

ilm_tiering_policy

Use this clause to specify a storage tiering policy. This type of policy instructs the database to migrate data to a specified tablespace, either when a specified condition is met or when data usage reaches a specified limit. Use the SEGMENT or GROUP clause to specify a segment-level or group-level policy. You can migrate data to a read/write tablespace or a read-only tablespace.

TIER TO *tablespace*

Use this clause to migrate data to a read/write *tablespace*.

- If you specify the ON *function* clause, then data will be migrated when function returns TRUE. Refer to the ON clause for the full semantics of this clause.
- If you omit the ON *function* clause, then data will be migrated when data usage of the tablespace quota reaches the percentage defined by TBS_PERCENT_USED. The database

will make a best effort to migrate enough data so that the amount of free space within the tablespace quota reaches the percentage defined by TBS_PERCENT_FREE. Refer to *Oracle Database PL/SQL Packages and Types Reference* for more information on TBS_PERCENT_USED and TBS_PERCENT_FREE, which are constants in the DBMS_ILM_ADMIN package.

TIER TO *tablespace* READ ONLY

Use this clause to migrate data to a read-only tablespace. When migrating data to the tablespace, the database temporarily places the tablespace in read/write mode, migrates the data, and then places the tablespace back in read-only mode.

- If you specify the AFTER clause, then data will be migrated when the specified condition is met. Refer to the AFTER clause for the full semantics of this clause
- If you specify the ON *function* clause, then data will be migrated when function returns TRUE. Refer to the ON clause for the full semantics of this clause.

SEGMENT | GROUP

Specify SEGMENT to create a segment-level storage tiering policy. This type of policy instructs the database to migrate table segments to *tablespace*. Specify GROUP to create a group-level storage tiering policy. This type of policy instructs the database to migrate the table and its dependent objects, such as indexes and SecureFiles LOBs, to *tablespace*. The default is SEGMENT.

ilm_inmemory_policy

Use this clause to specify an In-Memory Column Store (IM column store) policy. This type of policy instructs the database to enable or disable the table for the IM column store, or to change the compression method for the table in the IM column store, when a specified condition is met.

SET INMEMORY

Use this clause to enable the table for the IM column store when the specified condition is met. You can optionally use the *inmemory_attributes* clause to specify how table data will be stored in the IM column store. Refer to *inmemory_attributes* for the full semantics of this clause.

MODIFY INMEMORY

Use this clause to change the compression method for table data stored in the IM column store when the specified condition is met. The table must be enabled for the IM column store.

You must specify a compression method that his higher than the current compression method. The order of compression methods from lowest to highest is:

```
NO INMEMORY
MEMCOMPRESS FOR DML
MEMCOMPRESS FOR QUERY LOW
MEMCOMPRESS FOR QUERY HIGH
MEMCOMPRESS FOR CAPACITY LOW
MEMCOMPRESS FOR CAPACITY HIGH
```

Refer to *inmemory_memcompress* for the full semantics of this clause.

NO INMEMORY

Use this clause to disable the table for the IM column store when the specified condition is met.

SEGMENT

The `SEGMENT` keyword is optional and is provided for semantic clarity. IM column store policies are always segment-level policies.

AFTER | ON

The `AFTER` and `ON` clauses enable you to specify the condition that must be met in order for the IM column store policy to take effect:

- If you specify the `AFTER` clause, then the policy will take effect when the specified condition is met. Refer to the AFTER clause for the full semantics of this clause
- If you specify the `ON` function clause, then the policy will take effect when function returns `TRUE`. Refer to the ON clause for the full semantics of this clause.

> **See Also:**
>
> *Oracle Database In-Memory Guide* for more information on using Automatic Data Optimization policies with the IM column store

supplemental_table_logging

Use the *supplemental_table_logging* clause to add or drop a redo log group or one or more supplementally logged columns in a redo log group.

- In the `ADD` clause, use *supplemental_log_grp_clause* to create named supplemental log group. Use the *supplemental_id_key_clause* to create a system-generated log group.
- On the `DROP` clause, use `GROUP` *log_group* syntax to drop a named supplemental log group and use the *supplemental_id_key_clause* to drop a system-generated log group.

The *supplemental_log_grp_clause* and the *supplemental_id_key_clause* have the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For full information on these clauses, refer to supplemental_log_grp_clause and supplemental_id_key_clause in the documentation on `CREATE TABLE`.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for information on supplemental redo log groups

allocate_extent_clause

Use the *allocate_extent_clause* to explicitly allocate a new extent for the table, the partition or subpartition, the overflow data segment, the LOB data segment, or the LOB index.

Restriction on Allocating Table Extents

You cannot allocate an extent for a temporary table or for a range- or composite-partitioned table.

> **See Also:**
>
> *allocate_extent_clause* for a full description of this clause and "Allocating Extents: Example"

deallocate_unused_clause

deallocate_unused_clause Use the `deallocate_unused_clause` to explicitly deallocate unused space at the end of the table, partition or subpartition, overflow data segment, LOB data segment, or LOB index and make the space available for other segments in the tablespace.

> **See Also:**
>
> *deallocate_unused_clause* for a full description of this clause and "Deallocating Unused Space: Example"

CACHE | NOCACHE

The `CACHE` and `NOCACHE` clauses have the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For complete information on these clauses, refer to "CACHE | NOCACHE | CACHE READS" in the documentation on `CREATE TABLE`. If you omit both of these clauses in an `ALTER TABLE` statement, then the existing value is unchanged.

RESULT_CACHE

The `RESULT_CACHE` clause has the same semantics in `CREATE TABLE` and `ALTER TABLE` statements. For complete information on this clause, refer to "RESULT_CACHE Clause" in the documentation on `CREATE TABLE`. If you omit this clause in an `ALTER TABLE` statement, then the existing setting is unchanged.

upgrade_table_clause

The `upgrade_table_clause` is relevant for object tables and for relational tables with object columns. It lets you instruct Oracle Database to convert the metadata of the target table to conform with the latest version of each referenced type. If table is already valid, then the table metadata remains unchanged.

Restriction on Upgrading Object Tables and Columns

Within this clause, you cannot specify `object_type_col_properties` as a clause of `column_properties`.

INCLUDING DATA

Specify `INCLUDING DATA` if you want Oracle Database to convert the data in the table to the latest type version format. You can define the storage for any new column while upgrading the table by using the *column_properties* and the *LOB_partition_storage* . This is the default.

You can convert data in the table at the time you upgrade the type by specifying `CASCADE INCLUDING TABLE DATA` in the `dependent_handling_clause` of the `ALTER TYPE` statement. See *Oracle Database PL/SQL Language Reference* for information on this clause. For information on whether a table contains data based on an older type version, refer to the `DATA_UPGRADED` column of the `USER_TAB_COLUMNS` data dictionary view.

NOT INCLUDING DATA

Specify `NOT INCLUDING DATA` if you want Oracle Database to leave column data unchanged.

Restriction on NOT INCLUDING DATA

You cannot specify `NOT INCLUDING DATA` if the table contains columns in Oracle8 release 8.0.x image format. To determine whether the table contains such columns, refer to the `V80_FMT_IMAGE` column of the `USER_TAB_COLUMNS` data dictionary view.

> **See Also:**
>
> - *Oracle Database Reference* for information on the data dictionary views
> - ALTER TYPE for information on converting dependent table data when modifying a type upon which the table depends

records_per_block_clause

The *`records_per_block_clause`* lets you specify whether Oracle Database restricts the number of records that can be stored in a block. This clause ensures that any bitmap indexes subsequently created on the table will be as compressed as possible.

Restrictions on Records in a Block

The *`record_per_block_clause`* is subject to the following restrictions:

- You cannot specify either `MINIMIZE` or `NOMINIMIZE` if a bitmap index has already been defined on table. You must first drop the bitmap index.
- You cannot specify this clause for an index-organized table or a nested table.

MINIMIZE

Specify `MINIMIZE` to instruct Oracle Database to calculate the largest number of records in any block in the table and to limit future inserts so that no block can contain more than that number of records.

Oracle recommends that a representative set of data already exist in the table before you specify `MINIMIZE`. If you are using table compression (see *table_compression*), then a representative set of compressed data should already exist in the table.

Restriction on MINIMIZE

You cannot specify `MINIMIZE` for an empty table.

NOMINIMIZE

Specify `NOMINIMIZE` to disable the `MINIMIZE` feature. This is the default.

row_movement_clause

You cannot disable row movement in a reference-partitioned table unless row movement is also disabled in the parent table. Otherwise, this clause has the same semantics in `CREATE`

TABLE and ALTER TABLE statements. For complete information on these clauses, refer to *row_movement_clause* in the documentation on CREATE TABLE.

flashback_archive_clause

You must have the FLASHBACK ARCHIVE object privilege on the specified flashback data archive to specify this clause. Use this clause to enable or disable historical tracking for the table.

- Specify FLASHBACK ARCHIVE to enable tracking for the table. You can specify *flashback_archive* to designate a particular flashback data archive for this table. The flashback data archive you specify much already exist.

  If you omit the archive name, then the database uses the default flashback data archive designated for the system. If no default flashback data archive has been designated for the system, then you must specify *flashback_archive*.

  You cannot specify FLASHBACK ARCHIVE to *change* the flashback data archive for this table. Instead you must first issue an ALTER TABLE statement with the NO FLASHBACK ARCHIVE clause and then issue an ALTER TABLE statement with the FLASHBACK ARCHIVE clause.
- Specify NO FLASHBACK ARCHIVE to disable tracking for the table.

> **See Also:**
>
> The CREATE TABLE *flashback_archive_clause* for information on creating a table with tracking enabled and CREATE FLASHBACK ARCHIVE for information on creating default flashback data archives

RENAME TO

Use the RENAME clause to rename *table* to *new_table_name*.

Using this clause invalidates any dependent materialized views. For more information on materialized views, see CREATE MATERIALIZED VIEW and *Oracle Database Data Warehousing Guide.*

If a domain index is defined on the table, then the database invokes the ODCIIndexAlter() method with the RENAME option. This operation establishes correspondence between the indextype metadata and the base table.

Restriction on Renaming Tables

You cannot rename a sharded table or a duplicated table.

shrink_clause

The shrink clause lets you manually shrink space in a table, index-organized table or its overflow segment, index, partition, subpartition, LOB segment, materialized view, or materialized view log. This clause is valid only for segments in tablespaces with automatic segment management. By default, Oracle Database compacts the segment, adjusts the high water mark, and releases the recuperated space immediately.

Compacting the segment requires row movement. Therefore, you must enable row movement for the object you want to shrink before specifying this clause. Further, if your application has any rowid-based triggers, you should disable them before issuing this clause.

> **Note:**
>
> Do not attempt to enable row movement for an index-organized table before specifying the *shrink_clause*. The ROWID of an index-organized table is its primary key, which never changes. Therefore, row movement is neither relevant nor valid for such tables.

COMPACT

If you specify COMPACT, then Oracle Database only defragments the segment space and compacts the table rows for subsequent release. The database does not readjust the high water mark and does not release the space immediately. You must issue another ALTER TABLE ... SHRINK SPACE statement later to complete the operation. This clause is useful if you want to accomplish the shrink operation in two shorter steps rather than one longer step.

For an index or index-organized table, specifying ALTER [INDEX | TABLE] ... SHRINK SPACE COMPACT is equivalent to specifying ALTER [INDEX | TABLE ... COALESCE. The *shrink_clause* can be cascaded (refer to the CASCADE clause, which follows) and compacts the segment more densely than does a coalesce operation, which can improve performance. However, if you do not want to release the unused space, then you can use the appropriate COALESCE clause.

CASCADE

If you specify CASCADE, then Oracle Database performs the same operations on all dependent objects of *table*, including secondary indexes on index-organized tables.

Restrictions on the *shrink_clause*

The *shrink_clause* is subject to the following restrictions:

- You cannot combine this clause with any other clauses in the same ALTER TABLE statement.

  You cannot specify this clause for a cluster, a clustered table, or any object with a LONG column.
- Segment shrink is not supported for tables with function-based indexes, domain indexes, or bitmap join indexes.
- This clause does not shrink mapping tables of index-organized tables, even if you specify CASCADE.
- You can specify this clause for a table with Advanced Row Compression enabled (ROW STORE COMPRESS ADVANCED). You cannot specify this clause for a table with any other type of table compression enabled.
- You cannot shrink a table that is the master table of an ON COMMIT materialized view. Rowid materialized views must be rebuilt after the shrink operation.

READ ONLY | READ WRITE

Specify READ ONLY to put the table in read-only mode. When the table is in READ ONLY mode, you cannot issue any DML statements that affect the table or any SELECT ... FOR UPDATE statements. You can issue DDL statements as long as they do not modify any table data. Operations on indexes associated with the table are allowed when the table is in READ ONLY mode. See *Oracle Database Administrator's Guide* for the complete list of operations that are allowed and disallowed on read-only tables.

Specify READ WRITE to return a read-only table to read/write mode.

REKEY *encryption_spec*

Use the REKEY clause to generate a new encryption key or to switch between different algorithms. This operation returns only after all encrypted columns in the table, including LOB columns, have been reencrypted.

DEFAULT COLLATION

This clause lets you change the default collation for the table. For *collation_name*, specify a valid named collation or pseudo-collation.

The new default collation for the table is assigned to columns of a character data type that are subsequently added to the table with an ALTER TABLE ADD statement or modified from a non-character data type with an ALTER TABLE MODIFY statement. The collations for existing columns in the table are not changed. Refer to the DEFAULT COLLATION clause of CREATE TABLE for the full semantics of this clause.

[NO] ROW ARCHIVAL

Specify this clause to enable or disable *table* for row archival.

- Specify ROW ARCHIVAL to enable *table* for row archival. A hidden column ORA_ARCHIVE_STATE is created in the table. If the table is already populated with data, then the value of ORA_ARCHIVE_STATE is set to 0 for each existing row in the table. You can subsequently use the UPDATE statement to set the value of ORA_ARCHIVE_STATE to 1 for rows you want to archive.
- Specify NO ROW ARCHIVAL to disable *table* for row archival. The hidden column ORA_ARCHIVE_STATE is dropped from the table.

Restrictions on [NO] ROW ARCHIVAL

The following restrictions apply to this clause:

- You cannot specify the ROW ARCHIVAL clause for a table that already contains a column named ORA_ARCHIVE_STATE.
- You cannot specify the NO ROW ARCHIVAL clause for tables owned by SYS.

See Also:

- The CREATE TABLE ROW ARCHIVAL clause for the full semantics of this clause
- *Oracle Database VLDB and Partitioning Guide* for more information on In-Database Archiving

attribute_clustering_clause

Use the ADD *attribute_clustering_clause* to enable the table for attribute clustering. The *attribute_clustering_clause* has the same semantics for ALTER TABLE and CREATE TABLE. Refer to the *attribute_clustering_clause* in the documentation on CREATE TABLE.

MODIFY CLUSTERING

Use this clause to allow or disallow attribute clustering for the table during direct-path insert operations or data movement operations. The table must be enabled for attribute clustering. The *clustering_when* clause and the *zonemap_clause* have the same semantics for ALTER TABLE and CREATE TABLE. Refer to the clustering_when clause and the zonemap_clause in the documentation on CREATE TABLE.

DROP CLUSTERING

Use this clause to disable the table for attribute clustering.

If a zone map on the table was created using the WITH MATERIALIZED ZONEMAP clause of CREATE TABLE or ALTER TABLE, then the zone map will be dropped. If a zone map on the table was created using the CREATE MATERIALIZED ZONEMAP statement, then the zone map will not be dropped.

alter_iot_clauses

index_org_table_clause

This clause lets you alter some of the characteristics of an existing index-organized table. Index-organized tables keep data sorted on the primary key and are therefore best suited for primary-key-based access and manipulation. See *index_org_table_clause* in the context of CREATE TABLE.

> **See Also:**
>
> "Modifying Index-Organized Tables: Examples"

prefix_compression

Use the *prefix_compression* clause to enable prefix compression for the table. Specify COMPRESS to instruct Oracle Database to combine the primary key index blocks of the index-organized table where possible to free blocks for reuse. You can specify this clause with the *parallel_clause*. Specify NOCOMPRESS to disable prefix compression for the table.

PCTTHRESHOLD *integer*

Refer to "PCTTHRESHOLD *integer*" in the documentation on CREATE TABLE.

INCLUDING *column_name*

Refer to "INCLUDING column_name" in the documentation on CREATE TABLE.

overflow_attributes

The *overflow_attributes* let you specify the overflow data segment physical storage and logging attributes to be modified for the index-organized table. Parameter values specified in this clause apply only to the overflow data segment.

> **See Also:**
>
> CREATE TABLE

add_overflow_clause

The *add_overflow_clause* lets you add an overflow data segment to the specified index-organized table. You can also use this clause to explicitly allocate an extent to or deallocate unused space from an existing overflow segment.

Use the `STORE IN` *tablespace* clause to specify tablespace storage for the entire overflow segment. Use the `PARTITION` clause to specify tablespace storage for the segment by partition.

For a partitioned index-organized table:

- If you do not specify `PARTITION`, then Oracle Database automatically allocates an overflow segment for each partition. The physical attributes of these segments are inherited from the table level.
- If you want to specify separate physical attributes for one or more partitions, then you must specify such attributes for *every* partition in the table. You need not specify the name of the partitions, but you must specify their attributes in the order in which they were created.

You can find the order of the partitions by querying the `PARTITION_NAME` and `PARTITION_POSITION` columns of the `USER_IND_PARTITIONS` view.

If you do not specify `TABLESPACE` for a particular partition, then the database uses the tablespace specified for the table. If you do not specify `TABLESPACE` at the table level, then the database uses the tablespace of the partition primary key index segment.

Restrictions on Overflow Attributes

Within the *segment_attributes_clause*:

- You cannot specify the `OPTIMAL` parameter of the *physical_attributes_clause*.
- You cannot specify tablespace storage for the overflow segment using this clause. For a nonpartitioned table, you can use `ALTER TABLE` ... `MOVE` ... `OVERFLOW` to move the segment to a different tablespace. For a partitioned table, use `ALTER TABLE` ... `MODIFY DEFAULT ATTRIBUTES` ... `OVERFLOW` to change the default tablespace of the overflow segment.

Additional restrictions apply if *table* is in a locally managed tablespace, because in such tablespaces several segment attributes are managed automatically by the database.

> **See Also:**
>
> *allocate_extent_clause* and *deallocate_unused_clause* for full descriptions of these clauses of the `add_overflow_clause`

alter_overflow_clause

The `alter_overflow_clause` lets you change the definition of the overflow segment of an existing index-organized table.

The restrictions that apply to the `add_overflow_clause` also apply to the `alter_overflow_clause`.

> **Note:**
>
> When you add a column to an index-organized table, Oracle Database evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified `OVERFLOW`, then the database raises an error and does not execute the `ALTER TABLE` statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.

alter_mapping_table_clauses

The `alter_mapping_table_clauses` is valid only if `table` is index organized and has a mapping table.

allocate_extent_clause

Use the `allocate_extent_clause` to allocate a new extent at the end of the mapping table for the index-organized table. Refer to *allocate_extent_clause* for a full description of this clause.

deallocate_unused_clause

Specify the `deallocate_unused_clause` to deallocate unused space at the end of the mapping table of the index-organized table. Refer to *deallocate_unused_clause* for a full description of this clause.

Oracle Database automatically maintains all other attributes of the mapping table or its partitions.

COALESCE Clause

Specify `COALESCE` to instruct Oracle Database to merge the contents of index blocks of the index the database uses to maintain the index-organized table where possible to free blocks for reuse. Refer to the *shrink_clause* for information on the relationship between these two clauses.

alter_XMLSchema_clause

This clause is valid as part of `alter_table_properties` only if you are modifying an `XMLType` table with `BINARY XML` storage. Refer to XMLSchema_spec in the documentation on `CREATE TABLE` for more information on the `ALLOW` and `DISALLOW` clauses.

column_clauses

Use these clauses to add, drop, or otherwise modify a column.

add_column_clause

The `add_column_clause` lets you add a column to a table.

> **See Also:**
>
> CREATE TABLE for a description of the keywords and parameters of this clause and "Adding a Table Column: Example"

column_definition

Unless otherwise noted in this section, the elements of `column_definition` have the same behavior when adding a column to an existing table as they do when creating a new table. Refer to the*column_definition* clause of `CREATE TABLE` for information.

Restriction on *column_definition*

The `SORT` parameter is valid only when creating a new table. You cannot specify `SORT` in the column_definition of an `ALTER TABLE` ... `ADD` statement.

When you add a column, the initial value of each row for the new column is null, unless you specify the `DEFAULT` clause.

You can add an overflow data segment to each partition of a partitioned index-organized table.

You can add LOB columns to nonpartitioned and partitioned tables. You can specify LOB storage at the table and at the partition or subpartition level.

If you previously created a view with a query that used the `SELECT *` syntax to select all columns from table, and you now add a column to `table`, then the database does not automatically add the new column to the view. To add the new column to the view, re-create the view using the `CREATE VIEW` statement with the `OR REPLACE` clause. Refer to CREATE VIEW for more information.

Restrictions on Adding Columns

The addition of columns is subject to the following restrictions:

- You cannot add a LOB column or an `INVISIBLE` column to a cluster table.
- If you add a LOB column to a hash-partitioned table, then the only attribute you can specify for the new partition is `TABLESPACE`.

- You cannot add a column with a `NOT NULL` constraint if *`table`* has any rows unless you also specify the `DEFAULT` clause.
- If you specify this clause for an index-organized table, then you cannot specify any other clauses in the same statement.
- You cannot add a column to a duplicated table.

DEFAULT

Use the `DEFAULT` clause to specify a default for a new column or a new default for an existing column. Oracle Database assigns this value to the column if a subsequent `INSERT` statement omits a value for the column.

The data type of the expression must match the data type specified for the column. The column must also be large enough to hold this expression.

The `DEFAULT` expression can include any SQL function as long as the function does not return a literal argument, a column reference, or a nested function invocation.

The `DEFAULT` expression can include the sequence pseudocolumns `CURRVAL` and `NEXTVAL`, as long as the sequence exists and you have the privileges necessary to access it. Users who perform subsequent inserts that use the `DEFAULT` expression must have the `INSERT` privilege on the table and the `SELECT` privilege on the sequence. If the sequence is later dropped, then subsequent insert statements where the `DEFAULT` expression is used will result in an error. If you are adding a new column to a table, then the order in which `NEXTVAL` is assigned to each existing row is nondeterministic. If you do not fully qualify the sequence by specifying the sequence owner, for example, `SCOTT.SEQ1`, then Oracle Database will default the sequence owner to be the user who issues the `ALTER TABLE` statement. For example, if user `MARY` adds a column to `SCOTT.TABLE` and refers to a sequence that is not fully qualified, such as `SEQ2`, then the column will use sequence `MARY.SEQ2`. Synonyms on sequences undergo a full name resolution and are stored as the fully qualified sequence in the data dictionary; this is true for public and private synonyms. For example, if user `BETH` adds a column referring to public or private synonym `SYN1` and the synonym refers to `PETER.SEQ7`, then the column will store `PETER.SEQ7` as the default.

If you specify the `DEFAULT` clause for a column, then the default value is stored as metadata but the column itself is not populated with data. However, subsequent queries that specify the new column are rewritten so that the default value is returned in the result set. This optimized behavior is subject to the following restrictions:

- The table cannot have any LOB columns. It cannot be index-organized, temporary, or part of a cluster. It also cannot be a queue table, an object table, or the container table of a materialized view.
- If the table has a Virtual Private Database (VPD) policy on it, then the optimized behavior will not take place unless the user who issues the `ALTER TABLE` ... `ADD` statement has the `EXEMPT ACCESS POLICY` system privilege.
- The column being added cannot be encrypted, and cannot be an object column, nested table column, or a LOB column.
- The `DEFAULT` expression cannot include the sequence pseudocolumns `CURRVAL` or `NEXTVAL`.

If the optimized behavior cannot take place due to the preceding restrictions, then Oracle Database updates each row in the newly created column with the default value. In this case, the database does not fire any `UPDATE` triggers that are defined on the table.

Restrictions on Default Column Values

Default column values are subject to the following restrictions:

- A DEFAULT expression cannot contain references to PL/SQL functions or to other columns, the pseudocolumns LEVEL, PRIOR, and ROWNUM, or date constants that are not fully specified.
- The expression can be of any form except a scalar subquery expression.

ON NULL

If you specify the ON NULL clause, then Oracle Database assigns the DEFAULT column value when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.

When you specify ON NULL, the NOT NULL constraint and NOT DEFERRABLE constraint state are implicitly specified. If you specify an inline constraint that conflicts with NOT NULL and NOT DEFERRABLE, then an error is raised.

> **See Also:**
>
> "Specifying a Default Column Value: Examples"

identity_clause

The *identity_clause* has the same semantics when you add an identity column that it has when you create an identity column. Refer to CREATE TABLE *identity_clause* for more information.

When you add a new identity column to a table, all existing rows are updated using the sequence generator. The order in which a value is assigned to each existing row is nondeterministic.

identity_options

Use the *identity_options* clause to configure the sequence generator. The *identity_options* clause has the same parameters as the CREATE SEQUENCE statement. Refer to CREATE SEQUENCE for a full description of these parameters and characteristics. The exception is START WITH LIMIT VALUE, which is specific to *identity_options* and can only be used with ALTER TABLE MODIFY. Refer to identity_options for more information.

inline_constraint

Use *inline_constraint* to add a constraint to the new column.

inline_ref_constraint

This clause lets you describe a new column of type REF. Refer to *constraint* for syntax and description of this type of constraint, including restrictions.

virtual_column_definition

The *virtual_column_definition* has the same semantics when you add a column that it has when you create a column.

> **See Also:**
>
> The CREATE TABLE *virtual_column_definition* and "Adding a Virtual Table Column: Example" for more information

Restriction on Adding a Virtual Column

You cannot add a virtual column when the SQL expression for the virtual column involves a column on which an Oracle Data Redaction policy is defined.

column_properties

The clauses of *column_properties* determine the storage characteristics of an object type, nested table, varray, or LOB column.

object_type_col_properties

This clause is valid only when you are adding a new object type column or attribute. To modify the properties of an existing object type column, use the *modify_column_clauses*. The semantics of this clause are the same as for CREATE TABLE unless otherwise noted.

Use the *object_type_col_properties* clause to specify storage characteristics for a new object column or attribute or an element of a collection column or attribute.

For complete information on this clause, refer to object_type_col_properties in the documentation on CREATE TABLE.

nested_table_col_properties

The *nested_table_col_properties* clause lets you specify separate storage characteristics for a nested table, which in turn lets you to define the nested table as an index-organized table. You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here. See the CREATE TABLE clause *nested_table_col_properties* for more information about these clauses.)

- For *nested_item*, specify the name of a column (or a top-level attribute of the nested table object type) whose type is a nested table.

  If the nested table is a multilevel collection, and the inner nested table does not have a name, then specify COLUMN_VALUE in place of the *nested_item* name.
- For *storage_table*, specify the name of the table where the rows of *nested_item* reside. The storage table is created in the same schema and the same tablespace as the parent table.

Restrictions on Nested Table Column Properties

Nested table column properties are subject to the following restrictions:

- You cannot specify the *parallel_clause*.
- You cannot specify CLUSTER as part of the *physical_properties* clause.

> **See Also:**
>
> "Nested Tables: Examples"

varray_col_properties

The `varray_col_properties` clause lets you specify separate storage characteristics for the LOB in which a varray will be stored. If you specify this clause, then Oracle Database will always store the varray in a LOB, even if it is small enough to be stored inline. If `varray_item` is a multilevel collection, then the database stores all collection items nested within `varray_item` in the same LOB in which `varray_item` is stored.

Restriction on Varray Column Properties

You cannot specify `TABLESPACE` as part of `LOB_parameters` for a varray column. The LOB tablespace for a varray defaults to the tablespace of the containing table.

out_of_line_part_storage

This clause lets you specify storage attributes the newly added column for each partition or subpartition in a partitioned table. For any partition or subpartition you do not name in this clause, the storage attributes for the new column are the same as those specified in the `nested_table_col_properties` at the table level.

LOB_storage_clause

Use the `LOB_storage_clause` to specify the LOB storage characteristics for a newly added LOB column, LOB partition, or LOB subpartition, or when you are converting a `LONG` column into a LOB column. You cannot use this clause to modify an existing LOB. Instead, you must use the *modify_LOB_storage_clause*.

Unless otherwise noted in this section, all LOB parameters, in both the `LOB_storage_clause` and the `modify_LOB_storage_clause`, have the same semantics in an `ALTER TABLE` statement that they have in a `CREATE TABLE` statement. Refer to the `CREATE TABLE` *LOB_storage_clause* for complete information on this clause.

Restriction on LOB Parameters

The only parameter of `LOB_parameters` you can specify for a hash partition or hash subpartition is `TABLESPACE`.

CACHE READS Clause

When you add a new LOB column, you can specify the logging attribute with `CACHE READS`, as you can when defining a LOB column at create time. Refer to the `CREATE TABLE` clause CACHE READS for full information on this clause.

ENABLE | DISABLE STORAGE IN ROW

You cannot change `STORAGE IN ROW` once it is set. Therefore, you cannot specify this clause as part of the `modify_col_properties` clause. However, you can change this setting when adding a new column (*add_column_clause*) or when moving the table (*move_table_clause*). Refer to the `CREATE TABLE` clause ENABLE STORAGE IN ROW for complete information on this clause.

CHUNK *integer*

You use cannot use the *modify_col_properties* clause to change the value of CHUNK after it has been set. If you require a different CHUNK value for a column after it has been created, use ALTER TABLE … MOVE. Refer to the CREATE TABLE clause CHUNK integer for more information.

RETENTION

For BasicFiles LOBs, if the database is in automatic undo mode, then you can specify RETENTION instead of PCTVERSION to instruct Oracle Database to retain old versions of this LOB. This clause overrides any prior setting of PCTVERSION. Refer to the CREATE TABLE clause LOB_retention_clause for a full description of this parameter.

FREEPOOLS *integer*

For BasicFiles LOBs, if the database is in automatic undo mode, then you can use this clause to specify the number of freelist groups for this LOB. This clause overrides any prior setting of FREELIST GROUPS. Refer to the CREATE TABLE clause FREEPOOLS integer for a full description of this parameter. The database ignores this parameter for SecureFiles LOBs.

LOB_partition_storage

You can specify only one list of *LOB_partition_storage* clauses in a single ALTER TABLE statement, and all *LOB_storage_clauses* and *varray_col_properties* clause must precede the list of *LOB_partition_storage* clauses. Refer to the CREATE TABLE clause LOB_partition_storage for full information on this clause, including restrictions.

XMLType_column_properties

Refer to the CREATE TABLE clause *XMLType_column_properties* for a full description of this clause.

See Also:

- *LOB_storage_clause* for information on the *LOB_segname* and *LOB_parameters* clauses
- "XMLType Column Examples" for an example of XMLType columns in object-relational tables and "Using XML in SQL Statements " for an example of creating an XMLSchema
- *Oracle XML DB Developer's Guide* for more information on XMLType columns and tables and on creating an XMLSchema

modify_column_clauses

Use the *modify_column_clauses* to modify the properties of an existing column, the visibility of an existing column, or the substitutability of an existing object type column.

> **See Also:**
>
> "Modifying Table Columns: Examples"

modify_col_properties

Use this clause to modify the properties of the column. Any of the optional parts of the column definition (data type, default value, or constraint) that you omit from this clause remain unchanged.

datatype

You can change the data type of any column if all rows of the column contain nulls. However, if you change the data type of a column in a materialized view container table, then Oracle Database invalidates the corresponding materialized view.

You can omit the data type only if the statement also designates the column as part of the foreign key of a referential integrity constraint. The database automatically assigns the column the same data type as the corresponding column of the referenced key of the referential integrity constraint.

You can always increase the size of a character or raw column or the precision of a numeric column, whether or not all the rows contain nulls. You can reduce the size of a data type of a column as long as the change does not require data to be modified. The database scans existing data and returns an error if data exists that exceeds the new length limit.

When you increase the size of a `VARCHAR2`, `NVARCHAR2`, or `RAW` column to exceed 4000 bytes, Oracle Database performs an in-place length extension and does not migrate the inline storage to external LOB storage. This enables uninterrupted migration of large tables, especially after migration, to leverage extended data types. However, the inline storage of the column will not be preserved during table reorganization operations, such as `CREATE TABLE` ... `AS SELECT`, export, import, or online redefinition. To migrate to the new out-of-line storage of extended data type columns, you must recreate the table using one of the aforementioned methods. The inline storage of the column will be preserved during table or partition movement operations, such as `ALTER TABLE MOVE [[SUB]PARTITION]`, and partition maintenance operations, such as `ALTER TABLE SPLIT [SUB]PARTITION`, `ALTER TABLE MERGE [SUB]PARTITIONS`, and `ALTER TABLE COALESCE [SUB]PARTITIONS`.

> **Note:**
>
> Oracle recommends against excessively increasing the size of a `VARCHAR2`, `NVARCHAR2`, or `RAW` column beyond 4000 bytes for the following reasons:
>
> - Row chaining may occur.
> - Data that is stored inline must be read in its entirety, whether a column is selected or not. Therefore, extended data type columns that are stored inline can have a negative impact on performance.

You can reduce the size of a data type of a column as long as the change does not require data to be modified. The database scans existing data and returns an error if data exists that exceeds the new length limit.

You can change a `DATE` column to a `TIMESTAMP` or `TIMESTAMP WITH LOCAL TIME ZONE` column, and you can change a `TIMESTAMP` or `TIMESTAMP WITH LOCAL TIME ZONE` column to a `DATE` column. The following rules apply:

- When you change a `TIMESTAMP` or `TIMESTAMP WITH LOCAL TIME ZONE` column to a `DATE` column, Oracle Database updates each column value that has nonzero fractional seconds by rounding the value to the nearest second. If, while updating such a value, Oracle Database encounters a minute field greater than or equal to 60 (which can occur in a boundary case when the daylight saving rule switches), then it updates the minute field by subtracting 60 from it.
- After you change a `TIMESTAMP WITH LOCAL TIME ZONE` column to a `DATE` column, the values in the column still represent the local time that they represented in the database time zone. However, the database time zone is no longer associated with the values. When queried in SQL*Plus, the values are no longer automatically adjusted to the session time zone. It is now the responsibility of applications processing the column values to interpret them in a particular time zone.

If the table is empty, then you can increase or decrease the leading field or the fractional second value of a datetime or interval column. If the table is not empty, then you can only increase the leading field or fractional second of a datetime or interval column.

You can use the `TO_LOB` function to change a `LONG` column to a `CLOB` or `NCLOB` column, and a `LONG RAW` column to a `BLOB` column. However, you cannot use the `TO_LOB` function from within a PL/SQL package. Instead use the `TO_CLOB` (character) or `TO_BLOB` (raw) functions.

- The modified LOB column inherits all constraints and triggers that were defined on the original `LONG` column. If you want to change any constraints, then you must do so in a subsequent `ALTER TABLE` statement.
- If any domain indexes are defined on the `LONG` column, then you must drop them before modifying the column to a LOB.
- After the modification, you will have to rebuild all other indexes on all columns of the table.

You can use the `TO_CLOB` (character) function to convert `NCLOB` columns `CLOB` columns.

See Also:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information on `LONG` to LOB migration
- ALTER INDEX for information on dropping and rebuilding indexes

For `CHAR` and `VARCHAR2` columns, you can change the length semantics by specifying `CHAR` (to indicate character semantics for a column that was originally specified in bytes) or `BYTE` (to indicate byte semantics for a column that was originally specified in characters). To learn the length semantics of existing columns, query the `CHAR_USED` column of the `ALL_`, `USER_`, or `DBA_TAB_COLUMNS` data dictionary view.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on byte and character semantics
> - *Oracle Database Reference* for information on the data dictionary views

COLLATE

Use this clause to set or change the data-bound collation for a column. For *column_collation_name*, specify a valid named collation or pseudo-collation. Refer to the DEFAULT COLLATION clause of CREATE TABLE for more information on data-bound collations.

Restrictions on Changing Column Collation

The modification of the column collation is subject to the following restrictions:

- If the column belongs to an index key, then its collation can only be changed:
  - among collations: BINARY, USING_NLS_COMP, USING_NLS_SORT, and USING_NLS_SORT_CS
  - between collations BINARY_CI and USING_NLS_SORT_CI
  - between collations BINARY_AI and USING_NLS_SORT_AI
- If the column belongs to a range- or list-partitioning key, is referenced by a bitmap join index, belongs to the primary key of an index-organized table, or to the key of a domain index, including an Oracle Text index, then its collation can only be changed among the collations BINARY, USING_NLS_COMP, USING_NLS_SORT, and USING_NLS_SORT_CS.
- If the column belongs to an attribute clustering key, then its collation can only be changed between the collations BINARY and USING_NLS_COMP.

> **See Also:**
>
> Modifying the Collation of a Column for Fine-Grained Case-Insensitivity: Example

identity_clause

Use *identity_clause* to modify the properties of an identity column. You cannot specify this clause on a column that is not an identity column. If you do not specify ALWAYS or BY DEFAULT, then the current generation type is retained. Refer to CREATE TABLE *identity_clause* for more information on ALWAYS and BY DEFAULT.

identity_options

Use the *identity_options* clause to configure the sequence generator. The *identity_options* clause has the same parameters as the CREATE SEQUENCE statement. Refer to CREATE SEQUENCE for a full description of these parameters and characteristics. The exceptions are:

- START WITH LIMIT VALUE, which is specific to *identity_options*, can only be used with ALTER TABLE MODIFY. If you specify START WITH LIMIT VALUE, then Oracle Database locks the table and finds the maximum identity column value in the table (for increasing sequences) or the minimum identity column value (for decreasing sequences) and assigns the value as the sequence generator's high water mark. The next value returned by the sequence generator will be the high water mark + INCREMENT BY *integer* for increasing sequences, or the high water mark - INCREMENT BY *integer* for decreasing sequences.
- If you change the value of START WITH, then the default values will be used for all other parameters in this clause unless you specify otherwise.

DROP IDENTITY

Use this clause to remove the identity property from a column, including the sequence generator and NOT NULL and NOT DEFERRABLE constraints. Identity column values in existing rows are not affected.

ENCRYPT *encryption_spec* | DECRYPT

Use this clause to decrypt an encrypted column, to encrypt an unencrypted column, or to change the integrity algorithm or the SALT option of an encrypted column.

When encrypting an existing column, if you specify *encryption_spec*, it must match the encryption specification of any other encrypted columns in the same table. Refer to the CREATE TABLE clause *encryption_spec* for additional information and restrictions on the *encryption_spec*.

If a materialized view log is defined on the table, then Oracle Database encrypts or decrypts in the materialized view log any columns you encrypt or decrypt in this clause.

Restrictions on ENCRYPT *encryption_spec* | DECRYPT

This clause is subject to the following restrictions:

- If the new or existing column is a LOB column, then it must be stored as a SecureFiles LOB, and you cannot specify the SALT option.
- You cannot encrypt or decrypt a column on which a fine-grained audit policy for the UPDATE statement is enabled. However, you can disable the fine-grained audit policy, encrypt or decrypt the column, and then enable the fine-grained audit policy.

> **See Also:**
>
> "Data Encryption: Examples"

inline_constraint

This clause lets you add a constraint to a column you are modifying. To change the state of existing constraints on existing columns, use the *constraint_clauses*.

LOB_storage_clause

The *LOB_storage_clause* is permitted within *modify_col_properties* only if you are converting a LONG column to a LOB column. In this case only, you can specify LOB storage for the column using the *LOB_storage_clause*. However, you can specify only the single column as a *LOB_item*. Default LOB storage attributes are used for any attributes you omit in the *LOB_storage_clause*.

alter_XMLSchema_clause

This clause is valid within *modify_col_properties* only for XMLType tables with BINARY XML storage. Refer to XMLSchema_spec in the documentation on CREATE TABLE for more information on the ALLOW and DISALLOW clauses.

Restrictions on Modifying Column Properties

The modification of column properties is subject to the following restrictions:

- You cannot change the data type of a LOB column.
- You cannot modify a column of a table if a domain index is defined on the column. You must first drop the domain index and then modify the column.
- You cannot modify the data type or length of a column that is part of the partitioning or subpartitioning key of a table or index.
- You can change a CHAR column to VARCHAR2 (or VARCHAR) and a VARCHAR2 (or VARCHAR) column to CHAR only if the BLANK_TRIMMING initialization parameter is set to TRUE and the column size stays the same or increases. If the BLANK_TRIMMING initialization parameter is set to TRUE, then you can also reduce the column size to any size greater than or equal to the maximum trimmed data value.
- You cannot change a LONG or LONG RAW column to a LOB if the table is part of a cluster. If you do change a LONG or LONG RAW column to a LOB, then the only other clauses you can specify in this ALTER TABLE statement are the DEFAULT clause and the *LOB_storage_clause*.
- You can specify the *LOB_storage_clause* as part of *modify_col_properties* only when you are changing a LONG or LONG RAW column to a LOB.
- You cannot specify a column of data type ROWID for an index-organized table, but you can specify a column of type UROWID.
- You cannot change the data type of a column to REF.
- You cannot modify the properties of a column in a duplicated table.

> **See Also:**
>
> ALTER MATERIALIZED VIEW for information on revalidating a materialized view

modify_virtcol_properties

This clause lets you modify a virtual column in the following ways:

- Specify the COLLATE clause to set or change the data-bound collation for a virtual column. For *column_collation_name*, specify a valid named collation or pseudo-collation. Refer to the DEFAULT COLLATION clause of CREATE TABLE for more information on data-bound collations.
- If the virtual column refers to an editioned PL/SQL function, then you can modify the evaluation edition or the unusable editions for a virtual column. The

evaluation_edition_clause and the *unusable_editions_clause* have the same semantics when you modify a virtual column that they have when you create a virtual column. For complete information, refer to evaluation_edition_clause and unusable_editions_clause in the documentation on CREATE TABLE.

Restrictions on Modifying Virtual Columns

The following restrictions apply to modifying virtual columns:

- Specifying the COLLATE clause to set or change the data-bound collation for a virtual column is subject to the restrictions listed in Restrictions on Changing Column Collation.
- If an index is defined on a virtual column and you modify its evaluation edition or unusable editions, then the database will invalidate all indexes on the virtual column. If you attempt to modify any other properties of the virtual column, then an error occurs.

modify_col_visibility

Use this clause to change the visibility of *column*. For complete information, refer to "VISIBLE | INVISIBLE" in the documentation on CREATE TABLE.

Restriction on Modifying Column Visibility

You cannot change a VISIBLE column to INVISIBLE in a table owned by SYS.

modify_col_substitutable

Use this clause to set or change the substitutability of an existing object type column.

The FORCE keyword drops any hidden columns containing typeid information or data for subtype attributes. You must specify FORCE if the column or any attributes of its type are not FINAL.

Restrictions on Modifying Column Substitutability

The modification of column substitutability is subject to the following restrictions:

- You can specify this clause only once in any ALTER TABLE statement.
- You cannot modify the substitutability of a column in an object table if the substitutability of the table itself has been set.
- You cannot specify this clause if the column was created or added using the IS OF TYPE syntax, which limits the range of subtypes permitted in an object column or attribute to a particular subtype. Refer to substitutable_column_clause in the documentation on CREATE TABLE for information on the IS OF TYPE syntax.
- You cannot change a varray column to NOT SUBSTITUTABLE, even by specifying FORCE, if any of its attributes are nested object types that are not FINAL.

drop_column_clause

The *drop_column_clause* lets you free space in the database by dropping columns you no longer need or by marking them to be dropped at a future time when the demand on system resources is less.

- If you drop a nested table column, then its storage table is removed.
- If you drop a LOB column, then the LOB data and its corresponding LOB index segment are removed.

- If you drop a BFILE column, then only the locators stored in that column are removed, not the files referenced by the locators.
- If you drop or mark unused a column defined as an INCLUDING column, then the column stored immediately before this column will become the new INCLUDING column.

SET UNUSED Clause

Specify SET UNUSED to mark one or more columns as unused. For an internal heap-organized table, specifying this clause does not actually remove the target columns from each row in the table. It does not restore the disk space used by these columns. Therefore, the response time is faster than when you execute the DROP clause.

When you specify this clause for a column in an external table, the clause is transparently converted to an ALTER TABLE ... DROP COLUMN statement. The reason for this is that any operation on an external table is a metadata-only operation, so there is no difference in the performance of the two commands.

You can view all tables with columns marked UNUSED in the data dictionary views USER_UNUSED_COL_TABS, DBA_UNUSED_COL_TABS, and ALL_UNUSED_COL_TABS.

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views

Unused columns are treated as if they were dropped, even though their column data remains in the table rows. After a column has been marked UNUSED, you have no access to that column. A SELECT * query will not retrieve data from unused columns. In addition, the names and types of columns marked UNUSED will not be displayed during a DESCRIBE, and you can add to the table a new column with the same name as an unused column.

> **Note:**
>
> Until you actually drop these columns, they continue to count toward the absolute limit of 1000 columns in a single table. However, as with all DDL statements, you cannot roll back the results of this clause. You cannot issue SET USED counterpart to retrieve a column that you have SET UNUSED. Refer to CREATE TABLE for more information on the 1000-column limit.
>
> Also, if you mark a LONG column as UNUSED, then you cannot add another LONG column to the table until you actually drop the unused LONG column.

ONLINE

Specify ONLINE to indicate that DML operations on the table will be allowed while marking the column or columns UNUSED.

Restrictions on Marking Columns Unused

The following restrictions apply to the SET UNUSED clause:

- You cannot specify the ONLINE clause when marking a column with a DEFERRABLE constraint as UNUSED.
- Columns in tables owned by SYS cannot be marked as UNUSED.

DROP Clause

Specify DROP to remove the column descriptor and the data associated with the target column from each row in the table. If you explicitly drop a particular column, then all columns currently marked UNUSED in the target table are dropped at the same time.

When the column data is dropped:

- All indexes defined on any of the target columns are also dropped.
- All constraints that reference a target column are removed.
- If any statistics types are associated with the target columns, then Oracle Database disassociates the statistics from the column with the FORCE option and drops any statistics collected using the statistics type.

> **Note:**
>
> If the target column is a parent key of a nontarget column, or if a check constraint references both the target and nontarget columns, then Oracle Database returns an error and does not drop the column unless you have specified the CASCADE CONSTRAINTS clause. If you have specified that clause, then the database removes all constraints that reference any of the target columns.

> **See Also:**
>
> DISASSOCIATE STATISTICS for more information on disassociating statistics types

DROP UNUSED COLUMNS Clause

Specify DROP UNUSED COLUMNS to remove from the table all columns currently marked as unused. Use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, then the statement returns with no errors.

column

Specify one or more columns to be set as unused or dropped. Use the COLUMN keyword only if you are specifying only one column. If you specify a column list, then it cannot contain duplicates.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS if you want to drop all foreign key constraints that refer to the primary and unique keys defined on the dropped columns as well as all multicolumn constraints defined on the dropped columns. If any constraint is referenced by columns from other tables or remaining columns in the target table, then you must specify `CASCADE CONSTRAINTS`. Otherwise, the statement aborts and an error is returned.

INVALIDATE

The `INVALIDATE` keyword is optional. Oracle Database automatically invalidates all dependent objects, such as views, triggers, and stored program units. Object invalidation is a recursive process. Therefore, all directly dependent and indirectly dependent objects are invalidated. However, only local dependencies are invalidated, because the database manages remote dependencies differently from local dependencies.

An object invalidated by this statement is automatically revalidated when next referenced. You must then correct any errors that exist in that object before referencing it.

> **See Also:**
>
> *Oracle Database Concepts* for more information on dependencies

CHECKPOINT

Specify `CHECKPOINT` if you want Oracle Database to apply a checkpoint for the `DROP COLUMN` operation after processing *`integer`* rows; *`integer`* is optional and must be greater than zero. If *`integer`* is greater than the number of rows in the table, then the database applies a checkpoint after all the rows have been processed. If you do not specify *`integer`*, then the database sets the default of 512. Checkpointing cuts down the amount of undo logs accumulated during the `DROP COLUMN` operation to avoid running out of undo space. However, if this statement is interrupted after a checkpoint has been applied, then the table remains in an unusable state. While the table is unusable, the only operations allowed on it are `DROP TABLE`, `TRUNCATE TABLE`, and `ALTER TABLE DROP` ... `COLUMNS CONTINUE` (described in sections that follow).

You cannot use this clause with `SET UNUSED`, because that clause does not remove column data.

DROP COLUMNS CONTINUE Clause

Specify `DROP COLUMNS CONTINUE` to continue the drop column operation from the point at which it was interrupted. Submitting this statement while the table is in an invalid state results in an error.

Restrictions on Dropping Columns

Dropping columns is subject to the following restrictions:

- Each of the parts of this clause can be specified only once in the statement and cannot be mixed with any other `ALTER TABLE` clauses. For example, the following statements are not allowed:

  ```
  ALTER TABLE t1 DROP COLUMN f1 DROP (f2);
  ALTER TABLE t1 DROP COLUMN f1 SET UNUSED (f2);
  ALTER TABLE t1 DROP (f1) ADD (f2 NUMBER);
  ALTER TABLE t1 SET UNUSED (f3)
     ADD (CONSTRAINT ck1 CHECK (f2 > 0));
  ```

- You can drop an object type column only as an entity. To drop an attribute from an object type column, use the ALTER TYPE ... DROP ATTRIBUTE statement with the CASCADE INCLUDING TABLE DATA clause. Be aware that dropping an attribute affects all dependent objects. See *Oracle Database PL/SQL Language Reference* for more information.
- You can drop a column from an index-organized table only if it is not a primary key column. The primary key constraint of an index-organized table can never be dropped, so you cannot drop a primary key column even if you have specified CASCADE CONSTRAINTS.
- You can export tables with dropped or unused columns. However, you can import a table only if all the columns specified in the export files are present in the table (none of those columns has been dropped or marked unused). Otherwise, Oracle Database returns an error.
- You can set unused a column from a table that uses COMPRESS BASIC, but you cannot drop the column. However, all clauses of the *drop_column_clause* are valid for tables that use ROW STORE COMPRESS ADVANCED. See the semantics for *table_compression* for more information.
- You cannot drop a column on which a domain index has been built.
- You cannot drop a SCOPE table constraint or a WITH ROWID constraint on a REF column.
- You cannot use this clause to drop:
  - A pseudocolumn, cluster column, or partitioning column. You can drop nonpartitioning columns from a partitioned table if all the tablespaces where the partitions were created are online and in read/write mode.
  - A column from a nested table, an object table, a duplicated table, or a table owned by SYS.

> **See Also:**
>
> "Dropping a Column: Example"

add_period_clause

Use the *add_period_clause* to add a valid time dimension to *table*.

The *period_definition* clause of ALTER TABLE has the same semantics as in CREATE TABLE, with the following exceptions and additions:

- *valid_time_column* must not already exist in *table*.
- If you specify *start_time_column* and *end_time_column*, then these columns must already exist in *table* or you must specify the *add_column_clause* for each of these columns.
- If you specify *start_time_column* and *end_time_column* and these columns already exist in *table* and are populated with data, then for all rows where both columns have non-NULL values, the value of *start_time_column* must be earlier than the value of *end_time_column*.

> **See Also:**
>
> CREATE TABLE *period_definition* for the full semantics of this clause

drop_period_clause

Use the *drop_period_clause* to drop a valid time dimension from *table*.

For *valid_time_column*, specify the name of the valid time dimension you want to drop.

This clause has the following effects:

- The *valid_time_column* will be dropped from *table*.
- If the start time column and end time column were automatically created by Oracle Database when the valid time dimension was created, either with CREATE TABLE ... *period_definition* or ALTER TABLE ... *add_period_clause*, then they will be dropped. Otherwise, these columns will remain in *table* and revert to regular table columns.

> **See Also:**
>
> CREATE TABLE *period_definition* for more information on the *valid_time_column*, start time column, and end time column

rename_column_clause

Use the *rename_column_clause* to rename a column of *table*. The new column name must not be the same as any other column name in *table*.

When you rename a column, Oracle Database handles dependent objects as follows:

- Function-based indexes and check constraints that depend on the renamed column remain valid.
- Dependent views, triggers, functions, procedures, and packages are invalidated. Oracle Database attempts to revalidate them when they are next accessed, but you may need to alter these objects with the new column name if revalidation fails.
- If a domain index is defined on the column being renamed, then the database invokes the ODCIIndexAlter method with the RENAME option. This operation establishes correspondence between the indextype metadata and the base table

Restrictions on Renaming Columns

Renaming columns is subject to the following restrictions:

- You cannot combine this clause with any of the other *column_clauses* in the same statement.
- You cannot rename a column that is used to define a join index. Instead you must drop the index, rename the column, and re-create the index.
- You cannot rename a column in a duplicated table.

> **See Also:**
>
> "Renaming a Column: Example"

modify_collection_retrieval

Use the `modify_collection_retrieval` clause to change what Oracle Database returns when a collection item is retrieved from the database.

collection_item

Specify the name of a column-qualified attribute whose type is nested table or varray.

RETURN AS

Specify what Oracle Database should return as the result of a query:

- `LOCATOR` specifies that a unique locator for the nested table is returned.
- `VALUE` specifies that a copy of the nested table itself is returned.

> **See Also:**
>
> "Collection Retrieval: Example"

modify_LOB_storage_clause

The `modify_LOB_storage_clause` lets you change the physical attributes of `LOB_item`. You can specify only one `LOB_item` for each `modify_LOB_storage_clause`.

The sections that follow describe the semantics of parameters specific to modify_LOB_parameters. Unless otherwise documented in this section, the remaining LOB parameters have the same semantics when altering a table that they have when you are creating a table. Refer to the restrictions at the end of this section and to the `CREATE TABLE` clause *LOB_storage_parameters* for more information.

> **Note:**
>
> - You can modify LOB storage with an `ALTER TABLE` statement or with online redefinition by using the `DBMS_REDEFINITION` package. If you have not enabled LOB encryption, compression, or deduplication at create time, Oracle recommends that you use online redefinition to enable them after creation, as this process is more disk space efficient for changes to these three parameters. See *Oracle Database PL/SQL Packages and Types Reference* for more information on `DBMS_REDEFINITION`.
> - You cannot convert a LOB from one type of storage to the other. Instead you must migrate to SecureFiles or BasicFiles by using online redefinition or partition exchange.

PCTVERSION *integer*

Refer to the CREATE TABLE clause PCTVERSION integer for information on this clause.

LOB_retention_clause

If the database is in automatic undo mode, then you can specify RETENTION instead of PCTVERSION to instruct Oracle Database to retain old versions of this LOB. This clause overrides any prior setting of PCTVERSION.

FREEPOOLS *integer*

For BasicFiles LOBs, if the database is in automatic undo mode, then you can use this clause to specify the number of freelist groups for this LOB. This clause overrides any prior setting of FREELIST GROUPS. Refer to the CREATE TABLE clause FREEPOOLS integer for a full description of this parameter. The database ignores this parameter for SecureFiles LOBs.

REBUILD FREEPOOLS

This clause applies only to BasicFiles LOBs, not to SecureFiles LOBs. The REBUILD FREEPOOLS clause removes all the old versions of data from the LOB column. This clause is useful for removing all retained old version space in a LOB segment, freeing that space to be used immediately by new LOB data.

LOB_deduplicate_clause

This clause is valid only for SecureFiles LOBs. KEEP_DUPLICATES disables LOB deduplication. DEDUPLICATE enables LOB deduplication. All lobs in the segment are read, and any matching LOBs are deduplicated before returning.

LOB_compression_clause

This clause is valid only for SecureFiles LOBs. COMPRESS compresses all LOBs in the segment and then returns. NOCOMPRESS uncompresses all LOBs in the segment and then returns.

ENCRYPT | DECRYPT

LOB encryption has the same semantics as column encryption in general. See "ENCRYPT encryption_spec | DECRYPT" for more information.

CACHE, NOCACHE, CACHE READS

When you modify a LOB column from CACHE or NOCACHE to CACHE READS, or from CACHE READS to CACHE or NOCACHE, you can change the logging attribute. If you do not specify LOGGING or NOLOGGING, then this attribute defaults to the current logging attribute of the LOB column. If you do not specify CACHE, NOCACHE, or CACHE READS, then Oracle Database retains the existing values of the LOB attributes.

Restrictions on Modifying LOB Storage

Modifying LOB storage is subject to the following restrictions:

- You cannot modify the value of the INITIAL parameter in the *storage_clause* when modifying the LOB storage attributes.
- You cannot specify both the *allocate_extent_clause* and the *deallocate_unused_clause* in the same statement.
- You cannot specify both the PCTVERSION and RETENTION parameters.

- You cannot specify the *shrink_clause* for SecureFiles LOBs.

> **See Also:**
>
> *LOB_storage_clause* (in CREATE TABLE) for information on setting LOB parameters and "LOB Columns: Examples"

alter_varray_col_properties

The *alter_varray_col_properties* clause lets you change the storage characteristics of an existing LOB in which a varray is stored.

Restriction on Altering Varray Column Properties

You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the tablespace of the containing table.

REKEY *encryption_spec*

The REKEY clause causes the database to generate a new encryption key. All encrypted columns in the table are reencrypted using the new key and, if you specify the USING clause of the *encryption_spec*, a new encryption algorithm. You cannot combine this clause with any other clauses in this ALTER TABLE statement.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information on transparent column encryption

constraint_clauses

Use the *constraint_clauses* to add a new constraint using out-of-line declaration, modify the state of an existing constraint, or drop a constraint. Refer to *constraint* for a description of all the keywords and parameters of out-of-line constraints and *constraint_state*.

Adding a Constraint

The ADD clause lets you add a new out-of-line constraint or out-of-line REF constraint to the table.

Restrictions on Adding a Constraint

Adding constraints is subject to the following restrictions:

- You cannot add a constraint to a duplicated table.
- You cannot add a foreign key constraint to a sharded table.

> **See Also:**
>
> "Disabling a CHECK Constraint: Example", "Specifying Object Identifiers: Example", and "REF Columns: Examples"

Modifying a Constraint

The `MODIFY CONSTRAINT` clause lets you change the state of an existing constraint.

The `CASCADE` keyword is valid only when you are disabling a unique or primary key constraint on which a foreign key constraint is defined. In this case, you must specify `CASCADE` so that the unique or primary key constraint and all of its dependent foreign key constraints are disabled.

Restrictions on Modifying Constraints

Modifying constraints is subject to the following restrictions:

- You cannot change the state of a `NOT DEFERRABLE` constraint to `INITIALLY DEFERRED`.
- If you specify this clause for an index-organized table, then you cannot specify any other clauses in the same statement.
- You cannot change the `NOT NULL` constraint on a foreign key column of a reference-partitioned table, and you cannot change the state of a partitioning referential constraint of a reference-partitioned table.
- You cannot modify a constraint on a duplicated table.

> **See Also:**
>
> "Changing the State of a Constraint: Examples"

Renaming a Constraint

The `RENAME CONSTRAINT` clause lets you rename any existing constraint on *`table`*. The new constraint name cannot be the same as any existing constraint on any object in the same schema. All objects that are dependent on the constraint remain valid.

> **See Also:**
>
> "Renaming Constraints: Example"

drop_constraint_clause

The *`drop_constraint_clause`* lets you drop an integrity constraint from the database. Oracle Database stops enforcing the constraint and removes it from the data dictionary. You can specify only one constraint for each *`drop_constraint_clause`*, but you can specify multiple *`drop_constraint_clause`* in one statement.

PRIMARY KEY

Specify `PRIMARY KEY` to drop the primary key constraint of *`table`*.

UNIQUE

Specify `UNIQUE` to drop the unique constraint on the specified columns.

If you drop the primary key or unique constraint from a column on which a bitmap join index is defined, then Oracle Database invalidates the index. See CREATE INDEX for information on bitmap join indexes.

CONSTRAINT

Specify `CONSTRAINT` *`constraint_name`* to drop an integrity constraint other than a primary key or unique constraint.

CASCADE

Specify `CASCADE` if you want all other integrity constraints that depend on the dropped integrity constraint to be dropped as well.

KEEP INDEX | DROP INDEX

Specify `KEEP INDEX` or `DROP INDEX` to indicate whether Oracle Database should preserve or drop the index it has been using to enforce the `PRIMARY KEY` or `UNIQUE` constraint.

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed while dropping the constraint.

Restrictions on Dropping Constraints

Dropping constraints is subject to the following restrictions:

- You cannot drop a primary key or unique key constraint that is part of a referential integrity constraint without also dropping the foreign key. To drop the referenced key and the foreign key together, use the `CASCADE` clause. If you omit `CASCADE`, then Oracle Database does not drop the primary key or unique constraint if any foreign key references it.
- You cannot drop a primary key constraint (even with the `CASCADE` clause) on a table that uses the primary key as its object identifier (OID).
- If you drop a referential integrity constraint on a `REF` column, then the `REF` column remains scoped to the referenced table.
- You cannot drop the scope of a `REF` column.
- You cannot drop the `NOT NULL` constraint on a foreign key column of a reference-partitioned table, and you cannot drop a partitioning referential constraint of a reference-partitioned table.
- You cannot drop the `NOT NULL` constraint on a column that is defined with a default column value using the `ON NULL` clause.
- You cannot specify the `ONLINE` clause when dropping a `DEFERRABLE` constraint.

> **See Also:**
>
> "Dropping Constraints: Examples"

alter_external_table

Use the *alter_external_table* clauses to change the characteristics of an external table. This clause has no affect on the external data itself. The syntax and semantics of the *parallel_clause*, *enable_disable_clause*, *external_table_data_props*, and REJECT LIMIT clause are the same as described for CREATE TABLE. See the *external_table_clause* (in CREATE TABLE).

PROJECT COLUMN Clause

This clause lets you determine how the access driver validates the rows of an external table in subsequent queries. The default is PROJECT COLUMN ALL, which means that the access driver processes all column values, regardless of which columns are selected, and validates only those rows with fully valid column entries. If any column value would raise an error, such as a data type conversion error, then the row is rejected even if that column was not referenced in the select list. If you specify PROJECT COLUMN REFERENCED, then the access driver processes only those columns in the select list.

The ALL setting guarantees consistent result sets. The REFERENCED setting can result in different numbers of rows returned, depending on the columns referenced in subsequent queries, but is faster than the ALL setting. If a subsequent query selects all columns of the external table, then the settings behave identically.

Restrictions on Altering External Tables

Altering external tables is subject to the following restrictions:

- You cannot modify an external table using any clause outside of this clause.
- You cannot add a LONG, varray, or object type column to an external table, nor can you change the data type of an external table column to any of these data types.
- You cannot modify the storage parameters of an external table.

alter_table_partitioning

The clauses in this section apply only to partitioned tables. You cannot combine partition operations with other partition operations or with operations on the base table in the same ALTER TABLE statement.

Notes on Changing Table Partitioning

The following notes apply when changing table partitioning:

- If you drop, exchange, truncate, move, modify, or split a partition on a table that is a master table for one or more materialized views, then existing bulk load information about the table will be deleted. Therefore, be sure to refresh all dependent materialized views before performing any of these operations.
- If a bitmap join index is defined on *table*, then any operation that alters a partition of *table* causes Oracle Database to mark the index UNUSABLE.

- The only *alter_table_partitioning* clauses you can specify for a reference-partitioned table are *modify_table_default_attrs*, *move_table_[sub]partition*, *truncate_partition_subpart*, and *exchange_partition_subpart*. None of these operations cascade to any child table of the reference-partitioned table. No other partition maintenance operations are valid on a reference-partitioned table, but you can specify the other partition maintenance operations on the parent table of a reference-partitioned table, and the operation will cascade to the child reference-partitioned table.
- When adding partitions and subpartitions, bear in mind that you can specify up to a total of 1024K-1 partitions and subpartitions for each table.
- When you add a table partition or subpartition and you omit the partition name, the database generates a name using the rules described in "Notes on Partitioning in General".
- When you move, add (hash only), coalesce, drop, split, merge, rename, or truncate a table partition or subpartition, the procedures, functions, packages, package bodies, views, type bodies, and triggers that reference the table remain valid. All other dependent objects are invalidated.
- Deferred segment creation is not supported for partition maintenance operations that create new segments on tables with LOB columns; segments will always be created for the involved (sub)partitions.
- For sharded tables, the only clauses you can specify for modifying table partitions and subpartitions are `UNUSABLE LOCAL INDEXES` and `REBUILD UNUSABLE LOCAL INDEXES`. You cannot perform any other modifications for individual partitions and subpartitions on a sharded table.
- For sharded tables, the only supported partition maintenance operations are truncating partitions and subpartitions. You cannot perform any other partition maintenance operations on a sharded table.

For additional information on partition operations on tables with an associated `CONTEXT` domain index, refer to *Oracle Text Reference*.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

modify_table_default_attrs

The *modify_table_default_attrs* clause lets you specify new default values for the attributes of *table*. Only attributes named in the statement are affected. Partitions and LOB partitions you create subsequently will inherit these values unless you override them explicitly when creating the partition or LOB partition. Existing partitions and LOB partitions are not affected by this clause.

Only attributes named in the statement are affected, and the default values specified are overridden by any attributes specified at the individual partition or LOB partition level.

- `FOR` *partition_extended_name* applies only to composite-partitioned tables. This clause specifies new default values for the attributes of the partition identified in *partition_extended_name*. Subpartitions and LOB subpartitions of that partition that you create subsequently will inherit these values unless you override them explicitly when creating the subpartition or LOB subpartition. Existing subpartitions are not affected by this clause.
- `PCTTHRESHOLD`, *prefix_compression*, and the *alter_overflow_clause* are valid only for partitioned index-organized tables.

- You can specify the *prefix_compression* clause only if prefix compression is already specified at the table level. Further, you cannot specify an integer after the COMPRESS keyword. Prefix length can be specified only when you create the table.
- You cannot specify the PCTUSED parameter in *segment_attributes* for the index segment of an index-organized table.
- The *read_only_clause* lets you modify the default read-only or read/write mode for the table. The new default mode will be assigned to partitions or subpartitions that are subsequently added to the table, unless you override this behavior by specifying the mode for the new partition or subpartition. When you modify the default read-only or read/write mode of a table, you do not change the mode of the existing partitions and subpartitions in the table. Refer to the *read_only_clause* of CREATE TABLE for the full semantics of this clause.
- The *indexing_clause* lets you modify the default indexing property for the table. The new default indexing property will be assigned to partitions or subpartitions that are subsequently added to the table, unless you override this behavior by specifying the indexing property for the new partition or subpartition. When you modify the default indexing property of a table, you do not change the indexing property of the existing partitions and subpartitions in the table. Refer to the *indexing_clause* of CREATE TABLE for the full semantics of this clause.

alter_automatic_partitioning

This clause allows you to manage automatic list-partitioned tables, as follows:

- Use the SET PARTITIONING AUTOMATIC clause to convert a regular list-partitioned table to an automatic list-partitioned table.
- Use the SET PARTITIONING MANUAL clause to convert an automatic list-partitioned table to a regular list-partitioned table.
- You can specify the SET STORE IN clause only for automatic list-partitioned tables. It lets you specify one or more tablespaces into which the database will store data for any subsequent automatically created list partitions. This clause overrides any tablespaces that might have been set for the table by a previously issued SET STORE IN clause.

To determine whether an existing table is an automatic list-partitioned table, you can query the AUTOLIST column of the USER_, DBA_, ALL_PART_TABLES data dictionary views.

Restriction on *alter_automatic_partitioning*

You cannot convert a regular list-partitioned table that contains a DEFAULT partition to an automatic list-partitioned table.

> **See Also:**
>
> The AUTOMATIC clause in the documentation on CREATE TABLE for more information on automatic list-partitioned tables

alter_interval_partitioning

Use this clause:

- To convert an existing range-partitioned table to interval partitioning. The database automatically creates partitions of the specified numeric range or datetime interval as needed for data beyond the highest value allowed for the last range partition. If the table has reference-partitioned child tables, then the child tables are converted to interval reference-partitioned child tables.
- To change the interval of an existing interval-partitioned table. The database first converts existing interval partitions to range partitions and determines the high value of the defined range partitions. The database then automatically creates partitions of the specified numeric range or datetime interval as needed for data that is beyond that high value.
- To change the tablespace storage for an existing interval-partitioned table. If the table has interval reference-partitioned child tables, then the new tablespace storage is inherited by any child table that does not have its own table-level default tablespace.
- To change an interval-partitioned table back to a range-partitioned table. Use `SET INTERVAL ()` to disable interval partitioning. The database converts existing interval partitions to range partitions, using the higher boundaries of created interval partitions as upper boundaries for the range partitions to be created. If the table has interval reference-partitioned child tables, then the child tables are converted to ordinary reference-partitioned child tables.

For *`expr`*, specify a valid number or interval expression.

> **See Also:**
>
> The `CREATE TABLE` "INTERVAL Clause" and *Oracle Database VLDB and Partitioning Guide* for more information on interval partitioning

_set_subpartition_template_

Use the *`set_subpartition_template`* clause to create or replace existing default range, list, or hash subpartition definitions for each table partition. This clause is valid only for composite-partitioned tables. It replaces the existing subpartition template or creates a new template if you have not previously created one. Existing subpartitions are not affected, nor are existing local and global indexes. However, subsequent partitioning operations (such as add and merge operations) will use the new template.

You can drop an existing subpartition template by specifying `ALTER TABLE` *`table`* `SET SUBPARTITION TEMPLATE ()`.

The *`set_subpartition_template`* clause has the same semantics as the *`subpartition_template`* clause of `CREATE TABLE`. Refer to the *subpartition_template* clause of `CREATE TABLE` for more information.

_modify_table_partition_

The *`modify_table_partition`* clause lets you change the real physical attributes of a range, hash, list partition, or system partition. This clause optionally modifies the storage attributes of one or more LOB items for the partition. You can specify new values for physical attributes (with some restrictions, as noted in the sections that follow), logging, and storage parameters.

For all types of partitions, you can also specify how Oracle Database should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses".

For partitioned index-organized tables, you can also update the mapping table in conjunction with partition changes. See the *alter_mapping_table_clauses* .

read_only_clause

Use the `read_only_clause` to put a table partition in read-only or read/write mode. Refer to the *read_only_clause* of `CREATE TABLE` for the full semantics of this clause.

indexing_clause

Use the `indexing_clause` to modify the indexing property of a table partition. The indexing property determines whether the partition is included in partial indexes on the table. You can specify the `indexing_clause` in the `modify_range_partition`, `modify_hash_partition`, and `modify_list_partition` clauses.

Specify `INDEXING ON` to change the indexing property for a table partition to `ON`. This operation has no effect on full indexes on the table. It has the following effects on partial indexes on the table:

- Local partial indexes: The table partition is included in the index. The corresponding index partition is rebuilt and marked `USABLE`.
- Global partial indexes: The table partition is included in the index. Index entries for the table partition are added to the index as part of routine index maintenance.

Specify `INDEXING OFF` to change the indexing property for a table partition to `OFF`. This operation has no effect on full indexes on the table. It has the following effects on partial indexes on the table:

- Local partial indexes: The table partition is excluded from the index. The corresponding index partition is marked `UNUSABLE`.
- Global partial indexes: The table partition is excluded from the index. Index entries for the table partition are removed from the index. This is a metadata-only operation and the index entries will continue to be physically stored in the index. You can remove these orphaned index entries by specifying `COALESCE CLEANUP` in the ALTER INDEX statement or in the *modify_index_partition* clause.

Restriction on column of type object

You cannot partition a table that has an object type. The alter table modification to a partitioned state is only supported for non-partitioned heap tables with zero columns of type object.

Restriction on the *indexing_clause*

You can specify this clause only for partitions of a simple partitioned table. For composite-partitioned tables, you can specify the `indexing_clause` at the table subpartition level. Refer to *modify_table_subpartition* for more information.

Notes on Modifying Table Partitions

The following notes apply to operations on range, list, and hash table partitions:

- For all types of table partition, in the `partition_attributes` clause, the `shrink_clause` lets you compact an individual partition segment. Refer to *shrink_clause* for additional information on this clause.
- The syntax and semantics for modifying a system partition are the same as those for modifying a hash partition. Refer to *modify_hash_partition*.

- If *table* is composite partitioned, then:
  - If you specify the *allocate_extent_clause*, then Oracle Database allocates an extent for each subpartition of *partition*.
  - If you specify the *deallocate_unused_clause*, then Oracle Database deallocates unused storage from each subpartition of *partition*.
  - Any other attributes changed in this clause will be changed in subpartitions of *partition* as well, overriding existing values. To avoid changing the attributes of existing subpartitions, use the FOR PARTITION clause of *modify_table_default_attrs*.
- When you modify the *partition_attributes* of a table partition with equipartitioned nested tables, the changes do not apply to the nested table partitions corresponding to the table partition being modified. However, you can modify the storage table of the nested table partition directly with an ALTER TABLE statement.
- Unless otherwise documented, the remaining clauses of *partition_attributes* have the same behavior they have when you are creating a partitioned table. Refer to the CREATE TABLE *table_partitioning_clauses* for more information.

> **See Also:**
>
> "Modifying Table Partitions: Examples"

modify_range_partition

Use this clause to modify the characteristics of a range partition.

add_range_subpartition

This clause is valid only for range-range composite partitions. It lets you add one or more range subpartitions to *partition*.

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to add a subpartition to composite-partitioned external table. In this case, you can specify the optional *external_part_subpart_data_props* clause of the *range_subpartition_desc* clause. Refer to external_part_subpart_data_props for the full semantics of this clause.

Restriction on Adding Range Subpartitions

If *table* is an index-organized table, then you can add only one range subpartition at a time.

add_hash_subpartition

This clause is valid only for range-hash composite partitions. The *add_hash_subpartition* clause lets you add a hash subpartition to *partition*. Oracle Database populates the new subpartition with rows rehashed from the other subpartition(s) of *partition* as determined by the hash function. For optimal load balancing, the total number of subpartitions should be a power of 2.

In the *partitioning_storage_clause*, the only clause you can specify for subpartitions is the TABLESPACE clause. If you do not specify TABLESPACE, then the new subpartition will reside in the default tablespace of *partition*.

Oracle Database adds local index partitions corresponding to the selected partition.

Oracle Database marks UNUSABLE the local index partitions corresponding to the added partitions. The database invalidates any indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

add_list_subpartition

This clause is valid only for range-list and list-list composite partitions. It lets you add one or more list subpartitions to *partition*, and only if you have not already created a DEFAULT subpartition.

- The *list_values_clause* is required in this operation, and the values you specify in the *list_values_clause* cannot exist in any other subpartition of *partition*. However, these values can duplicate values found in subpartitions of other partitions.
- In the *partitioning_storage_clause*, the only clauses you can specify for subpartitions are the TABLESPACE clause and table compression.
- Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to add a subpartition to composite-partitioned external table. In this case, you can specify the optional *external_part_subpart_data_props* clause of the *list_subpartition_desc* clause. Refer to external_part_subpart_data_props for the full semantics of this clause.

For each added subpartition, Oracle Database also adds a subpartition with the same value list to all local index partitions of the table. The status of existing local and global index partitions of *table* are not affected.

Restrictions on Adding List Subpartitions

The following restrictions apply to adding list subpartitions:

- You cannot specify this clause if you have already created a DEFAULT subpartition for this partition. Instead you must split the DEFAULT partition using the *split_list_subpartition* clause.
- If *table* is an index-organized table, then you can add only one list subpartition at a time.

coalesce_table_subpartition

COALESCE SUBPARTITION applies only to hash subpartitions. Use the COALESCE SUBPARTITION clause if you want Oracle Database to select the last hash subpartition, distribute its contents into one or more remaining subpartitions (determined by the hash function), and then drop the last subpartition.

- Oracle Database drops local index partitions corresponding to the selected partition.
- Oracle Database marks UNUSABLE the local index partitions corresponding to one or more absorbing partitions. The database invalidates any global indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

modify_hash_partition

When modifying a hash partition, in the *partition_attributes* clause, you can specify only the *allocate_extent_clause* and *deallocate_unused_clause*. All other attributes of the partition are inherited from the table-level defaults except TABLESPACE, which stays the same as it was at create time.

modify_list_partition

Clauses available to you when modifying a list partition have the same semantics as when you are modifying a range partition. When modifying a list partition, the following additional clauses are available:

ADD | DROP VALUES Clauses

These clauses are valid only when you are modifying composite partitions. Local and global indexes on the table are not affected by either of these clauses.

- Use the ADD VALUES clause to extend the *partition_key_value* list of *partition* to include additional values. The added partition values must comply with all rules and restrictions listed in the CREATE TABLE clause *list_partitions* .
- Use the DROP VALUES clause to reduce the *partition_key_value* list of *partition* by eliminating one or more *partition_key_value*. When you specify this clause, Oracle Database checks to ensure that no rows with this value exist. If such rows do exist, then Oracle Database returns an error.

> **Note:**
>
> ADD VALUES and DROP VALUES operations on a table with a DEFAULT list partition are enhanced if you have defined a local prefixed index on the table.

Restrictions on Adding and Dropping List Values

Adding and dropping list values are subject to the following restrictions:

- You cannot add values to or drop values from a DEFAULT list partition.
- If *table* contains a DEFAULT partition and you attempt to add values to a nondefault partition, then Oracle Database will check that the values being added do not already exist in the DEFAULT partition. If the values do exist in the DEFAULT partition, then Oracle Database returns an error.

modify_table_subpartition

This clause applies only to composite-partitioned tables. Its subclauses let you modify the characteristics of an individual range, list, or hash subpartition.

The *shrink_clause* lets you compact an individual subpartition segment. Refer to *shrink_clause* for additional information on this clause.

You can also specify how Oracle Database should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses".

Use the *read_only_clause* to put a table subpartition in read-only or read/write mode. Refer to the *read_only_clause* of CREATE TABLE for the full semantics of this clause.

Use the *indexing_clause* to modify the indexing property of a table subpartition. The indexing property determines whether the subpartition is included in partial indexes on the table. Modifying the indexing property of table subpartitions has the same effect on index subpartitions as modifying the indexing property of table partitions has on index partitions. Refer to the indexing_clause of *modify_table_partition* for details.

Restriction on Modifying Hash Subpartitions

The only *modify_LOB_parameters* you can specify for *subpartition* are the *allocate_extent_clause* and *deallocate_unused_clause*.

ADD | DROP VALUES Clauses

These clauses are valid only when you are modifying list subpartitions. Local and global indexes on the table are not affected by either of these clauses.

- Use the ADD VALUES clause to extend the *subpartition_key_value* list of *subpartition* to include additional values. The added partition values must comply with all rules and restrictions listed in the CREATE TABLE clause *list_partitions* .
- Use the DROP VALUES clause to reduce the *subpartition_key_value* list of *subpartition* by eliminating one or more *subpartition_key_value*. When you specify this clause, Oracle Database checks to ensure that no rows with this value exist. If such rows do exist, then Oracle Database returns an error.

You can also specify how Oracle Database should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses".

Restriction on Modifying List Subpartitions

The only *modify_LOB_parameters* you can specify for *subpartition* are the *allocate_extent_clause* and *deallocate_unused_clause*.

move_table_partition

Use the *move_table_partition* clause to move *partition* to another segment. You can move partition data to another tablespace, recluster data to reduce fragmentation, or change create-time physical attributes.

If the table contains LOB columns, then you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this partition. Only the LOBs named are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, then its LOB data and LOB index segments are not moved.

If the table contains nested table columns, then you can use the *nested_table_col_properties clause* of the *table_partition_description* to move the nested table segments associated with this partition. Only the nested table items named are affected. If you do not specify the *nested_table_col_properties clause* of the *table_partition_description* for a particular nested table column, then its segments are not moved.

Oracle Database moves local index partitions corresponding to the specified partition. If the moved partitions are not empty, then the database marks them UNUSABLE. The database invalidates global indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

When you move a LOB data segment, Oracle Database drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

The move operation obtains its parallel attribute from the *parallel_clause*, if specified. When it is not specified, the default parallel attributes of the table, if any, are used. If neither is specified, then Oracle Database performs the move serially.

Specifying the *parallel_clause* in MOVE PARTITION does not change the default parallel attributes of *table*.

> **Note:**
>
> For index-organized tables, Oracle Database uses the address of the primary key, as well as its value, to construct logical rowids. The logical rowids are stored in the secondary index of the table. If you move a partition of an index-organized table, then the address portion of the rowids will change, which can hamper performance. To ensure optimal performance, rebuild the secondary index(es) on the moved partition to update the rowids.

> **See Also:**
>
> "Moving Table Partitions: Example"

MAPPING TABLE

The MAPPING TABLE clause is relevant only for an index-organized table that already has a mapping table defined for it. Oracle Database moves the mapping table along with the moved index-organized table partition. The mapping table partition inherits the physical attributes of the moved index-organized table partition. This is the only way you can change the attributes of the mapping table partition. If you omit this clause, then the mapping table partition retains its original attributes.

Oracle Database marks UNUSABLE all corresponding bitmap index partitions.

Refer to the *mapping_table_clauses* (in CREATE TABLE) for more information on this clause.

ONLINE

Specify ONLINE to indicate that DML operations on the table partition will be allowed while moving the table partition.

Restrictions on the ONLINE Clause

The ONLINE clause is subject to the following restrictions when moving table partitions:

- You cannot specify the ONLINE clause for tables owned by SYS.
- You cannot specify the ONLINE clause for index-organized tables.

- You cannot specify the ONLINE clause for heap-organized tables that contain object types or on which bitmap join indexes or domain indexes are defined.
- Parallel DML and direct path INSERT operations require an exclusive lock on the table. Therefore, these operations are not supported concurrently with an ongoing online partition MOVE, due to conflicting locks.

Restrictions on Moving Table Partitions

Moving table partitions is subject to the following restrictions:

- If *partition* is a hash partition, then the only attribute you can specify in this clause is TABLESPACE.
- You cannot specify this clause for a partition containing subpartitions. However, you can move subpartitions using the *move_table_subpartition* clause.

move_table_subpartition

Use the *move_table_subpartition* clause to move the subpartition identified by *subpartition_extended_name* to another segment. If you do not specify TABLESPACE, then the subpartition remains in the same tablespace.

If the subpartition is not empty, then Oracle Database marks UNUSABLE all local index subpartitions corresponding to the subpartition being moved. You can update all indexes on heap-organized tables during this operation using the *update_index_clauses*.

If the table contains LOB columns, then you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this subpartition. Only the LOBs specified are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, then its LOB data and LOB index segments are not moved.

When you move a LOB data segment, Oracle Database drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

ONLINE

Specify ONLINE to indicate that DML operations on the table subpartition will be allowed while moving the table subpartition.

Restrictions on the ONLINE Clause

The ONLINE clause for moving table subpartitions is subject to the same restrictions as the ONLINE clause for moving table partitions. Refer to "Restrictions on the ONLINE Clause."

Restriction on Moving Table Subpartitions

The only clauses of the *partitioning_storage_clause* you can specify are the TABLESPACE clause and *table_compression*.

add_table_partition

Use the *add_table_partition* clause to add one or more range, list, or system partitions to *table*, or to add one hash partition to *table*.

For each partition added, Oracle Database adds to any local index defined on *table* a new partition with the same name as that of the base table partition. If the index already has a partition with such a name, then Oracle Database generates a partition name of the form SYS_P*n*.

If *table* is index organized, then for each partition added Oracle Database adds a partition to any mapping table and overflow area defined on the table as well.

If *table* is the parent table of a reference-partitioned table, then you can use the *dependent_tables_clause* to propagate the partition maintenance operation you are specifying in this statement to all the reference-partitioned child tables.

The default indexing property of *table* is inherited by the new table partition(s). You can override this by setting the indexing property of a list, range, or system partition using the *indexing_clause* in the *table_partition_description* clause, or a hash partition using the *indexing_clause* in the *add_hash_partition_clause*.

For each partition added to a composite-partitioned table, Oracle Database adds a new index partition with the same subpartition descriptions to all local indexes defined on *table*. Global indexes defined on *table* are not affected. If you specify the indexing property for the new table partition, then the new subpartitions inherit the indexing property for the partition. Otherwise, the new subpartitions inherit the default indexing property for the table. You can override this by setting the indexing property of a subpartition using the *indexing_clause* in the *range_subpartition_desc*, *individual_hash_subparts*, and *list_subpartition_desc* clauses.

BEFORE Clause

You can specify the optional BEFORE clause only when adding system partitions to *table*. This clause lets you specify where the new partition(s) should be added in relation to existing partitions. You cannot split a system partition. Therefore, this clause is useful if you want to divide the contents of one existing partition among multiple new partitions. If you omit this clause, then the database adds the new partition(s) after the existing partitions.

Restriction on Adding Table Partitions

If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can add only one partition at a time.

> **See Also:**
>
> "Adding a Table Partition with a LOB and Nested Table Storage: Examples" and "Adding Multiple Partitions to a Table: Example"

add_range_partition_clause

The *add_range_partition_clause* lets you add a new range partition to the high end of a range-partitioned or composite range-partitioned table (after the last existing partition).

If a domain index is defined on *table*, then the index must not be marked IN_PROGRESS or FAILED.

Restrictions on Adding Range Partitions

Adding range partitions is subject to the following restrictions:

- If the upper partition bound of each partitioning key in the existing high partition is MAXVALUE, then you cannot add a partition to the table. Instead, use the *split_table_partition* clause to add a partition at the beginning or the middle of the table.

- The *prefix_compression* and OVERFLOW clauses, are valid only for a partitioned index-organized table. You can specify *prefix_compression* only if prefix compression is enabled at the table level. You can specify OVERFLOW only if the partitioned table already has an overflow segment.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.

range_values_clause

Specify the upper bound for the new partition. The *value_list* is a comma-delimited, ordered list of literal values corresponding to the partitioning key columns. The *value_list* must collate greater than the partition bound for the highest existing partition in the table.

table_partition_description

Use this clause to specify any create-time physical attributes for the new partition. If the table contains LOB columns, then you can also specify partition-level attributes for one or more LOB items.

external_part_subpart_data_props

Starting with Oracle Database 12*c* Release 2 (12.2), Oracle supports partitioned and composite-partitioned external tables. When adding a partition to such a table, you can optionally use this clause to specify the DEFAULT DIRECTORY and LOCATION for the partition. Refer to DEFAULT DIRECTORY and LOCATION in the documentation on CREATE TABLE for the full semantics of these clauses.

Subpartition Descriptions

These clauses are valid only for composite-partitioned tables. Use the *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* clause as appropriate, if you want to specify subpartitions for the new partition. This clause overrides any subpartition descriptions defined in *subpartition_template* at the table level.

add_hash_partition_clause

The *add_hash_partition_clause* lets you add a new hash partition to the high end of a hash-partitioned table. Oracle Database populates the new partition with rows rehashed from other partitions of *table* as determined by the hash function. For optimal load balancing, the total number of partitions should be a power of 2.

You can specify a name for the partition, and optionally a tablespace where it should be stored. If you do not specify a name, then the database assigns a partition name of the form SYS_P*n*. If you do not specify TABLESPACE, then the new partition is stored in the default tablespace of the table. Other attributes are always inherited from table-level defaults.

If this operation causes data to be rehashed among partitions, then the database marks UNUSABLE any corresponding local index partitions. You can update all indexes on heap-organized tables during this operation using the *update_index_clauses*.

Use the *parallel_clause* to specify whether to parallelize the creation of the new partition.

Use the *read_only_clause* to put a table partition in read-only or read/write mode. Refer to the *read_only_clause* of CREATE TABLE for the full semantics of this clause.

Use the *indexing_clause* to specify the indexing property for the partition. If you do not specify this clause, then the partition inherits the default indexing property of *table*.

> **See Also:**
>
> CREATE TABLE and *Oracle Database VLDB and Partitioning Guide* for more information on hash partitioning

add_list_partition_clause

The *add_list_partition_clause* lets you add a new partition to *table* using a new set of partition values. You can specify any create-time physical attributes for the new partition. If the table contains LOB columns, then you can also specify partition-level attributes for one or more LOB items.

Restrictions on Adding List Partitions

You cannot add a list partition if you have already defined a DEFAULT partition for the table. Instead, you must use the *split_table_partition* clause to split the DEFAULT partition.

> **See Also:**
>
> - *list_partitions* of CREATE TABLE for more information and restrictions on list partitions
> - "Working with Default List Partitions: Example"

add_system_partition_clause

Use this clause to add a partition to a system-partitioned table. Oracle Database adds a corresponding index partition to all local indexes defined on the table.

The *table_partition_description* lets you specify partition-level attributes of the new partition. The values of any unspecified attributes are inherited from the table-level values.

Restriction on Adding System Partitions

You cannot specify the OVERFLOW clause when adding a system partition.

> **See Also:**
>
> The CREATE TABLE clause *system_partitioning* for more information on system partitions

coalesce_table_partition

COALESCE applies only to hash partitions. Use the *coalesce_table_partition* clause to indicate that Oracle Database should select the last hash partition, distribute its contents into one or more remaining partitions as determined by the hash function, and then drop the last partition.

Oracle Database drops local index partitions corresponding to the selected partition. The database marks UNUSABLE the local index partitions corresponding to one or more absorbing partitions. The database invalidates any indexes on heap-organized tables. You can update all indexes during this operation using the *update_index_clauses*.

Restriction on Coalescing Table Partitions

If you update global indexes using the *update_all_indexes_clause*, then you can specify only the keywords UPDATE INDEXES, not the subclause.

drop_table_partition

The *drop_table_partition* clause removes partitions, and the data in those partitions, from a partitioned table. If you want to drop a partition but keep its data in the table, then you must merge the partition into one of the adjacent partitions.

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to drop a partition from a partitioned table or composite-partitioned external table.

> **See Also:**
>
> *merge_table_partitions*

Use the *partition_extended_names* clause to specify one or more partitions to be dropped. When specifying multiple partitions, you must specify all partitions by name, as shown in the upper branch of the syntax diagram, or all partitions using the FOR clause, as shown in the lower branch of the syntax diagram. You cannot use both types of syntax in one drop operation.

- If *table* has LOB columns, then Oracle Database also drops the LOB data and LOB index partitions and any subpartitions corresponding to the table partition(s) being dropped.
- If *table* has equipartitioned nested table columns, then Oracle Database also drops the nested table partitions corresponding to the table partition(s) being dropped.
- If *table* is index organized and has a mapping table defined on it, then the database drops the corresponding mapping table partition(s) as well.
- Oracle Database drops local index partitions and subpartitions corresponding to the dropped partition(s), even if they are marked UNUSABLE.

You can update indexes on *table* during this operation using the *update_index_clauses*. Updates to global indexes are metadata-only and the index entries for records that are dropped by the drop operation will continue to be physically stored in the index. You can remove these orphaned index entries by specifying

COALESCE CLEANUP in the ALTER INDEX statement or in the *modify_index_partition* clause.

If you specify the *parallel_clause* with the *update_index_clauses*, then the database parallelizes the index update, not the drop operation.

If you drop a range partition and later insert a row that would have belonged to the dropped partition, then the database stores the row in the next higher partition. However, if that partition is the highest partition, then the insert will fail, because the range of values represented by the dropped partition is no longer valid for the table.

Restrictions on Dropping Table Partitions

Dropping table partitions is subject to the following restrictions:

- You cannot drop a partition of a hash-partitioned table. Instead, use the *coalesce_table_partition* clause.
- You cannot drop all of the partitions in a table. Instead, drop the table.
- If you update global indexes using the update_all_indexes_clause, then you can specify only the UPDATE INDEXES keywords but not the subclause.
- If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can drop only one partition at a time.
- You cannot drop a partition of a duplicated table.

  Dropping a partition does not place the partition in the Oracle Database recycle bin, regardless of the setting of the recycle bin. Dropped partitions are immediately removed from the system.

> **See Also:**
>
> "Dropping a Table Partition: Example"

drop_table_subpartition

Use this clause to drop range or list subpartitions from a range, list, or hash composite-partitioned table. Oracle Database deletes any rows in the dropped subpartition(s).

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to drop a subpartition from a composite-partitioned external table.

Use the *subpartition_extended_names* clause to specify one or more subpartitions to be dropped. When specifying multiple subpartitions, you must specify all subpartitions by name, as shown in the upper branch of the syntax diagram, or all subpartitions using the FOR clause, as shown in the lower branch of the syntax diagram. You cannot use both types of syntax in one drop operation.

Oracle Database drops the corresponding subpartition(s) of any local index. Other index subpartitions are not affected. Any global indexes are marked UNUSABLE unless you specify the *update_global_index_clause* or *update_all_indexes_clause*. Updates to global indexes are metadata-only and the index entries for records that are dropped by the drop operation will continue to be physically stored in the index. You can remove these orphaned index entries by specifying COALESCE CLEANUP in the ALTER INDEX statement or in the *modify_index_partition* clause.

Restrictions on Dropping Table Subpartitions

Dropping table subpartitions is subject to the following restrictions:

- You cannot drop a hash subpartition. Instead use the `MODIFY PARTITION` ... `COALESCE SUBPARTITION` syntax.
- You cannot drop all of the subpartitions in a partition. Instead, use the *`drop_table_partition`* clause.
- If you update the global indexes, then you cannot specify the optional subclause of the *`update_all_indexes_clause`*.
- If *`table`* is an index-organized table, then you can drop only one subpartition at a time.
- When dropping multiple subpartitions, all of the subpartitions must be in the same partition.
- You cannot drop a subpartition of a duplicated table.

rename_partition_subpart

Use the *`rename_partition_subpart`* clause to rename a table partition or subpartition to *`new_name`*. For both partitions and subpartitions, *`new_name`* must be different from all existing partitions and subpartitions of the same table.

If *`table`* is index organized, then Oracle Database assigns the same name to the corresponding primary key index partition as well as to any existing overflow partitions and mapping table partitions.

Starting with Oracle Database 12*c* Release 2 (12.2), you can use this clause to rename a partition or subpartition in a partitioned or composite-partitioned external table.

> **See Also:**
>
> "Renaming Table Partitions: Examples"

truncate_partition_subpart

Specify `TRUNCATE` *`partition_extended_names`* to remove all rows from the partition(s) identified by *`partition_extended_names`* or, if the table is composite partitioned, all rows from the subpartitions of those partitions. Specify `TRUNCATE` *`subpartition_extended_names`* to remove all rows from individual subpartitions. If *`table`* is index organized, then Oracle Database also truncates any corresponding mapping table partitions and overflow area partitions.

When specifying multiple partitions, you must specify all partitions by name, as shown in the upper branch of the *`partition_extended_names`* syntax diagram, or all partitions using the `FOR` clause, as shown in the lower branch of the syntax diagram. You cannot use both types of syntax in one truncate operation. The same rule applies when specifying multiple subpartitions with the subpartition_extended_names clause.

For each specified partition or subpartition:

- If the partition or subpartition to be truncated contain data, then you must first disable any referential integrity constraints on the table. Alternatively, you can delete the rows and then truncate the partition.

- If *table* contains any LOB columns, then the LOB data and LOB index segments for the partition are also truncated. If *table* is composite partitioned, then the LOB data and LOB index segments for the subpartitions of the partition are truncated.
- If *table* contains any equipartitioned nested tables, then you cannot truncate the parent partition unless its corresponding nested table partition is empty.
- If a domain index is defined on *table*, then the index must not be marked IN_PROGRESS or FAILED, and the index partition corresponding to the table partition being truncated must not be marked IN_PROGRESS.

For each partition or subpartition truncated, Oracle Database also truncates corresponding local index partitions and subpartitions. If those index partitions or subpartitions are marked UNUSABLE, then the database truncates them and resets the UNUSABLE marker to VALID.

You can update indexes on *table* during this operation using the *update_index_clauses*. Updates to global indexes are metadata-only and the index entries for records that are dropped by the truncate operation will continue to be physically stored in the index. You can remove these orphaned index entries by specifying COALESCE CLEANUP in the ALTER INDEX statement or in the *modify_index_partition* clause.

If you specify the *parallel_clause* with the *update_index_clauses*, then the database parallelizes the index update, not the truncate operation.

DROP STORAGE

Specify DROP STORAGE to deallocate all space from the deleted rows, except the space allocated by the MINEXTENTS parameter. This space can subsequently be used by other objects in the tablespace.

DROP ALL STORAGE

Specify DROP ALL STORAGE to deallocate all space from the deleted rows, including the space allocated by the MINEXTENTS parameter. All segments for the partition(s) or subpartition(s), as well as all segments for their dependent objects, will be deallocated.

Restrictions on DROP ALL STORAGE

This clause is subject to the same restrictions as described in "Restrictions on Deferred Segment Creation".

REUSE STORAGE

Specify REUSE STORAGE to keep space from the deleted rows allocated to the partition(s) or subpartition(s). The space is subsequently available only for inserts and updates to the same partition(s) or subpartition(s).

CASCADE

Specify CASCADE to truncate the corresponding partition(s) or subpartition(s) in all reference-partitioned child tables of *table*.

Restrictions on Truncating Table Partitions and Subpartitions

Truncating table partitions and subpartitions is subject to the following restrictions:

- If you update global indexes using the *update_all_indexes_clause*, then you can specify only the UPDATE INDEXES keywords, not the subclause.
- If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can truncate only one table partition or one table subpartition at a time.

- You cannot truncate partitions or subpartitions in a duplicated table.

> **See Also:**
>
> "Truncating Table Partitions: Example"

split_table_partition

The *split_table_partition* clause lets you create, from the partition identified by *partition_extended_name*, multiple new partitions, each with a new segment, new physical attributes, and new initial extents. The segment associated with the current partition is discarded.

The new partitions inherit all unspecified physical attributes from the current partition.

> **Note:**
>
> Oracle Database can optimize and speed up SPLIT PARTITION and SPLIT SUBPARTITION operations if specific conditions are met. Refer to *Oracle Database VLDB and Partitioning Guide* for information on optimizing these operations.

- If you split a DEFAULT list partition, then the last resulting partition will have the DEFAULT value. All other resulting partitions will have the specified split values.
- If *table* is index organized, then Oracle Database splits any corresponding mapping table partition and places it in the same tablespace as the parent index-organized table partition. The database also splits any corresponding overflow area, and you can use the OVERFLOW clause to specify segment attributes for the new overflow areas.
- If *table* contains LOB columns, then you can use the *LOB_storage_clause* to specify separate LOB storage attributes for the LOB data segments resulting from the split. The database drops the LOB data and LOB index segments of the current partition and creates new segments for each LOB column, for each partition, even if you do not specify a new tablespace.
- If *table* contains nested table columns, then you can use the *split_nested_table_part* clause to specify the storage table names and segment attributes of the nested table segments resulting from the split. The database drops the nested table segments of the current partition and creates new segments for each nested table column, for each partition. This clause allows for multiple nested table columns in the parent table as well as multilevel nested table columns.

Oracle Database splits the corresponding local index partition, even if it is marked UNUSABLE. The database marks UNUSABLE, and you must rebuild the local index partitions corresponding to the split partitions. The new index partitions inherit their attributes from the partition being split. The database stores the new index partitions in the default tablespace of the index partition being split. If that index partition has no default tablespace, then the database uses the tablespace of the new underlying table partitions.

AT Clause

The AT clause applies only to range partitions and lets you split one range partition into two range partitions. Specify the new noninclusive upper bound for the first of the two new partitions. The value list must compare less than the original partition bound for the current partition and greater than the partition bound for the next lowest partition (if there is one).

VALUES Clause

The VALUES clause applies only to list partitions and allows you to split one list partition into two list partitions. If the table is partitioned on one key column, then use the upper branch of the *list_values* syntax to specify a list of values for that column. You can specify NULL if you have not already specified NULL for another partition in the table. If the table is partitioned on multiple key columns, then use the lower branch of the *list_values* syntax to specify a list of value lists. Each value list is enclosed in parentheses and represents a list of values for the key columns. Oracle Database creates the first new partition using the *list_values* you specify and creates the second new partition using the remaining partition values from the current partition. Therefore, the value list cannot contain all of the partition values of the current partition, nor can it contain any partition values that do not already exist for the current partition.

INTO Clause

The INTO clause lets you describe the new partitions resulting from the split.

- The AT ... INTO clause lets you describe the partitions resulting from splitting one range partition into two range partitions. In *range_partition_desc*, the keyword PARTITION is required even if you do not specify the optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, then Oracle Database assigns names of the form SYS_P*n*. Any attributes you do not specify are inherited from the current partition.
- The VALUES ... INTO clause lets you describe the partitions resulting from splitting one list partition into two list partitions. In *list_partition_desc*, the keyword PARTITION is required even if you do not specify the optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, then Oracle Database assigns names of the form SYS_P*n*. Any attributes you do not specify are inherited from the current partition.
- The INTO clause lets you split one range partition into two or more range partitions, or one list partition into two or more list partitions. If you do not specify new partition names, then Oracle Database assigns names of the form SYS_P*n*. Any attributes you do not specify are inherited from the current partition.
  - You must specify range partitions in ascending order of their partition bounds. The partition bound of the first specified range partition must be greater than the partition bound for the next lowest partition in the table (if there is one). Do not specify a partition bound for the last range partition; it will inherit the partition bound of the current partition.
  - For list partitions, all specified partition values for the new partitions must exist in the current partition. Do not specify any partition values for the last partition. Oracle Database creates the last partition using the remaining partition values from the current partition.

For range-hash composite-partitioned tables, if you specify subpartitioning for the new partitions, then you can specify only TABLESPACE and table compression for the subpartitions.

All other attributes are inherited from the current partition. If you do not specify subpartitioning for the new partitions, then their tablespace is also inherited from the current partition.

For range-list and list-list composite-partitioned tables, you cannot specify subpartitions for the new partitions at all. The list subpartitions of the split partition inherit the number of subpartitions and value lists from the current partition.

For all composite-partitioned tables for which you do not specify subpartition names for the newly created subpartitions, the newly created subpartitions inherit their names from the parent partition as follows:

- For those subpartitions in the parent partition with names of the form *partition_name* underscore (_) *subpartition_name* (for example, P1_SUBP1), Oracle Database generates corresponding names in the newly created subpartitions using the new partition names (for example P1A_SUB1 and P1B_SUB1).
- For those subpartitions in the parent partition with names of any other form, Oracle Database generates subpartition names of the form SYS_SUBP*n*.

Oracle Database splits the corresponding partition(s) in each local index defined on *table*, even if the index is marked UNUSABLE.

If *table* is the parent table of a reference-partitioned table, then you can use the *dependent_tables_clause* to propagate the partition maintenance operation you are specifying in this statement to all the reference-partitioned child tables.

Oracle Database invalidates any indexes on heap-organized tables. You can update these indexes during this operation using the *update_index_clauses*.

The *parallel_clause* lets you parallelize the split operation but does not change the default parallel attributes of the table.

ONLINE

Specify ONLINE to indicate that DML operations on the table will be allowed while splitting the table partition.

Restrictions on the ONLINE Clause

The ONLINE clause is subject to the following restrictions when splitting table partitions:

- You cannot specify the ONLINE clause for tables owned by SYS.
- You cannot specify the ONLINE clause for index-organized tables.
- You cannot specify the ONLINE clause if a domain index is defined on the table.
- You cannot specify the ONLINE clause for heap-organized tables that contain object types or on which bitmap join indexes are defined.
- Parallel DML and direct path INSERT operations require an exclusive lock on the table. Therefore, these operations are not supported concurrently with an ongoing online partition split, due to conflicting locks.

Restrictions on Splitting Table Partitions

Splitting table partitions is subject to the following restrictions:

- You cannot specify this clause for a hash partition.
- You cannot specify the *parallel_clause* for index-organized tables.

- If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can split the partition into only two new partitions.

> **See Also:**
>
> "Splitting Table Partitions: Examples"

split_table_subpartition

Use this clause to split a subpartition into multiple new subpartitions with nonoverlapping value lists.

> **Note:**
>
> Oracle Database can optimize and speed up SPLIT PARTITION and SPLIT SUBPARTITION operations if specific conditions are met. Refer to *Oracle Database VLDB and Partitioning Guide* for information on optimizing these operations.

AT Clause

The AT clause is valid only for range subpartitions. Specify the new noninclusive upper bound for the first of the two new subpartitions. The value list must compare less than the original subpartition bound for the subpartition identified by *subpartition_extended_name* and greater than the partition bound for the next lowest subpartition (if there is one).

VALUES Clause

The VALUES clause is valid only for list subpartitions. If the table is subpartitioned on one key column, then use the upper branch of the *list_values* syntax to specify a list of values for that column. You can specify NULL if you have not already specified NULL for another subpartition in the same partition. If the table is subpartitioned on multiple key columns, then use the lower branch of the *list_values* syntax to specify a list of value lists. Each value list is enclosed in parentheses and represents a list of values for the key columns. Oracle Database creates the first new subpartition using the subpartition value list you specify and creates the second new partition using the remaining partition values from the current subpartition. Therefore, the value list cannot contain all of the partition values of the current subpartition, nor can it contain any partition values that do not already exist for the current subpartition.

INTO Clause

The INTO clause lets you describe the new subpartitions resulting from the split.

- The AT ... INTO clause lets you describe the two subpartitions resulting from splitting one range partition into two range partitions. In *range_subpartition_desc*, the keyword SUBPARTITION is required even if you do not specify the optional names and attributes of the two new subpartitions. If you do not specify new subpartition names, then Oracle Database assigns names of the form SYS_SUBP*n* Any attributes you do not specify are inherited from the current subpartition.
- The VALUES ... INTO clause lets you describe the two subpartitions resulting from splitting one list partition into two list partitions. In *list_subpartition_desc*, the keyword

SUBPARTITION is required even if you do not specify the optional names and attributes of the two new subpartitions. If you do not specify new subpartition names, then Oracle Database assigns names of the form SYS_SUBP*n* Any attributes you do not specify are inherited from the current subpartition.

- The INTO clause lets you split one range subpartition into two or more range subpartitions, or one list subpartition into two or more list subpartitions. If you do not specify new subpartition names, then Oracle Database assigns names of the form SYS_SUBP*n*. Any attributes you do not specify are inherited from the current subpartition.
  - You must specify range subpartitions in ascending order of their subpartition bounds. The subpartition bound of the first specified range subpartition must be greater than the subpartition bound for the next lowest subpartition (if there is one). Do not specify a subpartition bound for the last range subpartition; it will inherit the partition bound of the current subpartition.
  - For list subpartitions, all specified subpartition values for the new subpartitions must exist in the current subpartition. Do not specify any subpartition values for the last subpartition. Oracle Database creates the last subpartition using the remaining partition values from the current subpartition.

Oracle Database splits any corresponding local subpartition index, even if it is marked UNUSABLE. The new index subpartitions inherit the names of the new table subpartitions unless those names are already held by index subpartitions. In that case, the database assigns new index subpartition names of the form SYS_SUBPn. The new index subpartitions inherit physical attributes from the parent subpartition. However, if the parent subpartition does not have a default TABLESPACE attribute, then the new subpartitions inherit the tablespace of the corresponding new table subpartitions.

Oracle Database invalidates indexes on heap-organized tables. You can update these indexes by using the *update_index_clauses*.

ONLINE

Specify ONLINE to indicate that DML operations on the table will be allowed while splitting the table subpartition.

Restrictions on the ONLINE Clause

The ONLINE clause for splitting table subpartitions is subject to the same restrictions as the ONLINE clause for splitting table partitions. Refer to Restrictions on the ONLINE Clause.

Restrictions on Splitting Table Subpartitions

Splitting table subpartitions is subject to the following restrictions:

- You cannot specify this clause for a hash subpartition.
- In subpartition descriptions, the only clauses of *partitioning_storage_clause* you can specify are TABLESPACE and table compression.
- You cannot specify the *parallel_clause* for index-organized tables.
- If *table* is an index-organized table, then you can split the subpartition into only two new subpartitions.

merge_table_partitions

The *merge_table_partitions* clause lets you merge the contents of two or more range, list, or system partitions of *table* into one new partition and then drop the original partitions. This clause is not valid for hash partitions. Use the *coalesce_table_partition* clause instead.

Specify a comma-separated list of two or more range, list, or system partitions to be merged. You can use the TO clause to specify two or more adjacent range partitions to be merged.

For each partition, use *partition* to specify a partition name or the FOR clause to specify a partition without using its name. See "References to Partitioned Tables and Indexes " for more information on the FOR clause.

- The partitions to be merged must be adjacent and must be specified in ascending order of their partition bounds if they are range partitions. List partitions and system partitions need not be adjacent in order to be merged.
- When you merge range partitions, the new partition inherits the partition bound of the highest of the original partitions.
- When you merge list partitions, the resulting partition value list is the union of the set of the partition values lists of the partitions being merged. If you merge a DEFAULT list partition with other list partitions, then the resulting partition will be the DEFAULT partition and will have the DEFAULT value.
- When you merge composite range partitions or composite list partitions, range-list or list-list composite partitions, you cannot specify subpartition descriptions. Oracle Database obtains the subpartitioning information from the subpartition template. If you have not specified a subpartition template, then the database creates one MAXVALUE subpartition from range subpartitions or one DEFAULT subpartition from list subpartitions.

Any attributes you do not specify explicitly for the new partition are inherited from table-level defaults. However, if you reuse one of the partition names for the new partition, then the new partition inherits values from the partition whose name is being reused rather than from table-level default values.

Oracle Database drops local index partitions corresponding to the selected partitions and marks UNUSABLE the local index partition corresponding to merged partition. The database also marks UNUSABLE any global indexes on heap-organized tables. You can update all these indexes during this operation using the *update_index_clauses*.

If *table* is the parent table of a reference-partitioned table, then you can use the *dependent_tables_clause* to propagate the partition maintenance operation you are specifying in this statement to all the reference-partitioned child tables.

ONLINE

Specify ONLINE to allow DML operations on the table partitions during the merge partitions operation.

Restriction on Merging Table Partitions

If *table* is an index-organized table, or if a local domain index is defined on *table*, then you can merge only two partitions at a time.

> **See Also:**
>
> "Merging Two Table Partitions: Example", "Merging Four Adjacent Range Partitions: Example", and "Working with Default List Partitions: Example"

merge_table_subpartitions

The *merge_table_subpartitions* clause lets you merge the contents of two or more range or list subpartitions of *table* into one new subpartition and then drop the original subpartitions. This clause is not valid for hash subpartitions. Use the *coalesce_hash_subpartition* clause instead.

Specify a comma-separated list of two or more range or list subpartitions to be merged. You can use the TO clause to specify two or more adjacent range subpartitions to be merged.

For each subpartition, use *subpartition* to specify a subpartition name or the FOR clause to specify a subpartition without using its name. See "References to Partitioned Tables and Indexes " for more information on the FOR clause.

The subpartitions to be merged must belong to the same partition. If they are range subpartitions, then they must be adjacent. If they are list subpartitions, then they need not be adjacent. The data in the resulting subpartition consists of the combined data from the merged subpartitions.

If you specify the INTO clause, then in the *range_subpartition_desc* or *list_subpartition_desc* you cannot specify the *range_values_clause* or *list_values_clause*, respectively. Further, the only clauses you can specify in the *partitioning_storage_clause* are the TABLESPACE clause and *table_compression*.

Any attributes you do not specify explicitly for the new subpartition are inherited from partition-level values. However, if you reuse one of the subpartition names for the new subpartition, then the new subpartition inherits values from the subpartition whose name is being reused rather than from partition-level default values.

Oracle Database merges corresponding local index subpartitions and marks the resulting index subpartition UNUSABLE. The database also marks UNUSABLE both partitioned and nonpartitioned global indexes on heap-organized tables. You can update all indexes during this operation using the *update_index_clauses*.

ONLINE

Specify ONLINE to allow DML operations on the table subpartitions during the merge subpartitions operation.

Restriction on Merging Table Subpartitions

If *table* is an index-organized table, then you can merge only two subpartitions at a time.

exchange_partition_subpart

Use the EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause to exchange the data and index segments of:

- One nonpartitioned table with:

  - one range, list, or hash partition
  - one range, list, or hash subpartition
- One range-partitioned table with the range subpartitions of a range-range or list-range composite-partitioned table partition
- One hash-partitioned table with the hash subpartitions of a range-hash or list-hash composite-partitioned table partition
- One list-partitioned table with the list subpartitions of a range-list or hash-list composite-partitioned table partition

In all cases, the structure of the table and the partition or subpartition being exchanged, including their partitioning keys, must be identical. In the case of list partitions and subpartitions, the corresponding value lists must also match.

This clause facilitates high-speed data loading when used with transportable tablespaces.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on transportable tablespaces

If *table* contains LOB columns, then for each LOB column Oracle Database exchanges LOB data and LOB index partition or subpartition segments with corresponding LOB data and LOB index segments of *table*.

If *table* has nested table columns, then for each such column Oracle Database exchanges nested table partition segments with corresponding nested table segments of the nonpartitioned table.

If *table* contains an identity column, then so must the partition or subpartition being exchanged, and vice versa. The sequence generators must both be increasing or decreasing. The sequence generators are not exchanged, so *table* and the partition or subpartition will continue to use the same sequence generators. The high water mark for both sequence generators will be adjusted so that new identity column values will not conflict with existing values.

All of the segment attributes of the two objects (including tablespace and logging) are also exchanged.

Existing statistics for the table being exchanged into the partitioned table will be exchanged. However, the global statistics for the partitioned table will not be altered. Use the DBMS_STATS.GATHER_TABLE_STATS procedure to re-create global statistics. You can set the GRANULARITY attribute equal to 'APPROX_GLOBAL AND PARTITION' to speed up the process and aggregate new global statistics based on the existing partition statistics. See *Oracle Database PL/SQL Packages and Types Reference* for more information on this packaged procedure.

Oracle Database invalidates any global indexes on the objects being exchanged. You can update the global indexes on the table whose partition is being exchanged by using either the update_global_index_clause or the update_all_indexes_clause. For the *update_all_indexes_clause*, you can specify only the keywords UPDATE INDEXES, not the subclause. Global indexes on the table being exchanged remain invalidated. The *update_global_index_clause* and *update_all_indexes_clause* do not update local indexes during an exchange operation. You can specify local index maintenance by using the

INCLUDING | EXCLUDING INDEXES clause. If you specify the *parallel_clause* with either of these clauses, then the database parallelizes the index update, not the exchange operation.

> **See Also:**
>
> "Notes on Exchanging Partitions and Subpartitions"

WITH TABLE

Specify the *table* with which the partition or subpartition will be exchanged. If you omit *schema*, then Oracle Database assumes that *table* is in your own schema.

INCLUDING | EXCLUDING INDEXES

Specify INCLUDING INDEXES if you want local index partitions or subpartitions to be exchanged with the corresponding table index (for a nonpartitioned table) or local indexes (for a hash-partitioned table). Specify EXCLUDING INDEXES if you want all index partitions or subpartitions corresponding to the partition and all the regular indexes and index partitions on the exchanged table to be marked UNUSABLE. If you omit this clause, then the default is EXCLUDING INDEXES.

WITH | WITHOUT VALIDATION

Specify WITH VALIDATION if you want Oracle Database to return an error if any rows in the exchanged table do not map into partitions or subpartitions being exchanged. Specify WITHOUT VALIDATION if you do not want Oracle Database to check the proper mapping of rows in the exchanged table. If you omit this clause, then the default is WITH VALIDATION.

exceptions_clause

See "Handling Constraint Exceptions " for information on this clause. In the context of exchanging partitions, this clause is valid only if the partitioned table has been defined with a UNIQUE constraint, and that constraint must be in DISABLE VALIDATE state. This clause is valid only for exchanging partition, not subpartitions.

CASCADE

Specify CASCADE to exchange the corresponding partition or subpartition in all reference-partitioned child tables of *table*. The reference-partitioned table hierarchies of the source and target must match.

Restrictions on CASCADE

The following restrictions apply to the CASCADE clause:

- You cannot specify CASCADE if a parent key in the reference-partitioned table hierarchy is referenced by multiple partitioning constraints.
- You cannot specify CASCADE if a domain index or an XMLIndex index is defined on any of the reference-partitioned child tables of *table*.

> **See Also:**
>
> - The DBMS_IOT package in *Oracle Database PL/SQL Packages and Types Reference* for information on the SQL scripts
> - *Oracle Database Administrator's Guide* for information on eliminating migrated and chained rows
> - *constraint* for more information on constraint checking and "Creating an Exceptions Table for Index-Organized Tables: Example"

Notes on Exchanging Partitions and Subpartitions

The following notes apply when exchanging partitions and subpartitions:

- Both tables involved in the exchange must have the same primary key, and no validated foreign keys can be referencing either of the tables unless the referenced table is empty.
- When exchanging partitioned index-organized tables:
  - The source and target table or partition must have their primary key set on the same columns, in the same order.
  - If prefix compression is enabled, then it must be enabled for both the source and the target, and with the same prefix length.
  - Both the source and target must be index organized.
  - Both the source and target must have overflow segments, or neither can have overflow segments. Also, both the source and target must have mapping tables, or neither can have a mapping table.
  - Both the source and target must have identical storage attributes for any LOB columns.

> **See Also:**
>
> "Exchanging Table Partitions: Example"

dependent_tables_clause

This clause is valid only when you are altering the parent table of a reference-partitioned table. The clause lets you specify attributes of partitions that are created by the operation for reference-partitioned child tables of the parent table.

- If the parent table is not composite partitioned, then specify one or more child tables, and for each child table specify one *partition_spec* for each partition created in the parent table.
- If the parent table is composite, then specify one or more child tables, and for each child table specify one *partition_spec* for each subpartition created in the parent table.

> **See Also:**
>
> The CREATE TABLE clause *reference_partitioning* for information on creating reference-partitioned tables and *Oracle Database VLDB and Partitioning Guide* for information on partitioning by reference in general

UNUSABLE LOCAL INDEXES Clauses

These two clauses modify the attributes of local index partitions and index subpartitions corresponding to *partition*, depending on whether you are modifying a partition or subpartition.

- UNUSABLE LOCAL INDEXES marks UNUSABLE the local index partition or index subpartition associated with *partition*.
- REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index partition or index subpartition associated with *partition*.

Restrictions on UNUSABLE LOCAL INDEXES

This clause is subject to the following restrictions:

- You cannot specify this clause with any other clauses of the *modify_table_partition* clause.
- You cannot specify this clause in the *modify_table_partition* clause for a partition that has subpartitions. However, you can specify this clause in the *modify_table_subpartition* clause.

update_index_clauses

Use the *update_index_clauses* to update the indexes on *table* as part of the table partitioning operation. When you perform DDL on a table partition, if an index is defined on *table*, then Oracle Database invalidates the entire index, not just the partitions undergoing DDL. This clause lets you update the index partition you are changing during the DDL operation, eliminating the need to rebuild the index after the DDL.

The *update_index_clauses* are not needed, and are not valid, for partitioned index-organized tables. Index-organized tables are primary key based, so Oracle can keep global indexes USABLE during operations that move data but do not change its value.

update_global_index_clause

Use this clause to update only global indexes on *table*. Oracle Database marks UNUSABLE all local indexes on *table*.

UPDATE GLOBAL INDEXES

Specify UPDATE GLOBAL INDEXES to update the global indexes defined on *table*.

Restriction on Updating Global Indexes

If the global index is a global domain index defined on a LOB column, then Oracle Database marks the domain index UNUSABLE instead of updating it.

INVALIDATE GLOBAL INDEXES

Specify `INVALIDATE GLOBAL INDEXES` to invalidate the global indexes defined on *`table`*.

If you specify neither, then Oracle Database invalidates the global indexes.

Restrictions on Invalidating Global Indexes

This clause is supported only for global indexes. It is not supported for index-organized tables. In addition, this clause updates only indexes that are `USABLE` and `VALID`. `UNUSABLE` indexes are left unusable, and `INVALID` global indexes are ignored.

update_all_indexes_clause

Use this clause to update all indexes on *`table`*.

update_index_partition

This clause is valid only for operations on table partitions and affects only local indexes.

- The *`index_partition_description`* lets you specify physical attributes, tablespace storage, and logging for each partition of each local index. If you specify only the `PARTITION` keyword, then Oracle Database updates the index partition as follows:
  - For operations on a single table partition (such as `MOVE PARTITION` and `SPLIT PARTITION`), the corresponding index partition inherits the attributes of the affected index table partition, Oracle Database does not generate names for new index partitions, so any new index partitions resulting from this operation inherit their names from the corresponding new table partition.
  - For `MERGE PARTITION` operations, the resulting local index partition inherits its name from the resulting table partition and inherits its attributes from the local index.

  For a domain index, you can use the `PARAMETERS` clause to specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The `PARAMETERS` clause is valid only for domain indexes, and is the only part of the *`index_partition_description`* you can specify for a domain index.

  > **See Also:**
  >
  > *Oracle Database Data Cartridge Developer's Guide* for more information on domain indexes

- For a composite-partitioned index, the *`index_subpartition_clause`* lets you specify tablespace storage for each subpartition. Refer to the index_subpartition_clause (in `CREATE INDEX`) for more information on this component of the *`update_index_partition`* clause.

For information on the `USABLE` and `UNUSABLE` keywords, refer to `ALTER INDEX` ... USABLE | UNUSABLE.

update_index_subpartition

This clause is valid only for operations on subpartitions of composite-partitioned tables and affects only local indexes on composite-partitioned tables. It lets you specify tablespace storage for one or more subpartitions.

Restrictions on Updating All Indexes

The following restrictions apply to the *`update_all_indexes_clause`*:

- You cannot specify this clause for index-organized tables.
- When you exchange a partition or subpartition with the *exchange_partition_subpart* clause, the *update_all_indexes_clause* is applicable only to global indexes. Therefore, you cannot specify the *update_index_partition* or *update_index_subpartition* clauses. You can, however, specify local index maintenance during an exchange operation by using the INCLUDING | EXCLUDING INDEXES clause.

> **See Also:**
>
> "Updating Global Indexes: Example" and "Updating Partitioned Indexes: Example"

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for queries and DML on the table.

For complete information on this clause, refer to *parallel_clause* in the documentation on `CREATE TABLE`.

Restrictions on Changing Table Parallelization

Changing parallelization is subject to the following restrictions:

- If *table* contains any columns of LOB or user-defined object type, then subsequent `INSERT`, `UPDATE`, and `DELETE` operations on *table* are executed serially without notification. Subsequent queries, however, are executed in parallel.
- If you specify the *parallel_clause* in conjunction with the *move_table_clause*, then the parallelism applies only to the move, not to subsequent DML and query operations on the table.

> **See Also:**
>
> "Specifying Parallel Processing: Example"

filter_condition

This clause lets you specify which rows to preserve during the following `ALTER TABLE` operations: moving, splitting, or merging table partitions or subpartitions; moving a table; or converting a nonpartitioned table to a partitioned table. The database preserves only the rows that satisfy the condition specified in the *where_clause*. Refer to the *where_clause* in the documentation on `SELECT` for the full semantics of this clause.

Restrictions on Filter Conditions

The following restrictions apply to the *filter_condition* clause:

- Filter conditions are supported only for heap-organized tables.

- Filter conditions can refer only to columns in the table being altered. Filter conditions cannot contain operations, such as joins or subqueries, that reference other database objects.
- Filter conditions are unsupported for tables with primary or unique keys that are referenced by enabled foreign keys.

Restrictions and Notes on Using Filter Conditions with Online Operations

The following restrictions and notes apply when you specify a filter condition for an online `ALTER TABLE` operation:

- You cannot specify both the *`filter_condition`* and `ONLINE` clauses if supplemental logging is enabled.
- When you specify both the *`filter_condition`* and `ONLINE` clauses, DML operations on the table are allowed during the `ALTER TABLE` operation. The filter condition does not have a direct effect on the concurrent DML operations. However, consider this combination carefully, because the filter operation and the DML operations could unintentionally conflict, as follows:
  - Inserts into a nonpartitioned table will succeed. Inserts into a partitioned table will succeed if they do not violate the partitioning key criteria.
  - Delete operations will apply only to rows that are preserved by the filter condition throughout the `ALTER TABLE` operation.
  - Update operations will apply only to rows that are preserved by the filter condition throughout the `ALTER TABLE` operation. These update operations will succeed, regardless of whether the update operation would have disqualified the rows for preservation by the filter condition.
  - Rows that do not qualify for preservation by the filter condition at the onset of the `ALTER TABLE` operation will not be preserved, regardless of whether an update operation would qualify the rows for preservation.

allow_disallow_clustering

This clause is valid for tables that use attribute clustering. It lets you allow or disallow attribute clustering for data movement that occurs during the move table operation specified by the *`move_table_clause`*, and the table partition and subpartition maintenance operations specified by the *`coalesce_table_[sub]partition`*, *`merge_table_[sub]partitions`*, *`move_table_[sub]partition`*, and *`split_table_[sub]partition`* clauses.

- Specify `ALLOW CLUSTERING` to allow attribute clustering for data movement. This clause overrides a `NO ON DATA MOVEMENT` setting in the DDL that created or altered the table.
- Specify `DISALLOW CLUSTERING` to disallow attribute clustering for data movement. This clause overrides a `YES ON DATA MOVEMENT` setting in the DDL that created or altered the table.

The *`allow_disallow_clustering`* clause has no effect if you specify it for a table that does not use attribute clustering.

> **See Also:**
>
> clustering_when clause of CREATE TABLE for more information on the NO ON DATA MOVEMENT and YES ON DATA MOVEMENT clauses

{ DEFERRED | IMMEDIATE } INVALIDATION

This clause lets you control when the database invalidates dependent cursors while performing table partition maintenance operations.

- If you specify DEFERRED INVALIDATION, then the database avoids or defers invalidating dependent cursors, when possible.
- If you specify IMMEDIATE INVALIDATION, then the database immediately invalidates dependent cursors, as it did in Oracle Database 12*c* Release 1 (12.1) and prior releases. This is the default.

If you omit this clause, then the value of the CURSOR_INVALIDATION initialization parameter determines when cursors are invalidated.

You can specify this clause only when performing table partition maintenance operations; it is not supported for any other ALTER TABLE operations.

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* for more information on cursor invalidation
> - *Oracle Database Reference* for more information in the CURSOR_INVALIDATION initialization parameter

move_table_clause

The *move_table_clause* lets you relocate data of a nonpartitioned table or of a partition of a partitioned table into a new segment, optionally in a different tablespace, and optionally modify any of its storage attributes.

You can also move any LOB data segments associated with the table or partition using the *LOB_storage_clause* and *varray_col_properties* clause. LOB items not specified in this clause are not moved.

If you move the table to a different tablespace and the COMPATIBLE parameter is set to 10.0 or higher, then Oracle Database leaves the storage table of any nested table columns in the tablespace in which it was created. If COMPATIBLE is set to any value less than 10.0, then the database silently moves the storage table to the new tablespace along with the table.

ONLINE Clause

Specify ONLINE if you want DML operations on the table to be allowed while the table is being moved.

Restrictions on Moving Tables Online

Moving tables online is subject to the following restrictions:

- You cannot combine this clause with any other clause in the same statement.
- You cannot specify this clause for a partitioned index-organized table.
- You cannot specify this clause if a domain index is defined on the table.
- Parallel DML and direct path INSERT operations require an exclusive lock on the table. Therefore, these operations are not supported concurrently with an ongoing online table MOVE, due to conflicting locks.
- You cannot specify this clause for index-organized tables that contain any LOB, VARRAY, Oracle-supplied type, or user-defined object type columns.

index_org_table_clause

For an index-organized table, the *index_org_table_clause* of the *move_table_clause* lets you additionally specify overflow segment attributes. The *move_table_clause* rebuilds the primary key index of the index-organized table. The overflow data segment is not rebuilt unless the OVERFLOW keyword is explicitly stated, with two exceptions:

- If you alter the values of PCTTHRESHOLD or the INCLUDING column as part of this ALTER TABLE statement, then the overflow data segment is rebuilt.
- If you explicitly move any of out-of-line columns (LOBs, varrays, nested table columns) in the index-organized table, then the overflow data segment is also rebuilt.

The index and data segments of LOB columns are not rebuilt unless you specify the LOB columns explicitly as part of this ALTER TABLE statement.

mapping_table_clause

Specify MAPPING TABLE if you want Oracle Database to create a mapping table if one does not already exist. If it does exist, then the database moves the mapping table along with the index-organized table, and marks any bitmapped indexes UNUSABLE. The new mapping table is created in the same tablespace as the parent table.

Specify NOMAPPING to instruct the database to drop an existing mapping table.

Refer to *mapping_table_clauses* (in CREATE TABLE) for more information on this clause.

Restriction on Mapping Tables

You cannot specify NOMAPPING if any bitmapped indexes have been defined on *table*.

prefix_compression

Use the *prefix_compression* clause to enable or disable prefix compression in an index-organized table.

- COMPRESS enables prefix compression, which eliminates repeated occurrence 1of primary key column values in index-organized tables. Use *integer* to specify the prefix length (number of prefix columns to compress).

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.
- NOCOMPRESS disables prefix compression in index-organized tables. This is the default.

TABLESPACE *tablespace*

Specify the tablespace into which the rebuilt index-organized table is to be stored.

LOB_storage_clause

Use this clause to move a LOB segment to a different tablespace. You cannot use this clause to move a LOB segment if the table contains a `LONG` column. Instead, you must either convert the `LONG` column to a LOB, or you must export the table, re-create the table specifying the desired tablespace storage for the LOB column, and re-import the table data.

UPDATE INDEXES

This clause is valid only when performing online or offline moves of heap-organized tables. It allows you to update all global indexes on the table.

You can optionally change the tablespace for an index or index partition, as follows:

- Specify the *`segment_attributes_clause`* to change the tablespace of a nonpartitioned global index. Within this clause, you can specify only the `TABLESPACE` clause.
- Specify the *`update_index_partition`* clause to change the tablespace for a partition of a partitioned global index. Within this clause, you can specify only the `TABLESPACE` clause of the *`segment_attributes_clause`*.

Restrictions on Moving Tables

Moving tables is subject to the following restrictions:

- If you specify `MOVE`, then it must be the first clause in the `ALTER TABLE` statement, and the only clauses outside this clause that are allowed are the *`physical_attributes_clause`*, the *`parallel_clause`*, and the *`LOB_storage_clause`*.
- You cannot move a table containing a `LONG` or `LONG RAW` column.
- You cannot `MOVE` an entire partitioned table (either heap- or index-organized). You must move individual partitions or subpartitions.

> **Note:**
>
> For any LOB columns you specify in a *`move_table_clause`*:
>
> - Oracle Database drops the old LOB data segment and corresponding index segment and creates new segments, even if you do not specify a new tablespace.
> - If the LOB index in *`table`* resided in a different tablespace from the LOB data, then Oracle Database collocates the LOB index in the same tablespace with the LOB data after the move.

> **See Also:**
>
> *move_table_partition* and *move_table_subpartition*

modify_to_partitioned

This clause lets you change a nonpartitioned heap table to a partitioned heap table, including indexes, in both offline and online mode. Further, you can repartition partitioned tables online and offline with this clause.

You can change a nonpartitioned table into any type of partitioned or composite partitioned table. All data in the original table is preserved. Newly created partitions or subpartitions will store data in same tablespace as the original table, unless you specify otherwise in the `table_partitioning_clauses`. All triggers, constraints, and VPD policies defined on the original table are preserved. If table compression is defined on the original table, then the partitioned table will use the same type of table compression.

table_partitioning_clauses

Use this clause to specify the partitioning attributes for the table. This clause has the same semantics here as it has for the `CREATE TABLE` statement. Refer to the `CREATE TABLE` *table_partitioning_clauses* for the full semantics of this clause.

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed while changing to a partitioned table.

UPDATE INDEXES

Use this clause to specify how existing indexes on the table are converted into global partitioned indexes or local partitioned indexes.

- For `index`, specify the name of an existing index on the table.
- Specify the `local_partitioned_index` clause to convert `index` into a local partitioned index. This clause has the same semantics here as it has for the `CREATE INDEX` statement. Refer to the clause *local_partitioned_index* in the documentation on `CREATE INDEX` for the full semantics of this clause.
- Specify the `global_partitioned_index` clause to convert `index` into a global partitioned index. This clause has the same semantics here as it has for the `CREATE INDEX` statement. Refer to the clause *global_partitioned_index* in the documentation on `CREATE INDEX` for the full semantics of this clause.
- Specify the `GLOBAL` keyword to allow prefixed partitioned and nonpartitioned global indexes to retain their global shape. This clause prevents such indexes from being converted to local partitioned indexes; it has no effect on nonprefixed global indexes.

If you specify only the `UPDATE INDEXES` keywords, or omit the `UPDATE INDEXES` clause altogether, then existing indexes are converted as follows:

- Nonprefixed indexes retain their original shape: normal indexes are converted to nonpartitioned global indexes, nonpartitioned global indexes remain the same, and partitioned global indexes remain the same and retain their partitioning shape.
- Prefixed indexes are converted to local partitioned indexes. Prefixed indexes include partitioning keys in the index definition, but the index definition is not limited to including only the partitioning keys.
- Bitmap indexes are converted to local partitioned indexes, regardless of whether they are prefixed or not.

Restrictions on Changing a Nonpartitioned Table to a Partitioned Table

The following restrictions apply to the *modify_to_partitioned* clause:

- You cannot specify this clause for an index-organized table.
- You cannot specify this clause if a domain index is defined on the table.
- You cannot specify ONLINE when changing a nonpartitioned table to a reference-partitioned child table. This operation is supported only in offline mode.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on converting a nonpartitioned table into a partitioned table

modify_opaque_type

Use the *modify_opaque_type* clause to instruct the database to store the specified abstract data type or XMLType in an ANYDATA column using unpacked storage.

You can specify any abstract data type with this clause. However, it is primarily useful because it allows you to specify the following data types, which cannot be stored in an ANYDATA column using conventional storage:

- XMLType
- Abstract data types that contain one or more attributes of type XMLType, CLOB, BLOB, or NCLOB.

When you use **unpacked storage**, data types are stored in system-generated hidden columns that are associated with the ANYDATA column. You can insert and query these data types as you would data types that are stored in an ANYDATA column using conventional storage.

anydata_column

Specify the name of a column of type ANYDATA. If *type_name* is an abstract data type that does not contain an attribute of type XMLType, CLOB, BLOB, or NCLOB, then *anydata_column* must be empty.

type_name

Specify the name of one or more abstract data types or XMLType. The abstract data type can contain an attribute of type XMLType, CLOB, BLOB, or NCLOB. The type can be EDITIONABLE. When you subsequently insert these data types into *anydata_column*, they will use unpacked storage. If you previously specified this clause for the same *anydata_column*, then unpacked storage will continue to be used for the previously specified data types as well as the newly specified data types.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the ANYDATA type and "Unpacked Storage in ANYDATA Columns: Example"

enable_disable_clause

The *enable_disable_clause* lets you specify whether and how Oracle Database should apply an integrity constraint. The DROP and KEEP clauses are valid only when you are disabling a unique or primary key constraint.

> **See Also:**
>
> - The *enable_disable_clause* (in CREATE TABLE) for a complete description of this clause, including notes and restrictions that relate to this statement

TABLE LOCK

Oracle Database permits DDL operations on a table only if the table can be locked during the operation. Such table locks are not required during DML operations.

> **Note:**
>
> Table locks are not acquired on temporary tables.

- Specify ENABLE TABLE LOCK to enable table locks, thereby allowing DDL operations on the table. All currently executing transactions must commit or roll back before Oracle Database enables the table lock.

  > **Note:**
  >
  > Oracle Database waits until active DML transactions in the database have completed before locking the table. Sometimes the resulting delay is considerable.

- Specify DISABLE TABLE LOCK to disable table locks, thereby preventing DDL operations on the table.

  > **Note:**
  >
  > Parallel DML operations are not performed when the table lock of the target table is disabled.

ALL TRIGGERS

Use the ALL TRIGGERS clause to enable or disable all triggers associated with the table.

- Specify ENABLE ALL TRIGGERS to enable all triggers associated with the table. Oracle Database fires the triggers whenever their triggering condition is satisfied.

  To enable a single trigger, use the *enable_clause* of ALTER TRIGGER.

> **See Also:**
>
> CREATE TRIGGER , ALTER TRIGGER , and "Enabling Triggers: Example"

- Specify `DISABLE ALL TRIGGERS` to disable all triggers associated with the table. Oracle Database does not fire a disabled trigger even if the triggering condition is satisfied.

CONTAINER_MAP

Use the `CONTAINER_MAP` clause to enable or disable the table to be queried using a container map.

- Specify `ENABLE CONTAINER_MAP` to enable the table to be queried using a container map.
- Specify `DISABLE CONTAINER_MAP` to disable the table from being queried using a container map.

CONTAINERS_DEFAULT

Use the `CONTAINERS_DEFAULT` clause to enable or disable the table for the `CONTAINERS` clause.

- Specify `ENABLE CONTAINERS_DEFAULT` to enable the table for the `CONTAINERS` clause.
- Specify `DISABLE CONTAINERS_DEFAULT` to disable the table for the `CONTAINERS` clause.

Examples

Adding Constraints to Tables: Example

The following statements create a new table to manipulate data and display the information in the newly created table:

```
CREATE TABLE JOBS_Temp AS SELECT * FROM HR.JOBS;

SELECT * FROM JOBS_Temp WHERE MIN_SALARY < 3000;

JOB_ID        JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
PU_CLERK   Purchasing Clerk                      2500     5500
ST_CLERK   Stock Clerk                          2008     5000
SH_CLERK   Shipping Clerk                      2500     5500
```

The following statement updates the column values to a higher value:

```
UPDATE JOBS_Temp SET MIN_SALARY = 2300 WHERE MIN_SALARY < 2010;
```

The following statement adds a constraint:

```
ALTER TABLE JOBS_Temp ADD CONSTRAINT chk_sal_min CHECK (MIN_SALARY >=2010);
```

The following statement displays the table information:

```
SELECT * FROM JOBS_Temp WHERE MIN_SALARY < 3000;

JOB_ID       JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
PU_CLERK   Purchasing Clerk                      2500     5500
ST_CLERK   Stock Clerk                          2300     5000
SH_CLERK   Shipping Clerk                      2500     5500
```

The following statement displays the constraint:

```
SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS WHERE TABLE_NAME='JOBS_TEMP';

CONSTRAINT_NAME
--------------------------------------------------------------------------------
SYS_C008830
CHK_SAL_MIN
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Adding Constraints to Tables*

Collection Retrieval: Example

The following statement modifies nested table column `ad_textdocs_ntab` in the sample table `sh.print_media` so that when queried it returns actual values instead of locators:

```
ALTER TABLE print_media MODIFY NESTED TABLE ad_textdocs_ntab
   RETURN AS VALUE;
```

Specifying Parallel Processing: Example

The following statement specifies parallel processing for queries to the sample table `oe.customers`:

```
ALTER TABLE customers
   PARALLEL;
```

Changing the State of a Constraint: Examples

The following statement places in `ENABLE VALIDATE` state an integrity constraint named `emp_manager_fk` in the `employees` table:

```
ALTER TABLE employees
   ENABLE VALIDATE CONSTRAINT emp_manager_fk
   EXCEPTIONS INTO exceptions;
```

Each row of the `employees` table must satisfy the constraint for Oracle Database to enable the constraint. If any row violates the constraint, then the constraint remains disabled. The database lists any exceptions in the table `exceptions`. You can also identify the exceptions in the `employees` table with the following statement:

```
SELECT e.*
   FROM employees e, exceptions ex
   WHERE e.rowid = ex.row_id
      AND ex.table_name = 'EMPLOYEES'
      AND ex.constraint = 'EMP_MANAGER_FK';
```

The following statement tries to place in `ENABLE NOVALIDATE` state two constraints on the `employees` table:

```
ALTER TABLE employees
   ENABLE NOVALIDATE PRIMARY KEY
   ENABLE NOVALIDATE CONSTRAINT emp_last_name_nn;
```

This statement has two `ENABLE` clauses:

- The first places a primary key constraint on the table in `ENABLE NOVALIDATE` state.
- The second places the constraint named `emp_last_name_nn` in `ENABLE NOVALIDATE` state.

In this case, Oracle Database enables the constraints only if both are satisfied by each row in the table. If any row violates either constraint, then the database returns an error and both constraints remain disabled.

Consider the foreign key constraint on the `location_id` column of the `departments` table, which references the primary key of the `locations` table. The following statement disables the primary key of the `locations` table:

```
ALTER TABLE locations
   MODIFY PRIMARY KEY DISABLE CASCADE;
```

The unique key in the `locations` table is referenced by the foreign key in the `departments` table, so you must specify `CASCADE` to disable the primary key. This clause disables the foreign key as well.

Creating an Exceptions Table for Index-Organized Tables: Example

The following example creates the `except_table` table to hold rows from the index-organized table `hr.countries` that violate the primary key constraint:

```
EXECUTE DBMS_IOT.BUILD_EXCEPTIONS_TABLE ('hr', 'countries', 'except_table');

ALTER TABLE countries
   ENABLE PRIMARY KEY
   EXCEPTIONS INTO except_table;
```

To specify an exception table, you must have the privileges necessary to insert rows into the table. To examine the identified exceptions, you must have the privileges necessary to query the exceptions table.

> **See Also:**
>
> INSERT and SELECT for information on the privileges necessary to insert rows into tables

Disabling a CHECK Constraint: Example

The following statement defines and disables a `CHECK` constraint on the `employees` table:

```
ALTER TABLE employees ADD CONSTRAINT check_comp
   CHECK (salary + (commission_pct*salary) <= 5000)
   DISABLE;
```

The constraint `check_comp` ensures that no employee's total compensation exceeds $5000. The constraint is disabled, so you can increase an employee's compensation above this limit.

Enabling Triggers: Example

The following statement enables all triggers associated with the `employees` table:

```
ALTER TABLE employees
   ENABLE ALL TRIGGERS;
```

Deallocating Unused Space: Example

The following statement frees all unused space for reuse in table `employees`, where the high water mark is above `MINEXTENTS`:

```
ALTER TABLE employees
    DEALLOCATE UNUSED;
```

Modifying the Collation of a Column for Fine-Grained Case-Insensitivity: Example

This example shows how to modify a column to be case-insensitive. First, create and populate table `students` as follows:

```
CREATE TABLE students (last_name VARCHAR2(20), id NUMBER);

INSERT INTO students VALUES('Dodd', 364);
INSERT INTO students VALUES('de Niro', 132);
INSERT INTO students VALUES('Vogel', 837);
INSERT INTO students VALUES('van der Kamp', 549);
INSERT INTO students VALUES('van Der Meer', 624);
```

The following statement returns column `last_name` in alphabetical order. Notice that the results are case-sensitive; lowercase letters are ordered after uppercase letters.

```
SELECT last_name, id
  FROM students
  ORDER BY last_name;

LAST_NAME                    ID
-------------------- ----------
Dodd                        364
Vogel                       837
de Niro                     132
van Der Meer                624
van der Kamp                549
```

The following statement changes the data-bound collation of column `last_name` to case-insensitive collation `BINARY_CI`:

```
ALTER TABLE students
  MODIFY (last_name COLLATE BINARY_CI);
```

The following statement again returns column `last_name` in alphabetical order. Notice that the results are now case-insensitive:

```
SELECT last_name, id
  FROM students
  ORDER BY last_name;

LAST_NAME                    ID
-------------------- ----------
```

```
de Niro                     132
Dodd                        364
van der Kamp                549
van Der Meer                624
Vogel                       837
```

Renaming a Column: Example

The following example renames the `credit_limit` column of the sample table `oe.customers` to `credit_amount`:

```
ALTER TABLE customers
   RENAME COLUMN credit_limit TO credit_amount;
```

Dropping a Column: Example

This statement illustrates the *`drop_column_clause`* with `CASCADE CONSTRAINTS`. Assume table `t1` is created as follows:

```
CREATE TABLE t1 (
   pk NUMBER PRIMARY KEY,
   fk NUMBER,
   c1 NUMBER,
   c2 NUMBER,
   CONSTRAINT ri FOREIGN KEY (fk) REFERENCES t1,
   CONSTRAINT ck1 CHECK (pk > 0 and c1 > 0),
   CONSTRAINT ck2 CHECK (c2 > 0)
);
```

An error will be returned for the following statements:

```
/* The next two statements return errors:
ALTER TABLE t1 DROP (pk);  -- pk is a parent key
ALTER TABLE t1 DROP (c1);  -- c1 is referenced by multicolumn
                           -- constraint ck1
```

Submitting the following statement drops column `pk`, the primary key constraint, the foreign key constraint, `ri`, and the check constraint, `ck1`:

```
ALTER TABLE t1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then `CASCADE CONSTRAINTS` is not required. For example, assuming that no other referential constraints from other tables refer to column `pk`, then it is valid to submit the following statement without the `CASCADE CONSTRAINTS` clause:

```
ALTER TABLE t1 DROP (pk, fk, c1);
```

Dropping Unused Columns: Example

The following statements create a new table to manipulate data and display the information in the newly created table:

```
CREATE TABLE JOBS_Temp AS SELECT * FROM HR.JOBS;

SELECT * FROM JOBS_Temp WHERE MAX_SALARY > 20000;

JOB_ID       JOB_TITLE                  MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                       20080      40000
```

```
AD_VP       Administration Vice President        15000      30000
SA_MAN       Sales Manager                  10000      20080
```

The following statement adds two new columns:

```
ALTER TABLE JOBS_Temp ADD (DUMMY1 NUMBER(2), DUMMY2 NUMBER(2));
```

The following statements inserts values into the newly added columns:

```
INSERT INTO JOBS_Temp(JOB_ID, JOB_TITLE, DUMMY1, DUMMY2) VALUES ('D','DUMMY',10,20);

INSERT INTO JOBS_Temp(JOB_ID, JOB_TITLE, DUMMY1, DUMMY2) VALUES ('D','DUMMY',10,20)
```

The following statement sets the newly added columns to unused:

```
ALTER TABLE JOBS_TEMP SET UNUSED (DUMMY1, DUMMY2);
```

The following statement displays the count of unused columns:

```
SELECT * FROM USER_UNUSED_COL_TABS WHERE TABLE_NAME='JOBS_TEMP';

TABLE_NAM      COUNT
--------- ----------
JOBS_TEMP       2
```

The following statement drops the unused columns:

```
ALTER TABLE JOBS_TEMP DROP UNUSED COLUMNS;
```

The following statement displays the table information:

```
SELECT * FROM JOBS_TEMP;

JOB_ID       JOB_TITLE                   MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                    20080      40000
AD_VP       Administration Vice President        15000      30000
AD_ASST    Administration Assistant             3000     6000
FI_MGR       Finance Manager                 8200       16000
FI_ACCOUNT Accountant                     4200     9000
AC_MGR       Accounting Manager                 8200       16000
AC_ACCOUNT Public Accountant                 4200     9000
SA_MAN       Sales Manager                  10000      20080
SA_REP       Sales Representative              6000       12008
PU_MAN       Purchasing Manager                 8000       15000
PU_CLERK   Purchasing Clerk                 2500     5500
ST_MAN       Stock Manager                   5500     8500
ST_CLERK   Stock Clerk                      2008     5000
SH_CLERK   Shipping Clerk                  2500     5500
IT_PROG    Programmer                     4000      10000
MK_MAN       Marketing Manager                9000       15000
MK_REP       Marketing Representative            4000     9000
HR_REP       Human Resources Representative         4000     9000
PR_REP       Public Relations Representative         4500      10500
D        DUMMY
D        DUMMY
```

Live SQL:

View and run a related example on Oracle Live SQL at *Dropping Unused Columns*

Modifying Index-Organized Tables: Examples

This statement modifies the `INITRANS` parameter for the index segment of index-organized table `countries_demo`, which is based on `hr.countries`:

```
ALTER TABLE countries_demo INITRANS 4;
```

The following statement adds an overflow data segment to index-organized table `countries`:

```
ALTER TABLE countries_demo ADD OVERFLOW;
```

This statement modifies the `INITRANS` parameter for the overflow data segment of index-organized table `countries`:

```
ALTER TABLE countries_demo OVERFLOW INITRANS 4;
```

Splitting Table Partitions: Examples

The following statement splits the old partition `sales_q4_2000` in the sample table `sh.sales`, creating two new partitions, naming one `sales_q4_2000b` and reusing the name of the old partition for the other:

```
ALTER TABLE sales SPLIT PARTITION SALES_Q4_2000
   AT (TO_DATE('15-NOV-2000','DD-MON-YYYY'))
   INTO (PARTITION SALES_Q4_2000, PARTITION SALES_Q4_2000b);
```

The following statement splits the old partition `sales_q1_2002` into three new partitions `sales_jan_2002`, `sales_feb_2002`, and `sales_mar_2002`:

```
ALTER TABLE sales SPLIT PARTITION SALES_Q1_2002 INTO (
 PARTITION SALES_JAN_2002 VALUES LESS THAN (TO_DATE('01-FEB-2002','DD-MON-
YYYY')),
 PARTITION SALES_FEB_2002 VALUES LESS THAN (TO_DATE('01-MAR-2002','DD-MON-
YYYY')),
 PARTITION SALES_MAR_2002);
```

The following statements create a partitioned version of the pm.print_media table. The `LONG` column in the print_media table has been converted to LOB. The table is stored in tablespaces created in "Creating Oracle Managed Files: Examples". The object types underlying the `ad_textdocs_ntab` and `ad_header` columns are created in the script that creates the `pm` sample schema:

```
CREATE TABLE print_media_part (
    product_id NUMBER(6),
    ad_id             NUMBER(6),
    ad_composite      BLOB,
    ad_sourcetext     CLOB,
    ad_finaltext      CLOB,
    ad_fltextn        NCLOB,
    ad_textdocs_ntab  TEXTDOC_TAB,
    ad_photo          BLOB,
```

```
    ad_graphic          BFILE,
    ad_header           ADHEADER_TYP)
  NESTED TABLE ad_textdocs_ntab STORE AS textdoc_nt
  PARTITION BY RANGE (product_id)
    (PARTITION p1 VALUES LESS THAN (100),
     PARTITION p2 VALUES LESS THAN (200));
```

The following statement splits partition p2 of that table into partitions p2a and p2b:

```
ALTER TABLE print_media_part
   SPLIT PARTITION p2 AT (150) INTO
   (PARTITION p2a TABLESPACE omf_ts1
      LOB (ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2),
   PARTITION p2b
      LOB (ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2))
   NESTED TABLE ad_textdocs_ntab INTO (PARTITION nt_p2a, PARTITION nt_p2b);
```

In both partitions p2a and p2b, Oracle Database creates the LOB segments for columns ad_photo and ad_composite in tablespace omf_ts2. The LOB segments for the remaining columns in partition p2a are stored in tablespace omf_ts1. The LOB segments for the remaining columns in partition p2b remain in the tablespaces in which they resided prior to this ALTER statement. However, the database creates new segments for all the LOB data and LOB index segments, even if they are not moved to a new tablespace.

The database also creates new segments for nested table column ad_textdocs_ntab. The storage tables is those new segments are nt_p2a and nt_p2b.

Merging Two Table Partitions: Example

The following statement merges back into one partition the partitions created in "Splitting Table Partitions: Examples":

```
ALTER TABLE sales
   MERGE PARTITIONS sales_q4_2000, sales_q4_2000b
   INTO PARTITION sales_q4_2000;
```

The next statement reverses the example in "Splitting Table Partitions: Examples":

```
ALTER TABLE print_media_part
   MERGE PARTITIONS p2a, p2b INTO PARTITION p2ab TABLESPACE example
   NESTED TABLE ad_textdocs_ntab STORE AS nt_p2ab;
```

Merging Four Adjacent Range Partitions: Example

The following statement merges four adjacent range partitions, sales_q1_2000, sales_q2_2000, sales_q3_2000, and sales_q4_2000 into one partition sales_all_2000:

```
ALTER TABLE sales
  MERGE PARTITIONS sales_q1_2000 TO sales_q4_2000
  INTO PARTITION sales_all_2000;
```

Adding a Table Partition with a LOB and Nested Table Storage: Examples

The following statement adds a partition p3 to the print_media_part table (see preceding example) and specifies storage characteristics for the BLOB, CLOB, and nested table columns of that table:

```
ALTER TABLE print_media_part ADD PARTITION p3 VALUES LESS THAN (400)
  LOB(ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts1)
  LOB(ad_sourcetext, ad_finaltext) STORE AS (TABLESPACE omf_ts2)
  NESTED TABLE ad_textdocs_ntab STORE AS nt_p3;
```

The LOB data and LOB index segments for columns ad_photo and ad_composite in partition p3 will reside in tablespace omf_ts1. The remaining attributes for these LOB columns will be inherited first from the table-level defaults, and then from the tablespace defaults.

The LOB data segments for columns ad_source_text and ad_finaltext will reside in the omf_ts2 tablespace, and will inherit all other attributes first from the table-level defaults, and then from the tablespace defaults.

The partition for the storage table for nested table storage column ad_textdocs_ntab corresponding to partition p3 of the base table is named nt_p3 and inherits all other attributes first from the table-level defaults, and then from the tablespace defaults.

Adding Multiple Partitions to a Table: Example

The following statement adds three partitions to the table print_media_part created in "Splitting Table Partitions: Examples":

```
ALTER TABLE print_media_part ADD
  PARTITION p3 values less than (300),
  PARTITION p4 values less than (400),
  PARTITION p5 values less than (500);
```

Working with Default List Partitions: Example

The following statements use the list partitioned table created in "List Partitioning Example". The first statement splits the existing default partition into a new south partition and a default partition:

```
ALTER TABLE list_customers SPLIT PARTITION rest
   VALUES ('MEXICO', 'COLOMBIA')
   INTO (PARTITION south, PARTITION rest);
```

The next statement merges the resulting default partition with the asia partition:

```
ALTER TABLE list_customers
   MERGE PARTITIONS asia, rest INTO PARTITION rest;
```

The next statement re-creates the asia partition by splitting the default partition:

```
ALTER TABLE list_customers SPLIT PARTITION rest
   VALUES ('CHINA', 'THAILAND')
   INTO (PARTITION asia, PARTITION rest);
```

Dropping a Table Partition: Example

The following statement drops partition p3 created in "Adding a Table Partition with a LOB and Nested Table Storage: Examples":

```
ALTER TABLE print_media_part DROP PARTITION p3;
```

Exchanging Table Partitions: Example

This example creates the table exchange_table with the same structure as the partitions of the list_customers table created in "List Partitioning Example". It then replaces partition rest of table list_customers with table exchange_table without exchanging local index partitions with corresponding indexes on exchange_table and without verifying that data in exchange_table falls within the bounds of partition rest:

```
CREATE TABLE exchange_table (
   customer_id     NUMBER(6),
   cust_first_name VARCHAR2(20),
   cust_last_name  VARCHAR2(20),
   cust_address    CUST_ADDRESS_TYP,
   nls_territory   VARCHAR2(30),
   cust_email      VARCHAR2(40));

ALTER TABLE list_customers
   EXCHANGE PARTITION rest WITH TABLE exchange_table
   WITHOUT VALIDATION;
```

Modifying Table Partitions: Examples

The following statement marks all the local index partitions corresponding to the `asia` partition of the `list_customers` table `UNUSABLE`:

```
ALTER TABLE list_customers MODIFY PARTITION asia
   UNUSABLE LOCAL INDEXES;
```

The following statement rebuilds all the local index partitions that were marked `UNUSABLE`:

```
ALTER TABLE list_customers MODIFY PARTITION asia
   REBUILD UNUSABLE LOCAL INDEXES;
```

Moving Table Partitions: Example

The following statement moves partition `p2b` (from "Splitting Table Partitions: Examples") to tablespace `omf_ts1`:

```
ALTER TABLE print_media_part
   MOVE PARTITION p2b TABLESPACE omf_ts1;
```

Renaming Table Partitions: Examples

The following statement renames a partition of the `sh.sales` table:

```
ALTER TABLE sales RENAME PARTITION sales_q4_2003 TO sales_currentq;
```

Truncating Table Partitions: Example

The following statement uses the `print_media_demo` table created in "Partitioned Table with LOB Columns Example". It deletes all the data in the `p1` partition and deallocates the freed space:

```
ALTER TABLE print_media_demo
   TRUNCATE PARTITION p1 DROP STORAGE;
```

Updating Global Indexes: Example

The following statement splits partition `sales_q1_2000` of the sample table `sh.sales` and updates any global indexes defined on it:

```
ALTER TABLE sales SPLIT PARTITION sales_q1_2000
   AT (TO_DATE('16-FEB-2000','DD-MON-YYYY'))
   INTO (PARTITION q1a_2000, PARTITION q1b_2000)
   UPDATE GLOBAL INDEXES;
```

Updating Partitioned Indexes: Example

The following statement splits partition costs_Q4_2003 of the sample table sh.costs and updates the local index defined on it. It uses the tablespaces created in "Creating Basic Tablespaces: Examples".

```
CREATE INDEX cost_ix ON costs(channel_id) LOCAL;

ALTER TABLE costs
  SPLIT PARTITION costs_q4_2003 at
    (TO_DATE('01-Nov-2003','dd-mon-yyyy'))
    INTO (PARTITION c_p1, PARTITION c_p2)
  UPDATE INDEXES (cost_ix (PARTITION c_p1 tablespace tbs_02,
                           PARTITION c_p2 tablespace tbs_03));
```

Specifying Object Identifiers: Example

The following statements create an object type, a corresponding object table with a primary-key-based object identifier, and a table having a user-defined REF column:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));

CREATE TABLE emp OF emp_t (
   empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;

CREATE TABLE dept (dno NUMBER, mgr_ref REF emp_t SCOPE is emp);
```

The next statements add a constraint and a user-defined REF column, both of which reference table emp

```
ALTER TABLE dept ADD CONSTRAINT mgr_cons FOREIGN KEY (mgr_ref)
   REFERENCES emp;
ALTER TABLE dept ADD sr_mgr REF emp_t REFERENCES emp;
```

Adding a Table Column: Example

The following statement adds to the countries table a column named duty_pct of data type NUMBER and a column named visa_needed of data type VARCHAR2 with a size of 3 and a CHECK integrity constraint:

```
ALTER TABLE countries
   ADD (duty_pct     NUMBER(2,2)  CHECK (duty_pct < 10.5),
        visa_needed  VARCHAR2(3));
```

Adding a Virtual Table Column: Example

The following statement adds to a copy of the hr.employees table a column named income, which is a combination of salary plus commission. Both salary and commission are NUMBER columns, so the database creates the virtual column as a NUMBER column even though the data type is not specified in the statement:

```
CREATE TABLE emp2 AS SELECT * FROM employees;

ALTER TABLE emp2 ADD (income AS (salary + (salary*commission_pct)));
```

Modifying Table Columns: Examples

The following statement increases the size of the duty_pct column:

```
ALTER TABLE countries
   MODIFY (duty_pct NUMBER(3,2));
```

Because the `MODIFY` clause contains only one column definition, the parentheses around the definition are optional.

The following statement changes the values of the `PCTFREE` and `PCTUSED` parameters for the `employees` table to 30 and 60, respectively:

```
ALTER TABLE employees
   PCTFREE 30
   PCTUSED 60;
```

Adding, Altering, Renaming and Dropping Table Columns: Example

The following statements create a new table to manipulate data and display the information in the newly created table:

```
CREATE TABLE JOBS_Temp AS SELECT * FROM HR.JOBS;

SELECT * FROM JOBS_Temp WHERE MAX_SALARY > 30000;

JOB_ID       JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ----------------------------------- ---------- ----------
AD_PRES    President                           20080      40000
```

The following statement modifies an existing column definition:

```
ALTER TABLE JOBS_Temp MODIFY(JOB_TITLE VARCHAR2(100));
```

The following statement adds two new columns to the table:

```
ALTER TABLE JOBS_Temp ADD (BONUS NUMBER (7,2), COMM NUMBER (5,2), DUMMY NUMBER(2));
```

The following statement displays the newly added columns:

```
SELECT JOB_ID, BONUS, COMM, DUMMY FROM JOBS_Temp WHERE MAX_SALARY > 20000;

JOB_ID        BONUS       COMM      DUMMY
---------- ---------- ---------- ----------
AD_PRES
AD_VP
SA_MAN
```

The following statements rename an existing column and display the modified column:

```
ALTER TABLE JOBS_Temp RENAME COLUMN COMM TO COMMISSION;

SELECT JOB_ID, COMMISSION FROM JOBS_Temp WHERE MAX_SALARY > 20000;

JOB_ID       COMMISSION
---------- ----------
AD_PRES
AD_VP
SA_MAN
```

The following statement drops a single column from the table:

```
ALTER TABLE JOBS_Temp DROP COLUMN DUMMY;
```

The following statement drops multiple columns from the table:

```
ALTER TABLE JOBS_Temp DROP (BONUS, COMMISSION);
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Modifying Table Columns*

Data Encryption: Examples

The following statement encrypts the salary column of the `hr.employees` table using the encryption algorithm `AES256`. As described in "Semantics" above, you must first enable Transparent Data Encryption:

```
ALTER TABLE employees
   MODIFY (salary ENCRYPT USING 'AES256' 'NOMAC');
```

The following statement adds a new encrypted column `online_acct_pw` to the `oe.customers` table, using the default encryption algorithm `AES192`. Specifying `NO SALT` will allow a B-tree index to be created on the column, if desired.

```
ALTER TABLE customers
   ADD (online_acct_pw VARCHAR2(8) ENCRYPT 'NOMAC' NO SALT);
```

The following example decrypts the customer.online_acct_pw column:

```
ALTER TABLE customers
   MODIFY (online_acct_pw DECRYPT);
```

Allocating Extents: Example

The following statement allocates an extent of 5 kilobytes for the `employees` table and makes it available to instance 4:

```
ALTER TABLE employees
  ALLOCATE EXTENT (SIZE 5K INSTANCE 4);
```

Because this statement omits the `DATAFILE` parameter, Oracle Database allocates the extent in one of the data files belonging to the tablespace containing the table.

Specifying a Default Column Value: Examples

This statement modifies the `min_price` column of the `product_information` table so that it has a default value of 10:

```
ALTER TABLE product_information
  MODIFY (min_price DEFAULT 10);
```

If you subsequently add a new row to the `product_information` table and do not specify a value for the `min_price` column, then the value of the `min_price` column is automatically 10:

```
INSERT INTO product_information (product_id, product_name,
   list_price)
   VALUES (300, 'left-handed mouse', 40.50);

SELECT product_id, product_name, list_price, min_price
    FROM product_information
    WHERE product_id = 300;

PRODUCT_ID PRODUCT_NAME         LIST_PRICE  MIN_PRICE
```

```
---------- -------------------- ---------- ----------
       300 left-handed mouse          40.5         10
```

To discontinue previously specified default values, so that they are no longer automatically inserted into newly added rows, replace the values with `NULL`, as shown in this statement:

```
ALTER TABLE product_information
   MODIFY (min_price DEFAULT NULL);
```

The `MODIFY` clause need only specify the column name and the modified part of the definition, rather than the entire column definition. This statement has no effect on any existing values in existing rows.

The following example adds a column defined with `DEFAULT ON NULL` to a table. The `DEFAULT` column value includes the sequence pseudocolumn `NEXTVAL`.

Create sequence `s1` and table `t1` as follows:

```
CREATE SEQUENCE s1 START WITH 1;

CREATE TABLE t1 (name VARCHAR2(10));
INSERT INTO t1 VALUES('Kevin');
INSERT INTO t1 VALUES('Julia');
INSERT INTO t1 VALUES('Ryan');
```

Add column `id`, which defaults to `s1.NEXTVAL`. The default column value for `id` is assigned to each existing row in the table. The order in which `s1.NEXTVAL` is assigned to each row is nondeterministic.

```
ALTER TABLE t1 ADD (id NUMBER DEFAULT ON NULL s1.NEXTVAL NOT NULL);

SELECT id, name FROM t1 ORDER BY id;

        ID NAME
---------- ----------
         1 Kevin
         2 Julia
         3 Ryan
```

If you subsequently add a new row to the table and specify a NULL value for the `id` column, then the `DEFAULT ON NULL` expression `s1.NEXTVAL` is inserted.

```
INSERT INTO t1(id, name) VALUES(NULL, 'Sean');

SELECT id, name FROM t1 ORDER BY id;

        ID NAME
---------- ----------
         1 Kevin
         2 Julia
         3 Ryan
         4 Sean
```

Adding a Constraint to an XMLType Table: Example

The following example adds a primary key constraint to the `xwarehouses` table, created in "XMLType Examples":

```
ALTER TABLE xwarehouses
   ADD (PRIMARY KEY(XMLDATA."WarehouseID"));
```

Refer to XMLDATA Pseudocolumn for information about this pseudocolumn.

Renaming Constraints: Example

The following statement renames the `cust_fname_nn` constraint on the sample table `oe.customers` to `cust_firstname_nn`:

```
ALTER TABLE customers RENAME CONSTRAINT cust_fname_nn
   TO cust_firstname_nn;
```

Dropping Constraints: Examples

The following statement drops the primary key of the `departments` table:

```
ALTER TABLE departments
    DROP PRIMARY KEY CASCADE;
```

If you know that the name of the `PRIMARY KEY` constraint is `pk_dept`, then you could also drop it with the following statement:

```
ALTER TABLE departments
    DROP CONSTRAINT pk_dept CASCADE;
```

The `CASCADE` clause causes Oracle Database to drop any foreign keys that reference the primary key.

The following statement drops the unique key on the `email` column of the `employees` table:

```
ALTER TABLE employees
    DROP UNIQUE (email);
```

The `DROP` clause in this statement omits the `CASCADE` clause. Because of this omission, Oracle Database does not drop the unique key if any foreign key references it.

LOB Columns: Examples

The following statement adds `CLOB` column `resume` to the `employee` table and specifies LOB storage characteristics for the new column:

```
ALTER TABLE employees ADD (resume CLOB)
  LOB (resume) STORE AS resume_seg (TABLESPACE example);
```

To modify the LOB column `resume` to use caching, enter the following statement:

```
ALTER TABLE employees MODIFY LOB (resume) (CACHE);
```

The following statement adds a SecureFiles `CLOB` column `resume` to the `employee` table and specifies LOB storage characteristics for the new column. SecureFiles LOBs must be stored in tablespaces with automatic segment-space management. Therefore, the LOB data in this example is stored in the `auto_seg_ts` tablespace, which was created in "Specifying Segment Space Management for a Tablespace: Example":

```
ALTER TABLE employees ADD (resume CLOB)
LOB (resume) STORE AS SECUREFILE resume_seg (TABLESPACE auto_seg_ts);
```

To modify the LOB column `resume` so that it does not use caching, enter the following statement:

```
ALTER TABLE employees MODIFY LOB (resume) (NOCACHE);
```

Nested Tables: Examples

The following statement adds the nested table column `skills` to the `employee` table:

```
ALTER TABLE employees ADD (skills skill_table_type)
    NESTED TABLE skills STORE AS nested_skill_table;
```

You can also modify nested table storage characteristics. Use the name of the storage table specified in the *`nested_table_col_properties`* to make the modification. You cannot query or perform DML statements on the storage table. Use the storage table only to modify the nested table column storage characteristics.

The following statement creates table `vet_service` with nested table column `client` and storage table `client_tab`. Nested table `client_tab` is modified to specify constraints:

```
CREATE TYPE pet_t AS OBJECT
   (pet_id NUMBER, pet_name VARCHAR2(10), pet_dob DATE);
/

CREATE TYPE pet AS TABLE OF pet_t;
/

CREATE TABLE vet_service (vet_name VARCHAR2(30),
                          client   pet)
  NESTED TABLE client STORE AS client_tab;

ALTER TABLE client_tab ADD UNIQUE (pet_id);
```

The following statement alters the storage table for a nested table of `REF` values to specify that the `REF` is scoped:

```
CREATE TYPE emp_t AS OBJECT (eno number, ename char(31));
CREATE TYPE emps_t AS TABLE OF REF emp_t;
CREATE TABLE emptab OF emp_t;
CREATE TABLE dept (dno NUMBER, employees emps_t)
   NESTED TABLE employees STORE AS deptemps;
ALTER TABLE deptemps ADD (SCOPE FOR (COLUMN_VALUE) IS emptab);
```

Similarly, to specify storing the `REF` with rowid:

```
ALTER TABLE deptemps ADD (REF(column_value) WITH ROWID);
```

In order to execute these `ALTER TABLE` statements successfully, the storage table `deptemps` must be empty. Also, because the nested table is defined as a table of scalar values (`REF` values), Oracle Database implicitly provides the column name `COLUMN_VALUE` for the storage table.

> **See Also:**
>
> - CREATE TABLE for more information about nested table storage
> - *Oracle Database Object-Relational Developer's Guide* for more information about nested tables

REF Columns: Examples

The following statement creates an object type `dept_t` and then creates table `staff`:

```
CREATE TYPE dept_t AS OBJECT
   (deptno NUMBER, dname VARCHAR2(20));
/

CREATE TABLE staff
   (name   VARCHAR2(100),
    salary NUMBER,
    dept   REF dept_t);
```

An object table `offices` is created as:

```
CREATE TABLE offices OF dept_t;
```

The `dept` column can store references to objects of `dept_t` stored in any table. If you would like to restrict the references to point only to objects stored in the `departments` table, then you could do so by adding a scope constraint on the `dept` column as follows:

```
ALTER TABLE staff
    ADD (SCOPE FOR (dept) IS offices);
```

The preceding `ALTER TABLE` statement will succeed only if the `staff` table is empty.

If you want the `REF` values in the `dept` column of `staff` to also store the rowids, then issue the following statement:

```
ALTER TABLE staff
   ADD (REF(dept) WITH ROWID);
```

Unpacked Storage in ANYDATA Columns: Example

This example creates a table with an `ANYDATA` column, stores opaque data types in the `ANYDATA` column using unpacked storage, and then queries the data types. This example assumes that you are connected to the database as user `hr`.

Create table `t1`, which contains a `NUMBER` column `n` and an `ANYDATA` column `x`:

```
CREATE TABLE t1 (n NUMBER, x ANYDATA);
```

Create an object type `clob_typ`, which contains a `CLOB` attribute:

```
CREATE OR REPLACE TYPE clob_typ AS OBJECT (c clob);
/
```

Enable unpacked storage of the opaque data types `XMLType` and `clob_typ` in `ANYDATA` column `x` of table `t1`:

```
ALTER TABLE t1 MODIFY OPAQUE TYPE x STORE (XMLType, clob_typ) UNPACKED;
```

Insert `XMLType` and `clob_typ` objects into table `t1`. These types will use unpacked storage:

```
INSERT INTO t1
  VALUES(1, anydata.convertobject(XMLType('<Test>This is test XML</Test>')));

INSERT INTO t1
  VALUES(2, anydata.convertobject(clob_typ(TO_CLOB('This is a test CLOB'))));
```

Query table `t1` to view the names of the types stored in `ANYDATA` column `x`:

```
SELECT t1.*, anydata.getTypeName(t1.x) typename FROM t1;

    N X()                  TYPENAME
----- -------------------- --------------------
    1 ANYDATA()            SYS.XMLTYPE
    2 ANYDATA()            HR.CLOB_TYP
```

Create functions that allow you to query the values stored in the `XMLType` and `clob_typ` data types:

```
CREATE FUNCTION get_xmltype (ad IN ANYDATA) RETURN VARCHAR2 AS
      rtn_val PLS_INTEGER;
      my_xmltype XMLType;
      string_val VARCHAR2(30);
   BEGIN
      rtn_val := ad.getObject(my_xmltype);
      string_val := my_xmltype.getstringval();
      return (string_val);
   END;
/

CREATE FUNCTION get_clob_typ (ad IN ANYDATA) RETURN VARCHAR2 AS
      rtn_val PLS_INTEGER;
      my_clob_typ clob_typ;
      string_val VARCHAR2(30);
   BEGIN
      rtn_val := ad.getObject(my_clob_typ);
      string_val := (my_clob_typ.c);
      return (string_val);
   END;
/
```

Query table `t1` to view the values stored in each data type in `ANYDATA` column `x`:

```
SELECT t1.*, anydata.getTypeName(t1.x) typename,
  CASE
    WHEN anydata.gettypename(t1.x) = 'SYS.XMLTYPE' THEN get_xmltype(t1.x)
    WHEN anydata.gettypename(t1.x) = 'HR.CLOB_TYP' THEN get_clob_typ(t1.x)
  END string_value
FROM t1;

    N X()                  TYPENAME             STRING_VALUE
----- -------------------- -------------------- ------------------------------
    1 ANYDATA()            SYS.XMLTYPE          <Test>This is test XML</Test>
    2 ANYDATA()            HR.CLOB_TYP          This is a test CLOB
```

Additional Examples

For examples of defining integrity constraints with the `ALTER TABLE` statement, see the *constraint*.

For examples of changing the storage parameters of a table, see the storage_clause.

ALTER TABLESPACE

Purpose

Use the `ALTER TABLESPACE` statement to alter an existing tablespace or one or more of its data files or temp files.

You cannot use this statement to convert a dictionary-managed tablespace to a locally managed tablespace. For that purpose, use the `DBMS_SPACE_ADMIN` package, which is documented in *Oracle Database PL/SQL Packages and Types Reference*.

> **See Also:**
>
> *Oracle Database Administrator's Guide* and CREATE TABLESPACE for information on creating a tablespace

Prerequisites

To alter the `SYSAUX` tablespace, you must have the `SYSDBA` system privilege.

If you have the `ALTER TABLESPACE` system privilege, then you can perform any `ALTER TABLESPACE` operation. If you have the `MANAGE TABLESPACE` system privilege, then you can only perform the following operations:

- Take a tablespace online or offline
- Begin or end a backup
- Make a tablespace read only or read write
- Change the state of a tablespace to `PERMANENT` or `TEMPORARY`
- Set the default logging mode of a tablespace to `LOGGING` or `NOLOGGING`
- Put a tablespace in force logging mode or take it out of force logging mode
- Rename a tablespace or a tablespace data file
- Specify `RETENTION GUARANTEE` or `RETENTION NOGUARANTEE` for an undo tablespace
- Resize a data file for a tablespace
- Enable or disable autoextension of a data file for a tablespace
- Shrink the amount of space a temporary tablespace or a temp file is taking

Before you can make a tablespace read only, the following conditions must be met:

- The tablespace must be online.
- The tablespace must not contain any active rollback segments. For this reason, the `SYSTEM` tablespace can never be made read only, because it contains the `SYSTEM` rollback segment. Additionally, because the rollback segments of a read-only tablespace are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace read only.
- The tablespace must not be involved in an open backup, because the end of a backup updates the header file of all data files in the tablespace.

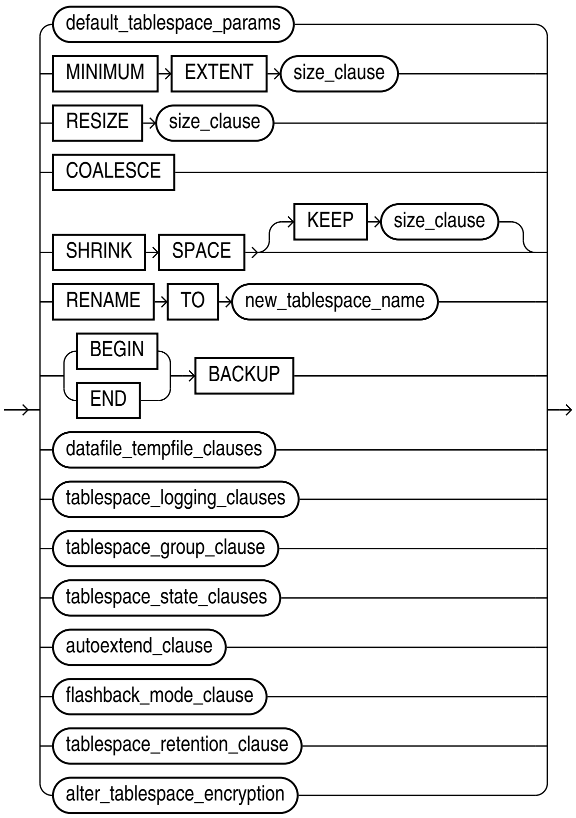
Performing this function in restricted mode may help you meet these restrictions, because only users with `RESTRICTED SESSION` system privilege can be logged on.

Syntax

***alter_tablespace*::=**

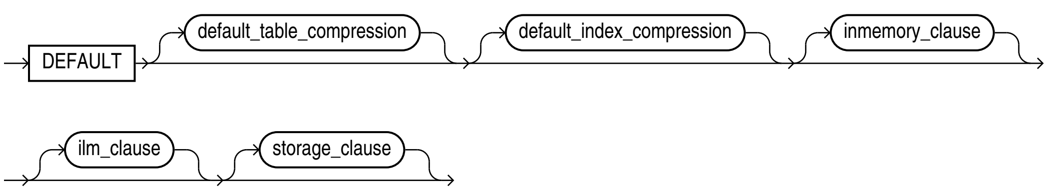
(*alter_tablespace_attrs*::=)

***alter_tablespace_attrs*::=**



(*default_tablespace_params*::=, *size_clause*::=, *datafile_tempfile_clauses*::=, *tablespace_logging_clauses*::=, *tablespace_group_clause*::=, *tablespace_state_clauses*::=, *autoextend_clause*::=, *flashback_mode_clause*::=, *tablespace_retention_clause*::=, *alter_tablespace_encryption*::=)

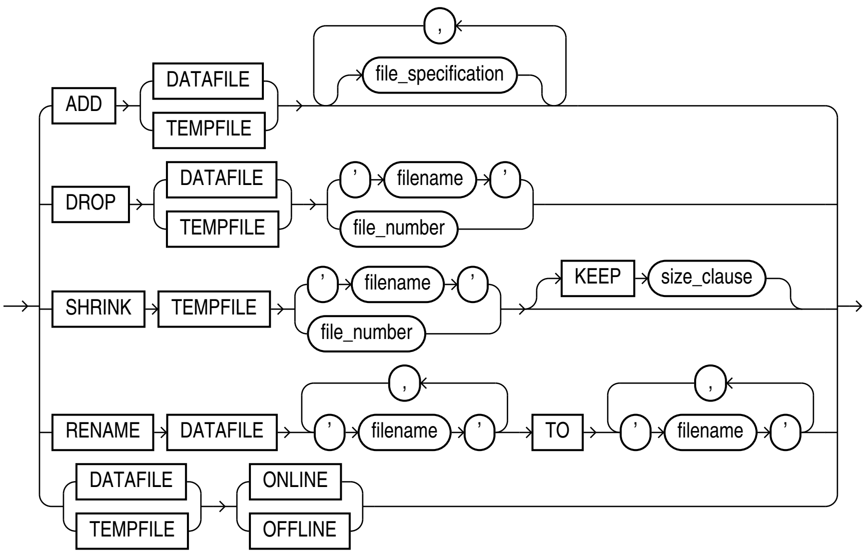
***default_tablespace_params*::=**



(*default_table_compression*::=—part of `CREATE TABLESPACE`, *default_index_compression*::=—part of `CREATE TABLESPACE`, *inmemory_clause*::=—part of `CREATE TABLESPACE`, *ilm_clause*::=—part of `ALTER TABLE`, *storage_clause*::=)

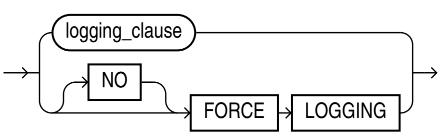
> **Note:**
>
> If you specify the `DEFAULT` clause, then you must specify at least one of the clauses *`default_table_compression`*, *`default_index_compression`*, *`inmemory_clause`*, *`ilm_clause`*, or *`storage_clause`*.

***datafile_tempfile_clauses*::=**



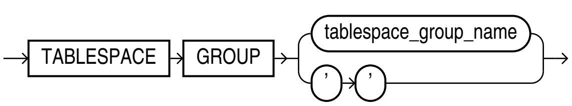
(*file_specification*::=).

***tablespace_logging_clauses*::=**

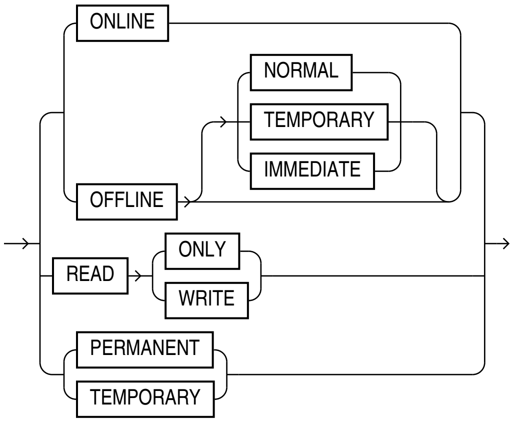


(*logging_clause*::=)

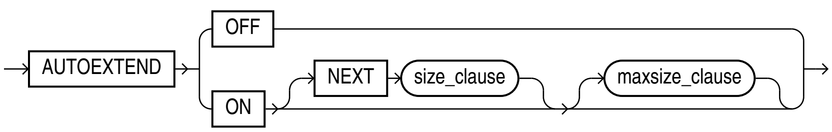
***tablespace_group_clause*::=**



***tablespace_state_clauses*::=**

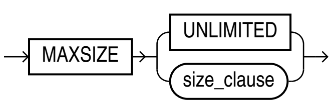


***autoextend_clause*::=**



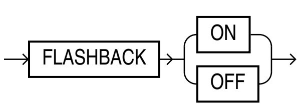
(*size_clause*::=)

***maxsize_clause*::=**

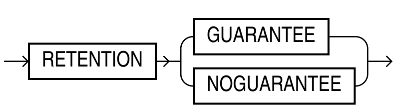


(*size_clause*::=)

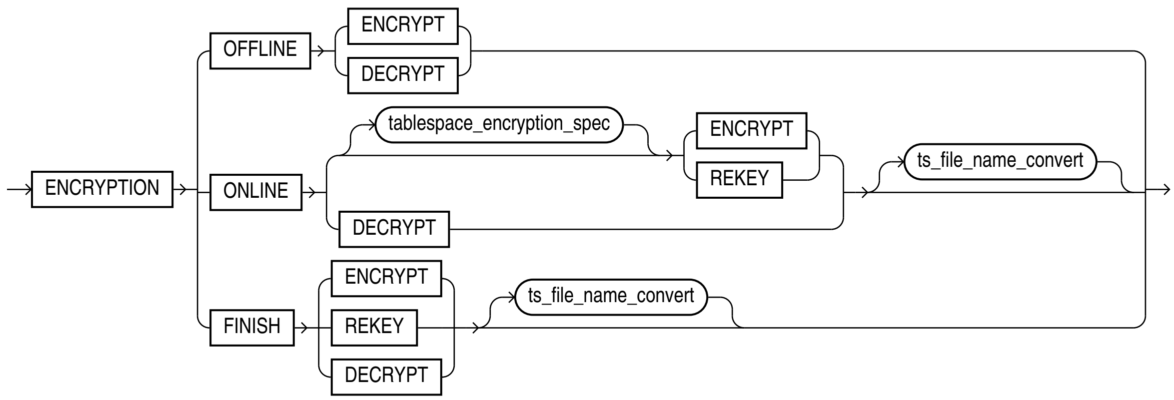
***flashback_mode_clause*::=**



***tablespace_retention_clause*::=**

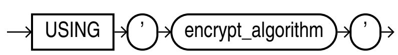


***alter_tablespace_encryption*::=**

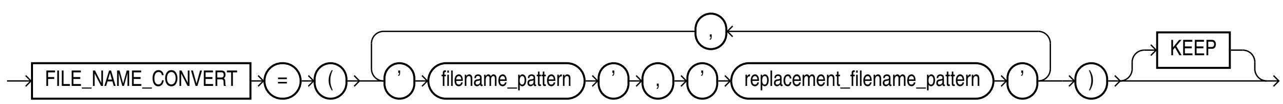


(*tablespace_encryption_spec*::=, *ts_file_name_convert*::=)

***tablespace_encryption_spec*::=**



***ts_file_name_convert*::=**



Semantics

tablespace

Specify the name of the tablespace to be altered.

Restrictions on Altering Tablespaces

Altering tablespaces is subject to the following restrictions:

- If *`tablespace`* is an undo tablespace, then the only other clauses you can specify in this statement are `ADD DATAFILE`, `RENAME DATAFILE`, `RENAME TO` (renaming the tablespace), `DATAFILE` ... `ONLINE`, `DATAFILE` ... `OFFLINE`, `BEGIN BACKUP`, and `END BACKUP`.
- You cannot make the `SYSTEM` tablespace read only or temporary and you cannot take it offline.
- For locally managed temporary tablespaces, the only clause you can specify in this statement is the `ADD` clause.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces

alter_tablespace_attrs

Use the `alter_tablespace_attrs` clauses to change the attributes of the tablespace.

default_tablespace_params

This clause lets you specify new default parameters for the tablespace. The new default parameters apply to objects subsequently created in the tablespace.

The clauses `default_table_compression`, `default_index_compression`, `inmemory_clause`, `ilm_clause`, and `storage_clause` have the same semantics in `CREATE TABLESPACE` and `ALTER TABLESPACE`. For complete information on these clauses, refer to the *default_tablespace_params* clause in the documentation on `CREATE TABLESPACE`.

MINIMUM EXTENT

This clause is valid only for permanent dictionary-managed tablespaces. The `MINIMUM EXTENT` clause lets you control free space fragmentation in the tablespace by ensuring that every used or free extent in a tablespace is at least as large as, and is a multiple of, the value specified in the `size_clause`.

Restriction on MINIMUM EXTENT

You cannot specify this clause for a locally managed tablespace or for a dictionary-managed temporary tablespace.

> **See Also:**
>
> *size_clause* for information about that clause, *Oracle Database Administrator's Guide* for more information about using `MINIMUM EXTENT` to control space fragmentation

RESIZE Clause

This clause is valid only for bigfile tablespaces. It lets you increase or decrease the size of the single data file to an absolute size. Use `K`, `M`, `G`, or `T` to specify the size in kilobytes, megabytes, gigabytes, or terabytes, respectively.

To change the size of a newly added data file or temp file in smallfile tablespaces, use the `ALTER DATABASE` ... `autoextend_clause` (see *database_file_clauses*).

> **See Also:**
>
> BIGFILE | SMALLFILE for information on bigfile tablespaces

COALESCE

For each data file in the tablespace, this clause combines all contiguous free extents into larger contiguous extents.

SHRINK SPACE Clause

This clause is valid only for temporary tablespaces. It lets you reduce the amount of space the tablespace is taking. In the optional `KEEP` clause, the *`size_clause`* defines the lower bound that a tablespace can be shrunk to. It is the opposite of `MAXSIZE` for an autoextensible tablespace. If you omit the `KEEP` clause, then the database will attempt to shrink the tablespace as much as possible as long as other tablespace storage attributes are satisfied.

RENAME Clause

Use this clause to rename *`tablespace`*. This clause is valid only if *`tablespace`* and all its data files are online and the `COMPATIBLE` parameter is set to 10.0.0 or greater. You can rename both permanent and temporary tablespaces.

If *`tablespace`* is read only, then Oracle Database does not update the data file headers to reflect the new name. The alert log will indicate that the data file headers have not been updated.

> **Note:**
>
> If you re-create the control file, and if the data files that Oracle Database uses for this purpose are restored backups whose headers reflect the old tablespace name, then the re-created control file will also reflect the old tablespace name. However, after the database is fully recovered, the control file will reflect the new name.

If *`tablespace`* has been designated as the undo tablespace for any instance in an Oracle Real Application Clusters (Oracle RAC) environment, and if a server parameter file was used to start up the database, then Oracle Database changes the value of the `UNDO_TABLESPACE` parameter for that instance in the server parameter file (`SPFILE`) to reflect the new tablespace name. If a single-instance database is using a parameter file (pfile) instead of an spfile, then the database puts a message in the alert log advising the database administrator to change the value manually in the pfile.

> **Note:**
>
> The `RENAME` clause does not change the value of the `UNDO_TABLESPACE` parameter in the running instance. Although this does not affect the functioning of the undo tablespace, Oracle recommends that you issue the following statement to manually change the value of `UNDO_TABLESPACE` to the new tablespace name for the duration of the instance:
>
> ```
> ALTER SYSTEM SET UNDO_TABLESPACE = new_tablespace_name SCOPE = MEMORY;
> ```
>
> You only need to issue this statement once. If the `UNDO_TABLESPACE` parameter is set to the new tablespace name in the pfile or spfile, then the parameter will be set correctly when the instance is next restarted.

Restriction on Renaming Tablespaces

You cannot rename the `SYSTEM` or `SYSAUX` tablespaces.

BACKUP Clauses

Use these clauses to move all data files in a tablespace into or out of online (sometimes called hot) backup mode.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on restarting the database without media recovery
> - `ALTER DATABASE` "BACKUP Clauses" for information on moving all data files in the database into and out of online backup mode
> - `ALTER DATABASE` *alter_datafile_clause* for information on taking individual data files out of online backup mode

BEGIN BACKUP

Specify `BEGIN BACKUP` to indicate that an open backup is to be performed on the data files that make up this tablespace. This clause does not prevent users from accessing the tablespace. You must use this clause before beginning an open backup.

Restrictions on Beginning Tablespace Backup

Beginning tablespace backup is subject to the following restrictions:

- You cannot specify this clause for a read-only tablespace or for a temporary locally managed tablespace.
- While the backup is in progress, you cannot take the tablespace offline normally, shut down the instance, or begin another backup of the tablespace.

> **See Also:**
>
> "Backing Up Tablespaces: Examples"

END BACKUP

Specify `END BACKUP` to indicate that an online backup of the tablespace is complete. Use this clause as soon as possible after completing an online backup. Otherwise, if an instance failure or `SHUTDOWN ABORT` occurs, then Oracle Database assumes that media recovery (possibly requiring archived redo log) is necessary at the next instance startup.

Restriction on Ending Tablespace Backup

You cannot use this clause on a read-only tablespace.

datafile_tempfile_clauses

The tablespace file clauses let you add or modify a data file or temp file.

ADD Clause

Specify `ADD` to add to the tablespace a data file or temp file specified by *`file_specification`*. Use the *`datafile_tempfile_spec`* form of *`file_specification`* (see *file_specification*) to list regular data files and temp files in an operating system file system or to list Oracle Automatic Storage Management disk group files.

For locally managed temporary tablespaces, this is the only clause you can specify at any time.

If you omit *`file_specification`*, then Oracle Database creates an Oracle Managed File of 100M with `AUTOEXTEND` enabled.

You can add a data file or temp file to a locally managed tablespace that is online or to a dictionary managed tablespace that is online or offline. Ensure the file is not in use by another database.

Restriction on Adding Data Files and Temp Files

You cannot specify this clause for a bigfile (single-file) tablespace, as such a tablespace has only one data file or temp file.

> **Note:**
>
> On some operating systems, Oracle does not allocate space for a temp file until the temp file blocks are actually accessed. This delay in space allocation results in faster creation and resizing of temp files, but it requires that sufficient disk space is available when the temp files are later used. To avoid potential problems, before you create or resize a temp file, ensure that the available disk space exceeds the size of the new temp file or the increased size of a resized temp file. The excess space should allow for anticipated increases in disk space use by unrelated operations as well. Then proceed with the creation or resizing operation.

> **See Also:**
>
> *file_specification* , "Adding and Dropping Data Files and Temp Files: Examples", and "Adding an Oracle-managed Data File: Example"

DROP Clause

Specify `DROP` to drop from the tablespace an empty data file or temp file specified by *`filename`* or *`file_number`*. This clause causes the data file or temp file to be removed from the data dictionary and deleted from the operating system. The database must be open at the time this clause is specified.

The `ALTER TABLESPACE` ... `DROP TEMPFILE` statement is equivalent to specifying the `ALTER DATABASE TEMPFILE` ... `DROP INCLUDING DATAFILES`.

Restrictions on Dropping Files

To drop a data file or temp file, the data file or temp file:

- Must be empty.
- Cannot be the first file that was created in the tablespace. In such cases, drop the tablespace instead.
- Cannot be in a read-only tablespace that was migrated from dictionary managed to locally managed. Dropping a data file from all other read-only tablespaces is supported.
- Cannot be offline.

> **See Also:**
>
> - `ALTER DATABASE` *alter_tempfile_clause* for additional information on dropping temp files
> - *Oracle Database Administrator's Guide* for information on data file numbers and for guidelines on managing data files
> - "Adding and Dropping Data Files and Temp Files: Examples"

SHRINK TEMPFILE Clause

This clause is valid only when altering a temporary tablespace. It lets you reduce the amount of space the specified temp file is taking. In the optional `KEEP` clause, the *`size_clause`* defines the lower bound that the temp file can be shrunk to. It is the opposite of `MAXSIZE` for an autoextensible tablespace. If you omit the `KEEP` clause, then the database will attempt to shrink the temp file as much as possible as long as other storage attributes are satisfied.

RENAME DATAFILE Clause

Specify `RENAME DATAFILE` to rename one or more of the tablespace data files. The database must be open, and you must take the tablespace offline before renaming it. Each *`filename`* must fully specify a data file using the conventions for filenames on your operating system.

This clause merely associates the tablespace with the new file rather than the old one. This clause does not actually change the name of the operating system file. You must change the name of the file through your operating system.

See Also:

"Moving and Renaming Tablespaces: Example"

ONLINE | OFFLINE Clauses

Use these clauses to take all data files or temp files in the tablespace offline or put them online. These clauses have no effect on the ONLINE or OFFLINE status of the tablespace itself.

The database must be mounted. If *tablespace* is SYSTEM, or an undo tablespace, or the default temporary tablespace, then the database must not be open.

tablespace_logging_clauses

Use these clauses to set or change the logging characteristics of the tablespace.

logging_clause

Specify LOGGING if you want logging of all tables, indexes, and partitions within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

When an existing tablespace logging attribute is changed by an ALTER TABLESPACE statement, all tables, indexes, and partitions created *after* the statement will have the new default logging attribute (which you can still subsequently override). The logging attribute of existing objects is not changed.

If the tablespace is in FORCE LOGGING mode, then you can specify NOLOGGING in this statement to set the default logging mode of the tablespace to NOLOGGING, but this will not take the tablespace out of FORCE LOGGING mode.

[NO] FORCE LOGGING

Use this clause to put the tablespace in force logging mode or take it out of force logging mode. The database must be open and in READ WRITE mode. Neither of these settings changes the default LOGGING or NOLOGGING mode of the tablespace.

Restriction on Force Logging Mode

You cannot specify FORCE LOGGING for an undo or a temporary tablespace.

See Also:

Oracle Database Administrator's Guide for information on when to use FORCE LOGGING mode and "Changing Tablespace Logging Attributes: Example"

tablespace_group_clause

This clause is valid only for locally managed temporary tablespaces. Use this clause to add `tablespace` to or remove it from the `tablespace_group_name` tablespace group.

- Specify a group name to indicate that `tablespace` is a member of this tablespace group. If `tablespace_group_name` does not already exist, then Oracle Database implicitly creates it when you alter tablespace to be a member of it.
- Specify an empty string (' ') to remove `tablespace` from the `tablespace_group_name` tablespace group.

Restriction on Tablespace Groups

You cannot specify a tablespace group for a permanent tablespace or for a dictionary-managed temporary tablespace.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on tablespace groups and "Assigning a Tablespace Group: Example"

tablespace_state_clauses

Use these clauses to set or change the state of the tablespace.

ONLINE | OFFLINE

Specify `ONLINE` to bring the tablespace online. Specify `OFFLINE` to take the tablespace offline and prevent further access to its segments. When you take a tablespace offline, all of its data files are also offline.

> **Note:**
>
> Before taking a tablespace offline for a long time, consider changing the tablespace allocation of any users who have been assigned the tablespace as either a default or temporary tablespace. While the tablespace is offline, such users cannot allocate space for objects or sort areas in the tablespace. See ALTER USER for more information on allocating tablespace quota to users.

Restriction on Taking Tablespaces Offline

You cannot take a temporary tablespace offline.

OFFLINE NORMAL

Specify `NORMAL` to flush all blocks in all data files in the tablespace out of the system global area (SGA). You need not perform media recovery on this tablespace before bringing it back online. This is the default.

OFFLINE TEMPORARY

If you specify TEMPORARY, then Oracle Database performs a checkpoint for all online data files in the tablespace but does not ensure that all files can be written. Files that are offline when you issue this statement may require media recovery before you bring the tablespace back online.

OFFLINE IMMEDIATE

If you specify IMMEDIATE, then Oracle Database does not ensure that tablespace files are available and does not perform a checkpoint. You must perform media recovery on the tablespace before bringing it back online.

> **Note:**
>
> The FOR RECOVER setting for ALTER TABLESPACE ... OFFLINE has been deprecated. The syntax is supported for backward compatibility. However, Oracle recommends that you use the transportable tablespaces feature for tablespace recovery.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for information on using transportable tablespaces to perform media recovery

READ ONLY | READ WRITE

Specify READ ONLY to place the tablespace in **transition read-only mode**. In this state, existing transactions can complete (commit or roll back), but no further DML operations are allowed to the tablespace except for rollback of existing transactions that previously modified blocks in the tablespace. You cannot make the SYSAUX, SYSTEM, or temporary tablespaces READ ONLY.

When a tablespace is read only, you can copy its files to read-only media. You must then rename the data files in the control file to point to the new location by using the SQL statement ALTER DATABASE ... RENAME.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on read-only tablespaces
> - ALTER DATABASE

Specify READ WRITE to indicate that write operations are allowed on a previously read-only tablespace.

PERMANENT | TEMPORARY

Specify PERMANENT to indicate that the tablespace is to be converted from a temporary to a permanent tablespace. A permanent tablespace is one in which permanent database objects can be stored. This is the default when a tablespace is created.

Specify TEMPORARY to indicate that the tablespace is to be converted from a permanent to a temporary tablespace. A temporary tablespace is one in which no permanent database objects can be stored. Objects in a temporary tablespace persist only for the duration of the session.

Restrictions on Temporary Tablespaces

Temporary tablespaces are subject to the following restrictions:

- You cannot specify TEMPORARY for the SYSAUX tablespace.
- If *tablespace* was not created with a standard block size, then you cannot change it from permanent to temporary.
- You cannot specify TEMPORARY for a tablespace in FORCE LOGGING mode.

autoextend_clause

This clause is valid only for bigfile (single-file) tablespaces. Use this clause to enable or disable autoextension of the single data file in the tablespace. To enable or disable autoextension of a newly added data file or temp file in smallfile tablespaces, use the *autoextend_clause* of the *database_file_clauses* in the ALTER DATABASE statement.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information about bigfile (single-file) tablespaces
> - *file_specification* for more information about the *autoextend_clause*

flashback_mode_clause

Use this clause to specify whether this tablespace should participate in any subsequent FLASHBACK DATABASE operation.

- For you to turn FLASHBACK mode on, the database must be mounted and closed.
- For you to turn FLASHBACK mode off, the database must be mounted, either open READ WRITE or closed.

This clause is not valid for temporary tablespaces.

Refer to CREATE TABLESPACE for more complete information on this clause.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information about Flashback Database

tablespace_retention_clause

This clause has the same semantics in CREATE TABLESPACE and ALTER TABLESPACE statements. Refer to tablespace_retention_clause in the documentation on CREATE TABLESPACE.

alter_tablespace_encryption

These clauses let you encrypt, decrypt, or rekey the tablespace.

OFFLINE

This clause lets you encrypt or decrypt the tablespace with offline conversion. The tablespace must be offline or the database must be mounted, but not open. The offline conversion method does not use auxiliary disk space or files; it operates directly on the existing datafiles. Therefore, you should perform a full backup of the tablespace before converting it offline.

- Specify ENCRYPT to encrypt the tablespace. The tablespace must be unencrpyted. The datafiles of the tablespace will be encrypted using the AES128 algorithm.
- Specify DECRYPT to decrypt the tablespace. The tablespace must have been previously encrypted with offline conversion (OFFLINE ENCRYPT).

If an offline conversion operation is interrupted, then you can reissue the offline conversion command to finish the operation.

ONLINE

This clause lets you encrypt, decrypt, or rekey the tablespace with online conversion. The tablespace must be online. The online conversion method creates a new datafile for each datafile in the tablespace. Therefore, before using this clause, ensure that the amount of free disk space is greater than or equal to the amount of disk space currently used by the tablespace.

- Specify ENCRYPT to encrypt the tablespace. The tablespace must be unencrpyted.
- Specify REKEY to encrypt an encrypted the tablespace using a different encryption algorithm. The tablespace must have been encrypted when it was created or encrypted with online conversion (ONLINE ENCRYPT).
- Specify DECRYPT to decrypt the tablespace. The tablespace must have been encrypted when it was created or encrypted with online conversion (ONLINE ENCRYPT).

If an online conversion operation is interrupted, then you can issue the FINISH clause to finish the operation. Refer to the FINISH clause.

tablespace_encryption_spec

Use this clause to specify the encryption algorithm to use when encrypting or rekeying the tablespace. If you omit this clause, then the datafiles will be encrypted using the AES128 algorithm. Refer to *tablespace_encryption_spec* in the documentation on CREATE TABLESPACE for the full semantics of this clause.

ts_file_name_convert

Use this clause to determine how the database generates the names of the new datafiles that are created during online conversion.

- For *filename_pattern*, specify a string found in an existing datafile name.
- For *replacement_filename_pattern*, specify a replacement string. Oracle Database will replace *filename_pattern* with *replacement_filename_pattern* when naming the new datafile.
- Specify KEEP to retain the original files after the tablespace conversion is finished. If you omit this clause, then the original files are deleted when the conversion is finished.

FINISH

This clause lets you finish a previously interrupted online conversion operation. The ENCRYPT, DECRYPT, REKEY, and *ts_file_name_convert* clauses have the same semantics here as they have for the ONLINE clause. Refer to the ONLINE clause for complete information.

Restriction on the *alter_tablespace_encryption* Clause

You cannot perform offline or online conversions on temporary tablespaces.

Examples

Backing Up Tablespaces: Examples

The following statement signals to the database that a backup is about to begin:

```
ALTER TABLESPACE tbs_01
    BEGIN BACKUP;
```

The following statement signals to the database that the backup is finished:

```
ALTER TABLESPACE tbs_01
   END BACKUP;
```

Moving and Renaming Tablespaces: Example

This example moves and renames a data file associated with the tbs_02 tablespace, created in "Enabling Autoextend for a Tablespace: Example", from diskb:tbs_f5.dbf to diska:tbs_f5.dbf:

1. Take the tablespace offline using an ALTER TABLESPACE statement with the OFFLINE clause:

   ```
   ALTER TABLESPACE tbs_02 OFFLINE NORMAL;
   ```

2. Copy the file from diskb:tbs_f5.dbf to diska:tbs_f5.dbf using your operating system commands.
3. Rename the data file using an ALTER TABLESPACE statement with the RENAME DATAFILE clause:

   ```
   ALTER TABLESPACE tbs_02
     RENAME DATAFILE 'diskb:tbs_f5.dbf'
     TO              'diska:tbs_f5.dbf';
   ```

4. Bring the tablespace back online using an ALTER TABLESPACE statement with the ONLINE clause:

   ```
   ALTER TABLESPACE tbs_02 ONLINE;
   ```

Adding and Dropping Data Files and Temp Files: Examples

The following statement adds a data file to the tablespace. When more space is needed, new 10-kilobytes extents will be added up to a maximum of 100 kilobytes:

```
ALTER TABLESPACE tbs_03
    ADD DATAFILE 'tbs_f04.dbf'
    SIZE 100K
    AUTOEXTEND ON
    NEXT 10K
    MAXSIZE 100K;
```

The following statement drops the empty data file:

```
ALTER TABLESPACE tbs_03
    DROP DATAFILE 'tbs_f04.dbf';
```

The following statements add a temp file to the temporary tablespace created in "Creating a Temporary Tablespace: Example" and then drops the temp file:

```
ALTER TABLESPACE temp_demo ADD TEMPFILE 'temp05.dbf' SIZE 5 AUTOEXTEND ON;

ALTER TABLESPACE temp_demo DROP TEMPFILE 'temp05.dbf';
```

Managing Space in a Temporary Tablespace: Example

The following statement manages the space in the temporary tablespace created in "Creating a Temporary Tablespace: Example" using the `SHRINK SPACE` clause. The `KEEP` clause is omitted, so the database will attempt to shrink the tablespace as much as possible as long as other tablespace storage attributes are satisfied.

```
ALTER TABLESPACE temp_demo SHRINK SPACE;
```

Adding an Oracle-managed Data File: Example

The following example adds an Oracle-managed data file to the `omf_ts1` tablespace (see "Creating Oracle Managed Files: Examples" for the creation of this tablespace). The new data file is 100M and is autoextensible with unlimited maximum size:

```
ALTER TABLESPACE omf_ts1 ADD DATAFILE;
```

Changing Tablespace Logging Attributes: Example

The following example changes the default logging attribute of a tablespace to `NOLOGGING`:

```
ALTER TABLESPACE tbs_03 NOLOGGING;
```

Altering a tablespace logging attribute has no affect on the logging attributes of the existing schema objects within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Changing Undo Data Retention: Examples

The following statement changes the undo data retention for tablespace `undots1` to normal undo data behavior:

```
ALTER TABLESPACE undots1
  RETENTION NOGUARANTEE;
```

The following statement changes the undo data retention for tablespace `undots1` to behavior that preserves unexpired undo data:

```
ALTER TABLESPACE undots1
  RETENTION GUARANTEE;
```

ALTER TABLESPACE SET

> **Note:**
>
> This SQL statement is valid only if you are using Oracle Sharding. For more information on Oracle Sharding, refer to *Oracle Database Administrator's Guide*.

Purpose

Use the ALTER TABLESPACE SET statement to change an attribute of an existing tablespace set. The attribute change is applied to all tablespaces in the tablespace set.

> **See Also:**
>
> CREATE TABLESPACE SET and DROP TABLESPACE SET

Prerequisites

You must be connected to a shard catalog database as an SDB user.

If you have the ALTER TABLESPACE system privilege, then you can perform any ALTER TABLESPACE SET operation. If you have the MANAGE TABLESPACE system privilege, then you can only perform the following operations:

- Take all tablespaces in a tablespace set online or offline
- Begin or end a backup
- Make all tablespaces in a tablespace set read only or read write
- Set the default logging mode of all tablespaces in a tablespace set to LOGGING or NOLOGGING
- Put all tablespaces in a tablespace set in force logging mode or take them out of force logging mode
- Resize all data files for a tablespace set
- Enable or disable autoextension of all data files for a tablespace set

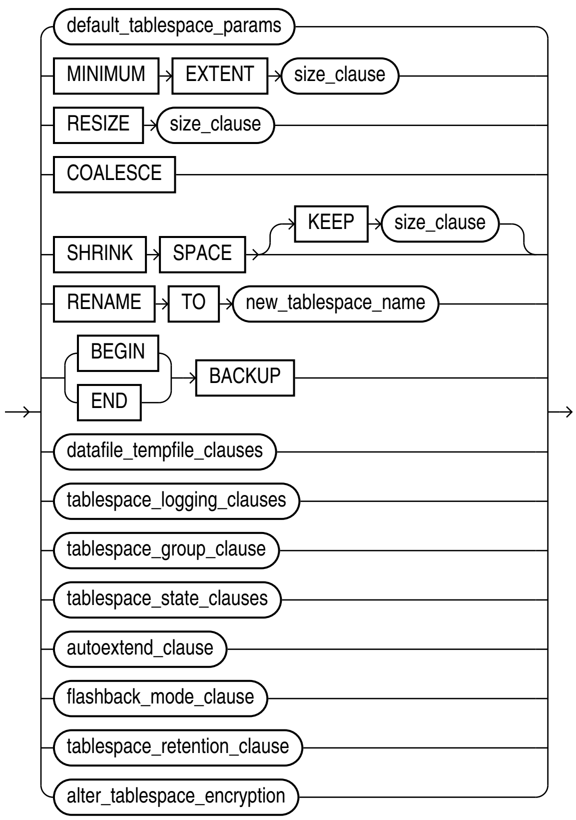
Before you can make a tablespace set read only, the following conditions must be met:

- The tablespaces in the tablespace set must be online.
- The tablespace set must not contain any active rollback segments. Additionally, because the rollback segments of a read-only tablespace set are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace set read only.
- The tablespace set must not be involved in an open backup, because the end of a backup updates the header file of all data files in the tablespace set.

Syntax

***alter_tablespace_set*::=**

***alter_tablespace_attrs*::=**



(See the following clauses of `ALTER TABLESPACE`: *default_tablespace_params*::=, *size_clause*::=, *datafile_tempfile_clauses*::=, *tablespace_logging_clauses*::=, *tablespace_state_clauses*::=, *autoextend_clause*::=, *alter_tablespace_encryption*::=)

Semantics

tablespace_set

Specify the name of the tablespace set to be altered.

alter_tablespace_attrs

Use this clause to change an attribute for all tablespaces in the tablespace set.

The subclauses of *`alter_tablespace_attrs`* have the same semantics here as for the `ALTER TABLESPACE` statement, with the following exceptions:

- You cannot specify the following subclauses for tablespace sets:
  - MINIMUM EXTENT *size_clause*
  - SHRINK SPACE [KEEP *size_clause*]
  - *tablespace_group_clause*
  - *flashback_mode_clause*
  - *tablespace_retention_clause*
- For the *datafile_tempfile_clauses*, only the following subclauses are supported for tablespace sets:
  - RENAME DATAFILE
  - DATAFILE { ONLINE | OFFLINE }
- For the *tablespace_state_clauses*, the PERMANENT and TEMPORARY subclauses are not supported for tablespace sets.

> **See Also:**
>
> *alter_tablespace_attrs* in the documentation on ALTER TABLESPACE for the full semantics of this clause

Examples

Altering a Tablespace Set: Example

The following statement puts all tablespaces in tablespace set ts1 in force logging mode:

```
ALTER TABLESPACE SET ts1
  FORCE LOGGING;
```

ALTER TRIGGER

Purpose

Triggers are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the ALTER TRIGGER statement to enable, disable, or compile a database trigger.

> **Note:**
>
> This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the CREATE TRIGGER statement with the OR REPLACE keywords.

See Also:

- CREATE TRIGGER for information on creating a trigger
- DROP TRIGGER for information on dropping a trigger
- *Oracle Database Concepts* for general information on triggers

Prerequisites

The trigger must be in your own schema or you must have `ALTER ANY TRIGGER` system privilege.

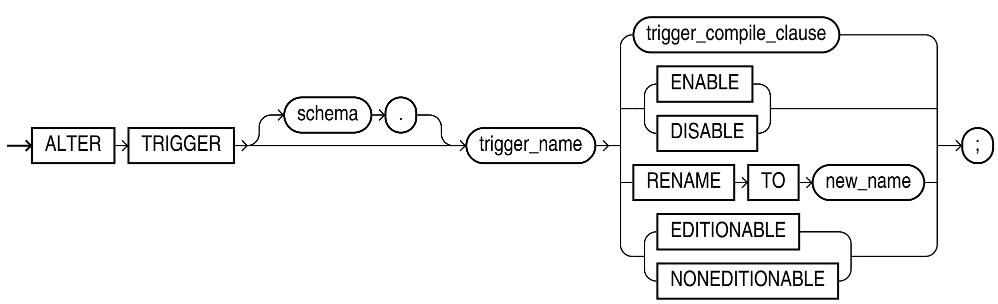
In addition, to alter a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` privilege.

See Also:

CREATE TRIGGER for more information on triggers based on `DATABASE` triggers

Syntax

alter_trigger::=



(*`trigger_compile_clause`*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

schema

Specify the schema containing the trigger. If you omit *`schema`*, then Oracle Database assumes the trigger is in your own schema.

trigger_name

Specify the name of the trigger to be altered.

trigger_compile_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling triggers.

ENABLE | DISABLE

Specify `ENABLE` to enable the trigger. You can also use the `ENABLE ALL TRIGGERS` clause of `ALTER TABLE` to enable all triggers associated with a table. See ALTER TABLE.

Specify `DISABLE` to disable the trigger. You can also use the `DISABLE ALL TRIGGERS` clause of `ALTER TABLE` to disable all triggers associated with a table.

RENAME Clause

Specify `RENAME TO` *`new_name`* to rename the trigger. Oracle Database renames the trigger and leaves it in the same state it was in before being renamed.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the `USER_SOURCE`, `ALL_SOURCE`, and `DBA_SOURCE` data dictionary views. As a result, comments and formatting may change in the `TEXT` column of those views even though the trigger source did not change.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the trigger becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `TRIGGER` in *`schema`*. The default is `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

Restriction on NONEDITIONABLE

You cannot specify `NONEDITIONABLE` for a crossedition trigger.

ALTER TYPE

Purpose

Object types are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `ALTER TYPE` statement to add or drop member attributes or methods. You can change the existing properties (`FINAL` or `INSTANTIABLE`) of an object type, and you can modify the scalar attributes of the type.

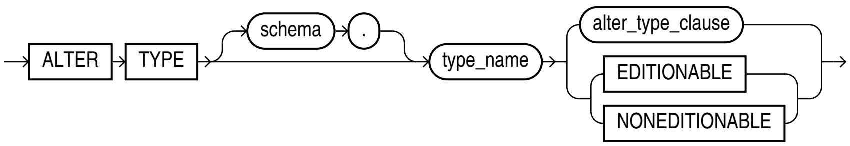
You can also use this statement to recompile the specification or body of the type or to change the specification of an object type by adding new object member subprogram specifications.

Prerequisites

The object type must be in your own schema and you must have `CREATE TYPE` or `CREATE ANY TYPE` system privilege, or you must have `ALTER ANY TYPE` system privileges.

Syntax

***alter_type*::=**



(*alter_type_clause*: See *Oracle Database PL/SQL Language Reference* for the syntax of this clause.)

Semantics

schema

Specify the schema that contains the type. If you omit *schema*, then Oracle Database assumes the type is in your current schema.

type_name

Specify the name of an object type, a nested table type, or a varray type.

Restriction on *type_name*

You cannot evolve an editioned object type. The ALTER TYPE statement fails with ORA-22348 if either of the following is true:

- The type is an editioned object type and the ALTER TYPE statement has no *type_compile_clause*. You can use the ALTER TYPE statement to recompile an editioned object type, but not for any other purpose.
- The type has a dependent that is an editioned object type and the ALTER TYPE statement has a CASCADE clause.

Refer to *Oracle Database PL/SQL Language Reference* for more information on the *type_compile_clause* and the CASCADE clause.

alter_type_clause

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of this clause and for complete information on creating and compiling object types.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the type becomes an editioned or noneditioned object if editioning is later enabled for the schema object type TYPE in *schema*. The default is EDITIONABLE. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

ALTER USER

Purpose

Use the `ALTER USER` statement:

- To change the authentication or database resource characteristics of a database user
- To permit a proxy server to connect as a client without authentication
- In an Oracle Automatic Storage Management (Oracle ASM) cluster, to change the password of a user in the password file that is local to the Oracle ASM instance of the current node

> **See Also:**
>
> *Oracle Database Security Guide* for detailed information about user authentication methods

Prerequisites

In general, you must have the `ALTER USER` system privilege. However, the current user can change his or her own password without this privilege.

To change the password of a `SYS` user, the password file must exist.

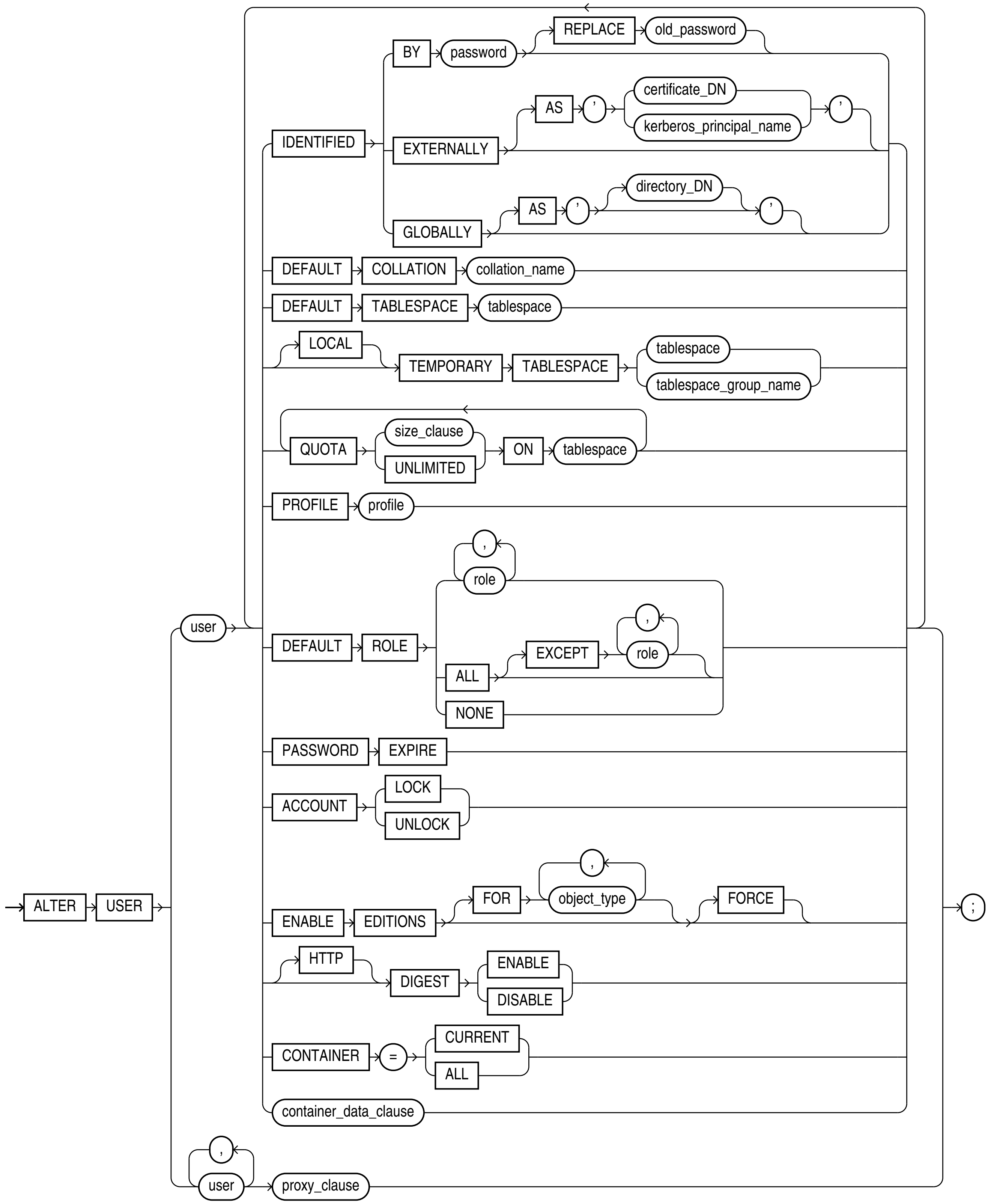
You must be authenticated `AS SYSASM` to change the password of a user other than yourself in an Oracle ASM instance password file.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). If the current container is the root, then you can specify `CONTAINER = ALL` or `CONTAINER = CURRENT`. If the current container is a pluggable database (PDB), then you can specify only `CONTAINER = CURRENT`.

To set and modify `CONTAINER_DATA` attributes using the *`container_data_clause`*, you must be connected to a CDB and the current container must be the root.

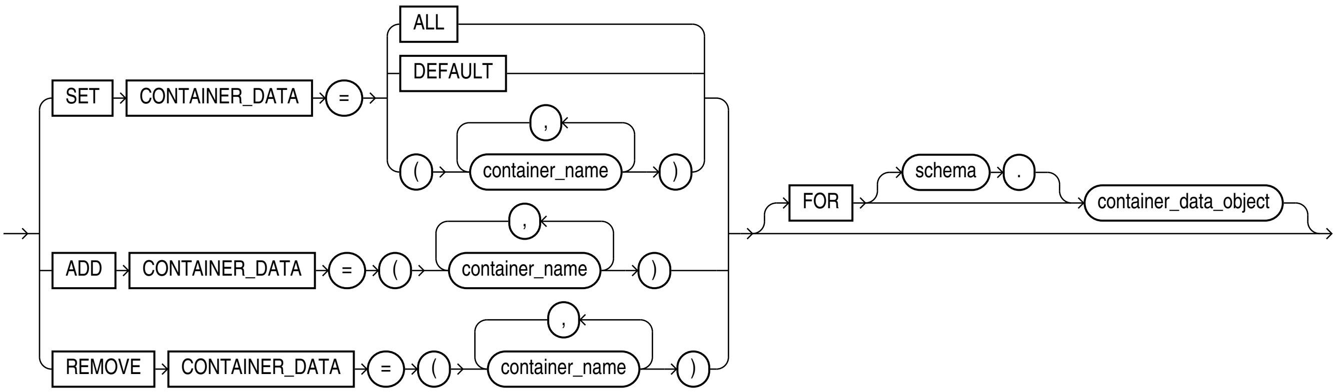
Syntax

***alter_user*::=**

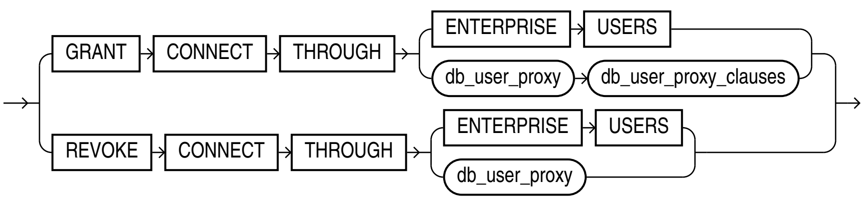


(*size_clause*::=)

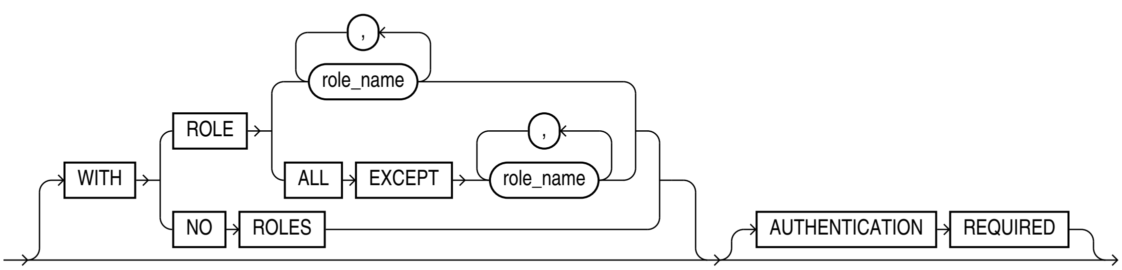
***container_data_clause*::=**



***proxy_clause*::=**



***db_user_proxy_clauses*::=**



Semantics

The keywords, parameters, and clauses described in this section are unique to `ALTER USER` or have different semantics than they have in `CREATE USER`. Keywords, parameters, and clauses that do not appear here have the same meaning as in the `CREATE USER` statement.

> **Note:**
>
> Oracle recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform.

> **See Also:**
>
> CREATE USER for information on the keywords and parameters and CREATE PROFILE for information on assigning limits on database resources to a user

IDENTIFIED Clause

BY *password*

Specify `BY` *`password`* to specify a new password for the user. Passwords are case sensitive. Any subsequent `CONNECT` string used to connect this user to the database must specify the password using the same case (upper, lower, or mixed) that is used in this `ALTER USER` statement. Passwords can contain single-byte, or multibyte characters, or both from your database character set.

> **Note:**
>
> Oracle Database expects a different timestamp for each resetting of a particular password. If you reset one password multiple times within one second (for example, by cycling through a set of passwords using a script), then the database may return an error message that the password cannot be reused. For this reason, Oracle recommends that you avoid using scripts to reset passwords.

You can omit the `REPLACE` clause if you are setting your own password or you have the `ALTER USER` system privilege and you are changing another user's password. However, unless you have the `ALTER USER` system privilege, you must always specify the `REPLACE` clause if a password complexity verification function has been enabled, either by running the `UTLPWDMG.SQL` script or by specifying such a function in the `PASSWORD_VERIFY_FUNCTION` parameter of a profile that has been assigned to the user.

In an Oracle ASM cluster, you can use this clause to change the password of a user in the password file that is local to an Oracle ASM instance of the current node. You must be authenticated `AS SYSASM` to specify `IDENTIFIED BY` *`password`* without the `REPLACE` *`old_password`* clause. If you are not authenticated `AS SYSASM`, then you can only change your own password by specifying `REPLACE` *`old_password`*.

Oracle Database does not check the old password, even if you provide it in the `REPLACE` clause, unless you are changing your own existing password.

> **See Also:**
>
> *Oracle Database Security Guide* for guidelines on creating passwords

GLOBALLY

Refer to CREATE USER for more information on this clause.

You can change a user's access verification method *from* `IDENTIFIED GLOBALLY` to either `IDENTIFIED BY` *`password`* or `IDENTIFIED EXTERNALLY`. You can change a user's access verification method *to* `IDENTIFIED GLOBALLY` from one of the other methods only if all external roles granted explicitly to the user are revoked.

EXTERNALLY

Refer to CREATE USER for more information on this clause.

> **See Also:**
>
> *Oracle Database Enterprise User Security Administrator's Guide* for more information on globally and externally identified users, "Changing User Identification: Example", and "Changing User Authentication: Examples"

NO AUTHENTICATION Clause

Use this clause to change an existing user account with authentication to a schema account without authentication to prevent logins to the account.

DEFAULT COLLATION Clause

Use this clause to change the default collation for the schema owned by the user. The new default collation is assigned to tables, views, and materialized views that are subsequently created in the schema. It does not influence default collations for existing tables views, and materialized views. Refer to the DEFAULT COLLATION Clause clause of `CREATE USER` for the full semantics of this clause.

DEFAULT TABLESPACE Clause

Use this clause to assign or reassign a tablespace for the user's permanent segments. This clause overrides any default tablespace that has been specified for the database.

Restriction on Default Tablespaces

You cannot specify a locally managed temporary tablespace, including an undo tablespace, or a dictionary-managed temporary tablespace, as a user's default tablespace.

[LOCAL] TEMPORARY TABLESPACE Clause

Use this clause to assign or reassign a temporary tablespace or tablespace group for the user's temporary segments.

- Specify *`tablespace`* to indicate the user's temporary tablespace. Specify `TEMPORARY TABLESPACE` to indicate a shared temporary tablespace. Specify `LOCAL TEMPORARY`

TABLESPACE to indicate a local temporary tablespace. If you are connected to a CDB, then you can specify CDB$DEFAULT to use the CDB-wide default temporary tablespace.

- Specify *tablespace_group_name* to indicate that the user can save temporary segments in any tablespace in the tablespace group specified by *tablespace_group_name*. Local temporary tablespaces cannot be part of a tablespace group.

Restriction on User Temporary Tablespace

Any individual tablespace you assign or reassign as the user's temporary tablespace must be a temporary tablespace and must have a standard block size.

> **See Also:**
>
> "Assigning a Tablespace Group: Example"

DEFAULT ROLE Clause

Specify the roles enabled by default for the user at logon.This clause can contain only roles that have been granted directly to the user with a GRANT statement, or roles created by the user with the CREATE ROLE privilege. You cannot use the DEFAULT ROLE clause to specify:

- Roles not granted to the user
- Roles granted through other roles
- Roles managed by an external service (such as the operating system), or by the Oracle Internet Directory
- Roles that are enabled by the SET ROLE statement, such as password-authenticated roles and secure application roles

> **See Also:**
>
> CREATE ROLE

Assigning Default Roles to Common Users in a CDB

You can modify the default role assigned to a common user both in the current container and across all containers in a CDB.

While assigning a default role to a common user across all containers, role must be a common role that was commonly granted to the common user.

While assigning a default role to a common user in the current container, role must be one of the following:

- A local role that was granted to the common user in the current container
- A common role that was granted to the common user, either commonly or locally in the current container

ENABLE EDITIONS

This clause is not reversible. Specify `ENABLE EDITIONS` to allow the user to create multiple versions of editionable objects in this schema using editions. Editionable objects in non-editions-enabled schemas cannot be editioned.

Use the `FOR` clause to specify one or more object types for which the user can create editionable objects. For a list of valid values for *`object_type`*, query the `V$EDITIONABLE_TYPES` dynamic performance view. If you omit the `FOR` clause, then the user can create editionable objects for all editionable object types.

> **See Also:**
>
> *Oracle Database Reference* for more information about the `V$EDITIONABLE_TYPES` dynamic performance view

If the schema to be editions-enabled contains any objects that are not editionable and that depend on editionable type objects in the schema, then you must specify `FORCE` to enable editions for this schema. In this case, all the objects that are not editionable and that depend on the editionable type objects in the schema being editions-enabled become invalid.

[HTTP] DIGEST Clause

This clause lets you enable or disable HTTP Digest Access Authentication for the user.

- Specify `ENABLE` to enable HTTP Digest Access Authentication. After you specify this clause, the user's password must be changed. This causes the database to generate an HTTP Digest verifier for the new password. Only then will HTTP Digest Access Authentication take effect. One way to ensure that the user's password is changed after you issue this clause is to specify the `PASSWORD EXPIRE` clause in the same statement with the `HTTP DIGEST ENABLE` clause, as follows:

  ```
  ALTER USER user PASSWORD EXPIRE HTTP DIGEST ENABLE;
  ```

  This causes the database to prompt the user for a new password on his or her next attempt to log in to the database. After that, HTTP Digest Access Authentication will take effect for the user.

- Specify `DISABLE` to disable HTTP Digest Access Authentication for the user. You do not need to change the user's password in order for this clause to take effect. Specifying the `DISABLE` clause removes the HTTP Digest from dictionary tables.

  ```
  ALTER USER user PASSWORD EXPIRE HTTP DIGEST DISABLE;
  ```

Refer to [HTTP] DIGEST Clause in the documentation on `CREATE USER` for more information on this clause.

CONTAINER Clause

If the current container is a PDB, then you can specify `CONTAINER = CURRENT` to change the attributes of a local user, or the container-specific attributes (such as the default tablespace) of a common user, in the current container. If the current container is the root, then you can specify `CONTAINER = ALL` to change the attributes of a common user across the entire CDB. If you omit this clause and the current container is a PDB, then `CONTAINER = CURRENT` is the default. If you omit this clause and the current container is the root, then `CONTAINER = ALL` is the default.

Restriction on Modifying Common Users in a CDB

Certain attributes of a common user must be modified for all the containers in a CDB and not for only some containers. Therefore, when you use any of the following clauses to modify a common user, ensure that you modify all of the containers by connecting to the root and specifying `CONTAINER=ALL`:

- `IDENTIFIED` clause
- `PASSWORD` clause
- `[HTTP] DIGEST` clause

container_data_clause

The *`container_data_clause`* allows you the set and modify `CONTAINER_DATA` attributes for a common user. Use the `FOR` clause to indicate whether to set or modify the default `CONTAINER_DATA` attribute or an object-specific `CONTAINER_DATA` attribute. These attributes determine the set of containers (which can never exclude the root) whose data will be visible via `CONTAINER_DATA` objects to the specified common user when the current session is the root.

To specify the *`container_data_clause`*, the current session must be the root and you must specify `CONTAINER = CURRENT`.

SET CONTAINER_DATA

Use this clause to set the default `CONTAINER_DATA` attribute or an object-specific `CONTAINER_DATA` attribute for a common user. When you specify this clause, you replace the existing value, if any, of the `CONTAINER_DATA` attribute.

Use *`container_name`* to specify one or more containers that will be accessible to the user.

Use `ALL` to specify that all current and future containers in the CDB will be accessible to the user.

Use `DEFAULT` to specify the default behavior, which is as follows:

- For a default `CONTAINER_DATA` attribute, the current container, that is, the root, and the CDB as a whole will be accessible to the user.
- For an object-specific `CONTAINER_DATA` attribute, the database will use the user's default `CONTAINER_DATA` attribute.

> **Note:**
>
> `CONTAINER_DATA` attributes that are set to `DEFAULT` are not visible in the `DBA_CONTAINER_DATA` view.

ADD CONTAINER_DATA

Use this clause to add containers to the default CONTAINER_DATA attribute or an object-specific CONTAINER_DATA attribute for a common user. Use *container_name* to specify one or more containers to add.

You cannot use this clause if the default CONTAINER_DATA attribute is set to ALL. If you use this clause when the default CONTAINER_DATA attribute is set to DEFAULT, then CDB$ROOT will automatically be added to the set of containers, unless the set already contains CDB$ROOT.

You cannot use this clause if the object-specific CONTAINER_DATA attribute is set to ALL or DEFAULT.

REMOVE CONTAINER_DATA

Use this clause to remove containers from the default CONTAINER_DATA attribute or an object-specific CONTAINER_DATA attribute for a common user. Use *container_name* to specify one or more containers to remove.

You cannot use this clause if the default CONTAINER_DATA attribute or object-specific CONTAINER_DATA attribute is set to ALL or DEFAULT.

FOR *container_data_object*

If you specify the FOR clause, then you can set and modify the object-specific CONTAINER_DATA attribute for *container_data_object* for a common user. *container_data_object* must be a CONTAINER_DATA table or view. If you omit *schema*, then Oracle Database assumes that *container_data_object* is in your own schema.

If you omit the FOR clause, then you can set and modify the default CONTAINER_DATA attribute for a common user.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about enabling common users to view information about PDB objects

proxy_clause

The *proxy_clause* lets you control the ability of an enterprise user (a user outside the database) or a database proxy (another database user) to connect as the database user being altered.

GRANT CONNECT THROUGH

Specify GRANT CONNECT THROUGH to allow the connection.

REVOKE CONNECT THROUGH

Specify REVOKE CONNECT THROUGH to prohibit the connection.

ENTERPRISE USER

This clause lets you expose *user* to proxy use by enterprise users. The administrator working in Oracle Internet Directory must then grant privileges for appropriate enterprise users to act on behalf of *user*.

db_user_proxy

This clause lets you expose `user` to proxy use by database user `db_user_proxy` (the proxy).

- The proxy will have all privileges that were directly granted to user.
- The proxy will have all roles associated with user, unless you specify the `WITH` clauses of `db_user_proxy_clauses` to limit the proxy to some or none of the roles of user. For each role associated with the proxy, if the role is enabled by default for `user` at login, then that role will also be enabled by default for the proxy at login.

db_user_proxy_clauses

Specify the `WITH` clauses to limit the proxy to some or none of the roles associated with `user`, and the `AUTHENTICATION REQUIRED` clause to specify whether authentication is required.

WITH ROLE

`WITH ROLE role_name` permits the proxy to connect as the specified user and to activate only the roles that are specified by `role_name`. This clause can contain only roles that are associated with `user`.

WITH ROLE ALL EXCEPT

`WITH ROLE ALL EXCEPT role_name` permits the proxy to connect as the specified user and to activate all roles associated with that user except those specified for `role_name`. This clause can contain only roles that are associated with `user`.

WITH NO ROLES

`WITH NO ROLES` permits the proxy to connect as the specified user, but prohibits the proxy from activating any of that user's roles after connecting.

AUTHENTICATION REQUIRED

Oracle Database does not expect the proxy to authenticate the user unless you specify the `AUTHENTICATION REQUIRED` clause. This clause ensures that authentication credentials for the user must be presented when the user is authenticated through the specified proxy. The credential is a password.

AUTHENTICATED USING

The `AUTHENTICATED USING` clauses, which appeared in the syntax of earlier releases, have been deprecated and are no longer needed. If you specify the `AUTHENTICATED USING PASSWORD` clause, then Oracle Database converts it to the `AUTHENTICATION REQUIRED` clause. Specifying the `AUTHENTICATED USING CERTIFICATE` clause or the `AUTHENTICATED USING DISTINGUISHED NAME` clause is equivalent to omitting the `AUTHENTICATION REQUIRED` clause.

> **See Also:**
>
> - *Oracle Security Overview* for an overview of database security and for information on middle-tier systems and proxy authentication
> - *Oracle Database Security Guide* for more information on proxies and their use of the database and "Proxy Users: Examples"

Examples

Changing User Identification: Example

The following statement changes the password of the user `sidney` (created in "Creating a Database User: Example") `second_2nd_pwd` and default tablespace to the tablespace `example`:

```
ALTER USER sidney
    IDENTIFIED BY second_2nd_pwd
    DEFAULT TABLESPACE example;
```

The following statement assigns the `new_profile` profile (created in "Creating a Profile: Example") to the sample user `sh`:

```
ALTER USER sh
    PROFILE new_profile;
```

In subsequent sessions, `sh` is restricted by limits in the `new_profile` profile.

The following statement makes all roles granted directly to `sh` default roles, except the `dw_manager` role:

```
ALTER USER sh
    DEFAULT ROLE ALL EXCEPT dw_manager;
```

At the beginning of `sh`'s next session, Oracle Database enables all roles granted directly to `sh` except the `dw_manager` role.

Changing User Authentication: Examples

The following statement changes the authentication mechanism of user `app_user1` (created in "Creating a Database User: Example"):

```
ALTER USER app_user1 IDENTIFIED GLOBALLY AS 'CN=tom,O=oracle,C=US';
```

The following statement causes user `sidney`'s password to expire:

```
ALTER USER sidney PASSWORD EXPIRE;
```

If you cause a database user's password to expire with `PASSWORD EXPIRE`, then the user (or the DBA) must change the password before attempting to log in to the database following the expiration. However, tools such as SQL*Plus allow the user to change the password on the first attempted login following the expiration.

Assigning a Tablespace Group: Example

The following statement assigns `tbs_grp_01` (created in "Adding a Temporary Tablespace to a Tablespace Group: Example") as the tablespace group for user `sh`:

```
ALTER USER sh
  TEMPORARY TABLESPACE tbs_grp_01;
```

Proxy Users: Examples

The following statement alters the user `app_user1`. The example permits the `app_user1` to connect through the proxy user `sh`. The example also allows `app_user1` to enable its `warehouse_user` role (created in "Creating a Role: Example") when connected through the proxy `sh`:

```
ALTER USER app_user1
   GRANT CONNECT THROUGH sh
   WITH ROLE warehouse_user;
```

To show basic syntax, this example uses the sample database Sales History user (`sh`) as the proxy. Normally a proxy user would be an application server or middle-tier entity. For information on creating the interface between an application user and a database by way of an application server, refer to *Oracle Call Interface Programmer's Guide*.

> **See Also:**
>
> - "Creating External Database Users: Examples" to see how to create the `app_user` user
> - "Creating a Role: Example" to see how to create the `dw_user` role

The following statement takes away the right of user `app_user1` to connect through the proxy user `sh`:

```
ALTER USER app_user1 REVOKE CONNECT THROUGH sh;
```

The following hypothetical examples shows another method of proxy authentication:

```
ALTER USER sully GRANT CONNECT THROUGH OAS1
   AUTHENTICATED USING PASSWORD;
```

The following example exposes the user `app_user1` to proxy use by enterprise users. The enterprise users cannot act on behalf of `app_user1` until the Oracle Internet Directory administrator has granted them appropriate privileges:

```
ALTER USER app_user1
   GRANT CONNECT THROUGH ENTERPRISE USERS;
```

ALTER VIEW

Purpose

Use the `ALTER VIEW` statement to explicitly recompile a view that is invalid or to modify view constraints. Explicit recompilation lets you locate recompilation errors before run time. You may want to recompile a view explicitly after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it.

You can also use `ALTER VIEW` to define, modify, or drop view constraints.

You cannot use this statement to change the definition of an existing view. Further, if DDL changes to the view's base tables invalidate the view, then you cannot use this statement to compile the invalid view. In these cases, you must redefine the view using `CREATE VIEW` with the `OR REPLACE` keywords.

When you issue an `ALTER VIEW` statement, Oracle Database recompiles the view regardless of whether it is valid or invalid. The database also invalidates any local objects that depend on the view.

If you alter a view that is referenced by one or more materialized views, then those materialized views are invalidated. Invalid materialized views cannot be used by query rewrite and cannot be refreshed.

See Also:

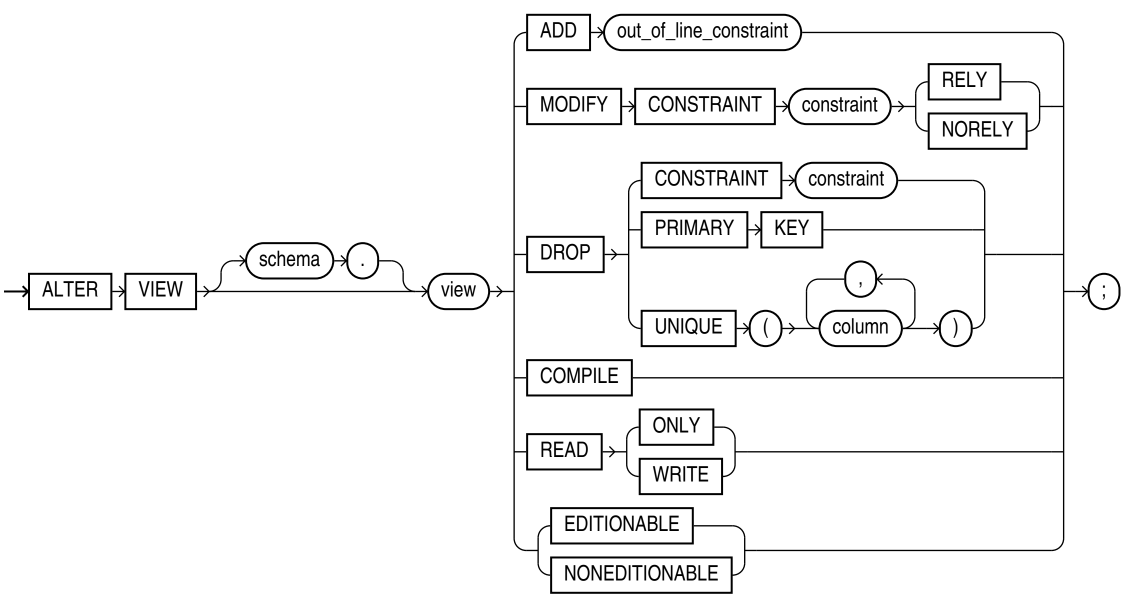
- CREATE VIEW for information on redefining a view and ALTER MATERIALIZED VIEW for information on revalidating an invalid materialized view
- *Oracle Database Data Warehousing Guide* for general information on data warehouses
- *Oracle Database Concepts* for more about dependencies among schema objects

Prerequisites

The view must be in your own schema or you must have `ALTER ANY TABLE` system privilege.

Syntax

alter_view::=



(*out_of_line_constraint*::=—part of *constraint*::= syntax)

Semantics

schema

Specify the schema containing the view. If you omit *`schema`*, then Oracle Database assumes the view is in your own schema.

view

Specify the name of the view to be recompiled.

MODIFY CONSTRAINT Clause

Use the `MODIFY CONSTRAINT` clause to change the `RELY` or `NORELY` setting of an existing view constraint. Refer to "Notes on View Constraints" for general information on view constraints.

Restriction on Modifying Constraints

You cannot change the setting of a unique or primary key constraint if it is part of a referential integrity constraint without dropping the foreign key or changing its setting to match that of *`view`*.

ADD Clause

Use the `ADD` clause to add a constraint to *`view`*. Refer to *constraint* for information on view constraints and their restrictions.

DROP Clause

Use the `DROP` clause to drop an existing view constraint.

Restriction on Dropping Constraints

You cannot drop a unique or primary key constraint if it is part of a referential integrity constraint on a view.

COMPILE

The `COMPILE` keyword directs Oracle Database to recompile the view.

{ READ ONLY | READ WRITE }

These clauses are valid only for editioning views.

- Specify `READ ONLY` to indicate that the editioning view cannot be updated.
- Specify `READ WRITE` to return a read-only editioning view to read/write status.

When you specify these clauses, the database does not invalidate dependent objects, but it may invalidate cursors.

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the view becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `VIEW` in *`schema`*. The default is `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

> **See Also:**
>
> CREATE VIEW for information about editioning views

Examples

Altering a View: Example

To recompile the view `customer_ro` (created in "Creating a Read-Only View: Example"), issue the following statement:

```
ALTER VIEW customer_ro
    COMPILE;
```

If Oracle Database encounters no compilation errors while recompiling `customer_ro`, then `customer_ro` becomes valid. If recompiling results in compilation errors, then the database returns an error and `customer_ro` remains invalid.

Oracle Database also invalidates all dependent objects. These objects include any procedures, functions, package bodies, and views that reference `customer_ro`. If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

ANALYZE

Purpose

Use the `ANALYZE` statement to collect statistics, for example, to:

- Collect or delete statistics about an index or index partition, table or table partition, index-organized table, cluster, or scalar object attribute.
- Validate the structure of an index or index partition, table or table partition, index-organized table, cluster, or object reference (`REF`).
- Identify migrated and chained rows of a table or cluster.

> **Note:**
>
> The use of `ANALYZE` for the collection of optimizer statistics is obsolete.
>
> If you want to collect optimizer statistics, use the `DBMS_STATS` package, which lets you collect statistics in parallel, global statistics for partitioned objects, and helps you fine tune your statistics collection in other ways. See *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_STATS` package.
>
> Use the `ANALYZE` statement only for the following cases:
>
> - To use the `VALIDATE` or `LIST CHAINED ROWS` clauses
> - To collect information on freelist blocks

Prerequisites

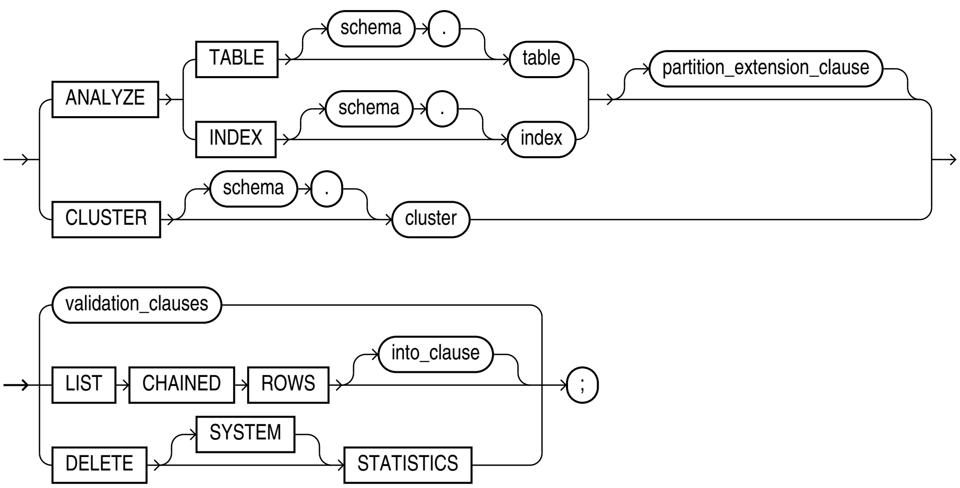
The schema object to be analyzed must be local, and it must be in your own schema or you must have the `ANALYZE ANY` system privilege.

If you want to list chained rows of a table or cluster into a list table, then the list table must be in your own schema, or you must have `INSERT` privilege on the list table, or you must have `INSERT ANY TABLE` system privilege.

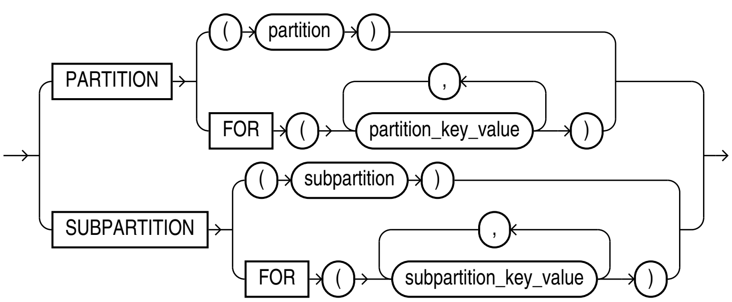
If you want to validate a partitioned table, then you must have the `INSERT` object privilege on the table into which you list analyzed rowids, or you must have the `INSERT ANY TABLE` system privilege.

Syntax

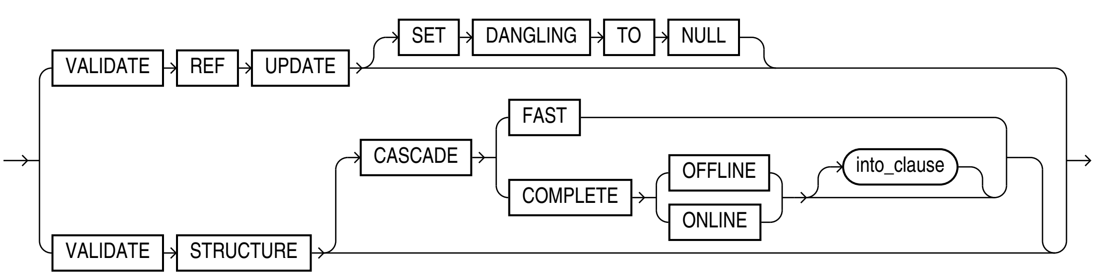
***analyze*::=**



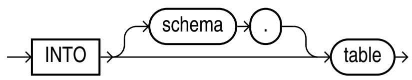
***partition_extension_clause*::=**



***validation_clauses*::=**



***into_clause*::=**



Semantics

schema

Specify the schema containing the table, index, or cluster. If you omit *schema*, then Oracle Database assumes the table, index, or cluster is in your own schema.

TABLE *table*

Specify a table to be analyzed. When you analyze a table, the database collects statistics about expressions occurring in any function-based indexes as well. Therefore, be sure to create function-based indexes on the table before analyzing the table. Refer to CREATE INDEX for more information about function-based indexes.

When analyzing a table, the database skips all domain indexes marked LOADING or FAILED.

For an index-organized table, the database also analyzes any mapping table and calculates its PCT_ACCESSS_DIRECT statistics. These statistics estimate the accuracy of guess data block addresses stored as part of the local rowids in the mapping table.

Oracle Database collects the following statistics for a table. Statistics marked with an asterisk are always computed exactly. Table statistics, including the status of domain indexes, appear in the data dictionary views USER_TABLES, ALL_TABLES, and DBA_TABLES in the columns shown in parentheses.

- Number of rows (NUM_ROWS)
- * Number of data blocks below the high water mark—the number of data blocks that have been formatted to receive data, regardless whether they currently contain data or are empty (BLOCKS)
- * Number of data blocks allocated to the table that have never been used (EMPTY_BLOCKS)
- Average available free space in each data block in bytes (AVG_SPACE)
- Number of chained rows (CHAIN_COUNT)
- Average row length, including the row overhead, in bytes (AVG_ROW_LEN)

Restrictions on Analyzing Tables

Analyzing tables is subject to the following restrictions:

- You cannot use ANALYZE to collect statistics on data dictionary tables.
- You cannot use ANALYZE to collect statistics on an external table. Instead, you must use the DBMS_STATS package.
- You cannot use ANALYZE to collect default statistics on a temporary table. However, if you have already created an association between one or more columns of a temporary table and a user-defined statistics type, then you can use ANALYZE to collect the user-defined statistics on the temporary table.

- You cannot compute or estimate statistics for the following column types: REF column types, varrays, nested tables, LOB column types (LOB column types are not analyzed, they are skipped), LONG column types, or object types. However, if a statistics type is associated with such a column, then Oracle Database collects user-defined statistics.

> **See Also:**
>
> - ASSOCIATE STATISTICS
> - *Oracle Database Reference* for information on the data dictionary views

partition_extension_clause

partition_extension_clause

Specify the partition or subpartition, or the partition or subpartition value, on which you want statistics to be gathered. You cannot use this clause when analyzing clusters.

If you specify PARTITION and *table* is composite-partitioned, then Oracle Database analyzes all the subpartitions within the specified partition.

INDEX *index*

Specify an index to be analyzed.

Oracle Database collects the following statistics for an index. Statistics marked with an asterisk are always computed exactly. For conventional indexes, when you compute or estimate statistics, the statistics appear in the data dictionary views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES in the columns shown in parentheses.

- * Depth of the index from its root block to its leaf blocks (BLEVEL)
- Number of leaf blocks (LEAF_BLOCKS)
- Number of distinct index values (DISTINCT_KEYS)
- Average number of leaf blocks for each index value (AVG_LEAF_BLOCKS_PER_KEY)
- Average number of data blocks for each index value (for an index on a table) (AVG_DATA_BLOCKS_PER_KEY)
- Clustering factor (how well ordered the rows are about the indexed values) (CLUSTERING_FACTOR)

For domain indexes, this statement invokes the user-defined statistics collection function specified in the statistics type associated with the index (see ASSOCIATE STATISTICS). If no statistics type is associated with the domain index, then the statistics type associated with its indextype is used. If no statistics type exists for either the index or its indextype, then no user-defined statistics are collected. User-defined index statistics appear in the STATISTICS column of the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

> **Note:**
>
> - When you analyze an index from which a substantial number of rows has been deleted, Oracle Database sometimes executes a `COMPUTE` statistics operation (which can entail a full table scan) even if you request an `ESTIMATE` statistics operation. Such an operation can be quite time consuming.
> - In some cases, analyzing an index with the `ANALYZE` statement takes an inordinate amount of time to complete. In these cases, you can use a SQL query to validate the index. If the query determines that there is an inconsistency between a table and the index, then you can use the `ANALYZE` statement for a thorough analysis of the index. Refer to *Oracle Database Administrator's Guide* for more information.

Restriction on Analyzing Indexes

You cannot analyze a domain index that is marked `IN_PROGRESS` or `FAILED`.

> **See Also:**
>
> - CREATE INDEX for more information on domain indexes
> - *Oracle Database Reference* for information on the data dictionary views
> - "Analyzing an Index: Example"

CLUSTER *cluster*

Specify a cluster to be analyzed. When you collect statistics for a cluster, Oracle Database also automatically collects the statistics for all the tables in the cluster and all their indexes, including the cluster index.

For both indexed and hash clusters, the database collects the average number of data blocks taken up by a single cluster key (`AVG_BLOCKS_PER_KEY`). These statistics appear in the data dictionary views `ALL_CLUSTERS`, `USER_CLUSTERS`, and `DBA_CLUSTERS`.

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views and "Analyzing a Cluster: Example"

validation_clauses

The validation clauses let you validate `REF` values and the structure of the analyzed object.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about validating tables, indexes, clusters, and materialized views

VALIDATE REF UPDATE Clause

Specify `VALIDATE REF UPDATE` to validate the `REF` values in the specified table, check the rowid portion in each `REF`, compare it with the true rowid, and correct it, if necessary. You can use this clause only when analyzing a table.

If the owner of the table does not have the `READ` or `SELECT` object privilege on the referenced objects, then Oracle Database will consider them invalid and set them to null. Subsequently these `REF` values will not be available in a query, even if it is issued by a user with appropriate privileges on the objects.

SET DANGLING TO NULL

`SET DANGLING TO NULL` sets to null any `REF` values (whether or not scoped) in the specified table that are found to point to an invalid or nonexistent object.

VALIDATE STRUCTURE

Specify `VALIDATE STRUCTURE` to validate the structure of the analyzed object. The statistics collected by this clause are not used by the Oracle Database optimizer.

> **See Also:**
>
> "Validating a Table: Example"

- For a table, Oracle Database verifies the integrity of each of the data blocks and rows. For an index-organized table, the database also generates compression statistics (optimal prefix compression count) for the primary key index on the table.
- For a cluster, Oracle Database automatically validates the structure of the cluster tables.
- For a partitioned table, Oracle Database also verifies that each row belongs to the correct partition. If a row does not collate correctly, then its rowid is inserted into the `INVALID_ROWS` table.
- For a temporary table, Oracle Database validates the structure of the table and its indexes during the current session.
- For an index, Oracle Database verifies the integrity of each data block in the index and checks for block corruption. This clause does not confirm that each row in the table has an index entry or that each index entry points to a row in the table. You can perform these operations by validating the structure of the table with the CASCADE clause.

  Oracle Database also computes compression statistics (optimal prefix compression count) for all normal indexes.

Oracle Database stores statistics about the index in the data dictionary views INDEX_STATS and INDEX_HISTOGRAM.

> **See Also:**
>
> *Oracle Database Reference* for information on these views

If Oracle Database encounters corruption in the structure of the object, then an error message is returned. In this case, drop and re-create the object.

CASCADE

Specify CASCADE if you want Oracle Database to validate the structure of the indexes associated with the table or cluster. If you use this clause when validating a table, then the database also validates the indexes defined on the table. If you use this clause when validating a cluster, then the database also validates all the cluster tables indexes, including the cluster index.

By default, CASCADE performs a COMPLETE validation, which can be resource intensive. Specify FAST if you want the database to check for the existence of corruptions without reporting details about the corruption. If the FAST check finds a corruption, you can then use the CASCADE option without the FAST clause to locate and learn details about it.

If you use this clause to validate an enabled (but previously disabled) function-based index, then validation errors may result. In this case, you must rebuild the index.

ONLINE | OFFLINE

Specify ONLINE to enable Oracle Database to run the validation while DML operations are ongoing within the object. The database reduces the amount of validation performed to allow for concurrency.

> **Note:**
>
> When you validate the structure of an object ONLINE, Oracle Database does not collect any statistics, as it does when you validate the structure of the object OFFLINE.

Specify OFFLINE, to maximize the amount of validation performed. This setting prevents INSERT, UPDATE, and DELETE statements from concurrently accessing the object during validation but allows queries. This is the default.

Restriction on ONLINE

You cannot specify ONLINE when analyzing a cluster.

INTO

The INTO clause of VALIDATE STRUCTURE is valid only for partitioned tables. Specify a table into which Oracle Database lists the rowids of the partitions whose rows do not collate correctly. If you omit *schema*, then the database assumes the list is in your own schema. If you omit this clause altogether, then the database assumes that the table is named INVALID_ROWS. The SQL script used to create this table is UTLVALID.SQL.

LIST CHAINED ROWS

`LIST CHAINED ROWS` lets you identify migrated and chained rows of the analyzed table or cluster. You cannot use this clause when analyzing an index.

In the `INTO` clause, specify a table into which Oracle Database lists the migrated and chained rows. If you omit *`schema`*, then the database assumes the chained-rows table is in your own schema. If you omit this clause altogether, then the database assumes that the table is named `CHAINED_ROWS`. The chained-rows table must be on your local database.

You can create the `CHAINED_ROWS` table using one of these scripts:

- `UTLCHAIN.SQL` uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- `UTLCHN1.SQL` uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own chained-rows table, then it must follow the format prescribed by one of these two scripts.

If you are analyzing index-organized tables based on primary keys (rather than universal rowids), then you must create a separate chained-rows table for each index-organized table to accommodate its primary-key storage. Use the SQL scripts `DBMSIOTC.SQL` and `PRVTIOTC.PLB` to define the `BUILD_CHAIN_ROWS_TABLE` procedure, and then execute this procedure to create an `IOT_CHAINED_ROWS` table for each such index-organized table.

See Also:

- The `DBMS_IOT` package in *Oracle Database PL/SQL Packages and Types Reference* for information on the packaged SQL scripts
- "Listing Chained Rows: Example"

DELETE STATISTICS

Specify `DELETE STATISTICS` to delete any statistics about the analyzed object that are currently stored in the data dictionary. Use this statement when you no longer want Oracle Database to use the statistics.

When you use this clause on a table, the database also automatically removes statistics for all the indexes defined on the table. When you use this clause on a cluster, the database also automatically removes statistics for all the cluster tables and all their indexes, including the cluster index.

Specify `SYSTEM` if you want Oracle Database to delete only system (not user-defined) statistics. If you omit `SYSTEM`, and if user-defined column or index statistics were collected for an object, then the database also removes the user-defined statistics by invoking the statistics deletion function specified in the statistics type that was used to collect the statistics.

See Also:

"Deleting Statistics: Example"

Examples

Deleting Statistics: Example

The following statement deletes statistics about the sample table `oe.orders` and all its indexes from the data dictionary:

```
ANALYZE TABLE orders DELETE STATISTICS;
```

Analyzing an Index: Example

The following statement validates the structure of the sample index `oe.inv_product_ix`:

```
ANALYZE INDEX inv_product_ix VALIDATE STRUCTURE;
```

Validating a Table: Example

The following statement analyzes the sample table `hr.employees` and all of its indexes:

```
ANALYZE TABLE employees VALIDATE STRUCTURE CASCADE;
```

For a table, the `VALIDATE REF UPDATE` clause verifies the `REF` values in the specified table, checks the rowid portion of each `REF`, and then compares it with the true rowid. If the result is an incorrect rowid, then the `REF` is updated so that the rowid portion is correct.

The following statement validates the `REF` values in the sample table `oe.customers`:

```
ANALYZE TABLE customers VALIDATE REF UPDATE;
```

The following statement validates the structure of the sample table `oe.customers` while allowing simultaneous DML:

```
ANALYZE TABLE customers VALIDATE STRUCTURE ONLINE;
```

Analyzing a Cluster: Example

The following statement analyzes the `personnel` cluster (created in "Creating a Cluster: Example"), all of its tables, and all of their indexes, including the cluster index:

```
ANALYZE CLUSTER personnel
    VALIDATE STRUCTURE CASCADE;
```

Listing Chained Rows: Example

The following statement collects information about all the chained rows in the table `orders`:

```
ANALYZE TABLE orders
   LIST CHAINED ROWS INTO chained_rows;
```

The preceding statement places the information into the table `chained_rows`. You can then examine the rows with this query (no rows will be returned if the table contains no chained rows):

```
SELECT owner_name, table_name, head_rowid, analyze_timestamp
    FROM chained_rows
```

```
   ORDER BY owner_name, table_name, head_rowid, analyze_timestamp;

OWNER_NAME  TABLE_NAME  HEAD_ROWID          ANALYZE_TIMESTAMP
----------  ----------  ------------------ -----------------
OE          ORDERS      AAAAZzAABAAABrXAAA 25-SEP-2000
```

ASSOCIATE STATISTICS

Purpose

Use the `ASSOCIATE STATISTICS` statement to associate a statistics type (or default statistics) containing functions relevant to statistics collection, selectivity, or cost with one or more columns, standalone functions, packages, types, domain indexes, or indextypes.

For a listing of all current statistics type associations, query the `USER_ASSOCIATIONS` data dictionary view. If you analyze the object with which you are associating statistics, then you can also query the associations in the `USER_USTATS` view.

> **See Also:**
>
> ANALYZE for information on the order of precedence with which `ANALYZE` uses associations

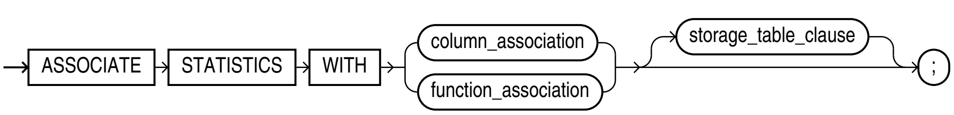
Prerequisites

To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype). In addition, unless you are associating only default statistics, you must have execute privilege on the statistics type. The statistics type must already have been defined.

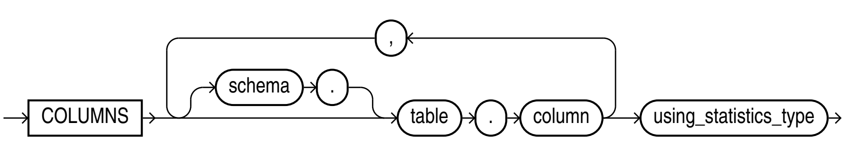
> **See Also:**
>
> CREATE TYPE for information on defining types

Syntax

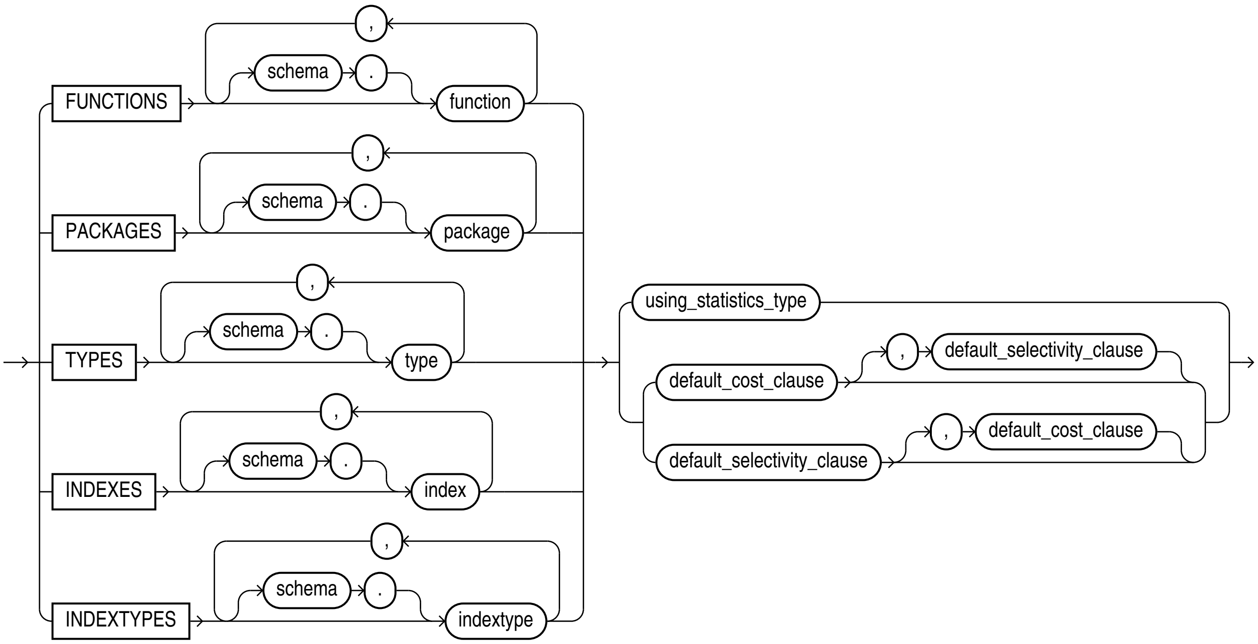
***associate_statistics*::=**



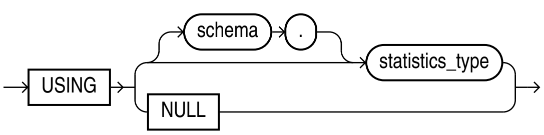
***column_association*::=**



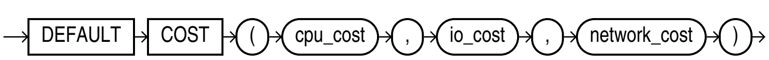
***function_association*::=**



***using_statistics_type*::=**



***default_cost_clause*::=**



***default_selectivity_clause*::=**

***storage_table_clause*::=**

Semantics

column_association

Specify one or more table columns. If you do not specify *schema*, then Oracle Database assumes the table is in your own schema.

function_association

Specify one or more standalone functions, packages, user-defined data types, domain indexes, or indextypes. If you do not specify *schema*, then Oracle Database assumes the object is in your own schema.

- FUNCTIONS refers only to standalone functions, not to method types or to built-in functions.
- TYPES refers only to user-defined types, not to built-in SQL data types.

Restriction on *function_association*

You cannot specify an object for which you have already defined an association. You must first disassociate the statistics from this object.

> **See Also:**
>
> DISASSOCIATE STATISTICS "Associating Statistics: Example"

using_statistics_type

Specify the statistics type (or a synonym for the type) being associated with column, function, package, type, domain index, or indextype. The *statistics_type* must already have been created.

The NULL keyword is valid only when you are associating statistics with a column or an index. When you associate a statistics type with an object type, columns of that object type inherit the statistics type. Likewise, when you associate a statistics type with an indextype, index instances of the indextype inherit the statistics type.You can override this inheritance by associating a different statistics type for the column or index. Alternatively, if you do not want to associate any statistics type for the column or index, then you can specify NULL in the *using_statistics_type* clause.

Restriction on Specifying Statistics Type

You cannot specify NULL for functions, packages, types, or indextypes.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information on creating statistics collection functions

default_cost_clause

Specify default costs for standalone functions, packages, types, domain indexes, or indextypes. If you specify this clause, then you must include one number each for CPU cost, I/O cost, and network cost, in that order. Each cost is for a single execution of the function or method or for a single domain index access. Accepted values are integers of zero or greater.

default_selectivity_clause

Specify as a percent the default selectivity for predicates with standalone functions, types, packages, or user-defined operators. The *default_selectivity_clause* must be a number between 0 and 100. Values outside this range are ignored.

Restriction on the *default_selectivity_clause*

You cannot specify DEFAULT SELECTIVITY for domain indexes or indextypes.

> **See Also:**
>
> "Specifying Default Cost: Example"

storage_table_clause

This clause is relevant only for statistics on INDEXTYPE.

- Specify WITH SYSTEM MANAGED STORAGE TABLES to indicate that the storage of statistics data is to be managed by the system. The type you specify in *statistics_type* should be storing the statistics related information in tables that are maintained by the system. Also, the indextype you specify must already have been created or altered to support the WITH SYSTEM MANAGED STORAGE TABLES clause.
- Specify WITH USER MANAGED STORAGE TABLES to indicate that the tables that store the user-defined statistics will be managed by the user. This is the default behavior.

Examples

Associating Statistics: Example

This statement creates an association for the standalone package emp_mgmt. See *Oracle Database PL/SQL Language Reference* for the example that creates this package.

```
ASSOCIATE STATISTICS WITH PACKAGES emp_mgmt DEFAULT SELECTIVITY 10;
```

Specifying Default Cost: Example

This statement specifies that using the domain index salary_index, created in "Using Extensible Indexing ", to implement a given predicate always has a CPU cost of 100, I/O cost of 5, and network cost of 0.

```
ASSOCIATE STATISTICS WITH INDEXES salary_index DEFAULT COST (100,5,0);
```

The optimizer will use these default costs instead of calling a cost function.

AUDIT (Traditional Auditing)

This section describes the AUDIT statement for **traditional auditing**, which is the same auditing functionality used in releases earlier than Oracle Database 12*c*.

Beginning with Oracle Database 12*c*, Oracle introduces **unified auditing**, which provides a full set of enhanced auditing features. For backward compatibility, traditional auditing is still supported. However, Oracle recommends that you plan the migration of your existing audit settings to the new unified audit policy syntax. For new audit requirements, Oracle recommends that you use the new unified auditing. Traditional auditing may be desupported in a future major release.

> **See Also:**
>
> AUDIT (Unified Auditing) for a description of the AUDIT statement for unified auditing

Purpose

Use the AUDIT statement to:

- Track the issuance of SQL statements in subsequent user sessions. You can track the issuance of a specific SQL statement or of all SQL statements authorized by a particular system privilege. Auditing operations on SQL statements apply only to subsequent sessions, not to current sessions.
- Track operations on a specific schema object. Auditing operations on schema objects apply to current sessions as well as to subsequent sessions.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for information on the DBMS_FGA package, which lets you create and administer value-based auditing policies
> - NOAUDIT (Traditional Auditing)

Prerequisites

To audit issuances of a SQL statement, you must have the AUDIT SYSTEM system privilege. However, the AUDIT SYSTEM system privilege is not required when you use the IN SESSION CURRENT clause.

To collect auditing results, you must enable auditing by setting the initialization parameter AUDIT_TRAIL to a value other than the default setting of NONE. You can specify auditing options regardless of whether auditing is enabled. However, Oracle Database does not generate audit records until you enable auditing.

To audit operations on a schema object, the object you choose for auditing must be in your own schema or you must have AUDIT ANY system privilege. In addition, if the object you choose for auditing is a directory object, even if you created it, then you must have AUDIT ANY system privilege.

When you are connected to a multitenant container database (CDB), you must have the privileges described in this section, either granted locally in the current container or granted commonly.

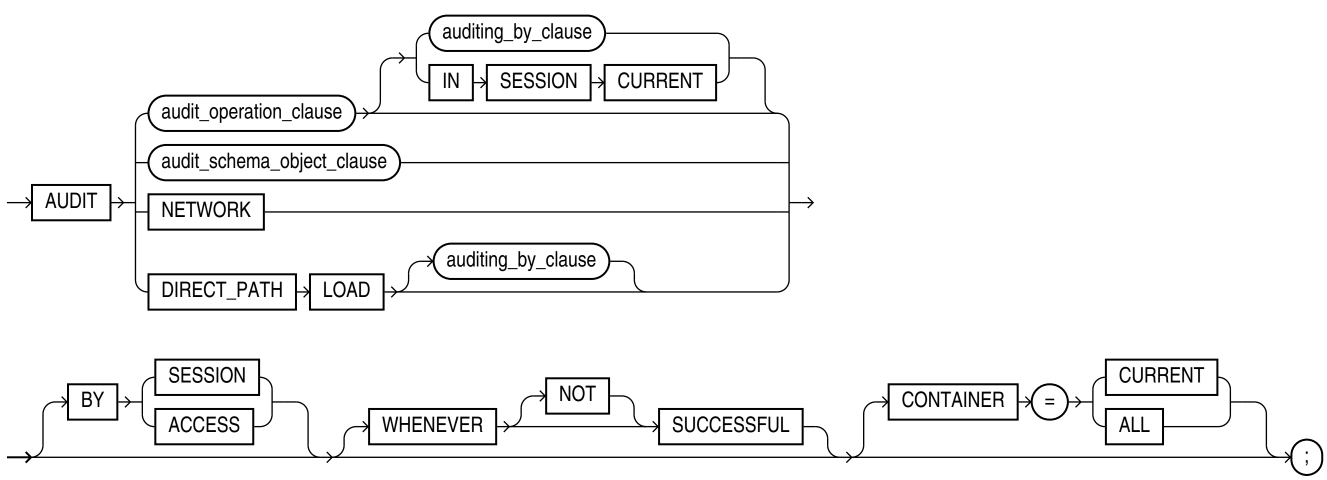
To specify the CONTAINER clause, you must be connected to a multitenant container database (CDB). To specify CONTAINER = CURRENT, the current container must be a pluggable database (PDB). To specify CONTAINER = ALL, the current container must be the root.

> **Note:**
>
> The AUDIT ANY system privileges allows the grantee to audit any object in any schema except the SYS schema. You can allow such a grantee to audit objects in the SYS schema by setting the O7_DICTIONARY_ACCESSIBILITY initialization parameter to TRUE. For security reasons, Oracle recommends that you use this setting only with great caution.

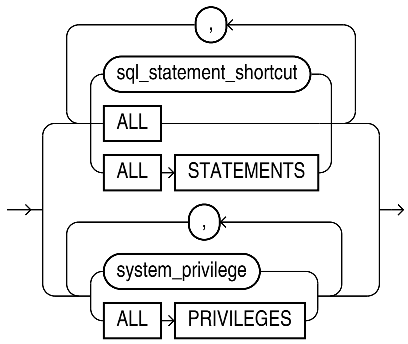
> **See Also:**
>
> *Oracle Database Reference* for information on the AUDIT_TRAIL parameter

Syntax

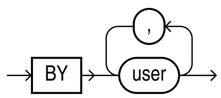
***audit*::=**



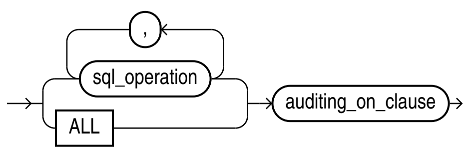
***audit_operation_clause*::=**



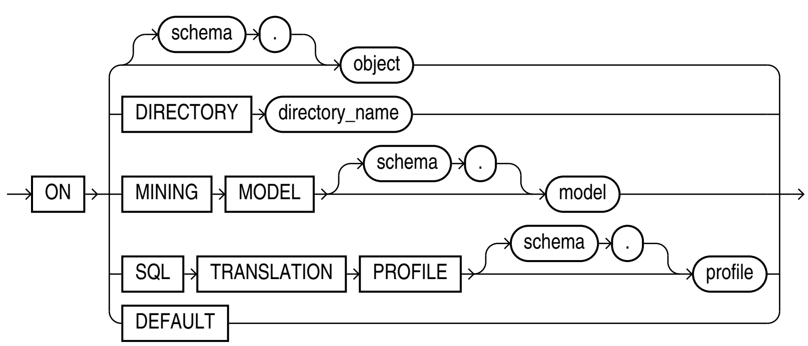
***auditing_by_clause*::=**



***audit_schema_object_clause*::=**



***auditing_on_clause*::=**



Semantics

Notes on Using the AUDIT Statement in a CDB

When you issue the `AUDIT` statement in a CDB, the database performs auditing as follows:

- If you issue the AUDIT statement when the current container is a PDB, then the database performs auditing in that PDB. If you specify the *auditing_by_clause*, then user must be a local user in the PDB or a common user. If you specify the *audit_schema_object_clause*, then the object must be a local object in the PDB.
- If you issue the AUDIT statement when the current container is the root, then the database performs auditing across the entire CDB, that is, in the root and all PDBs. If you specify the *auditing_by_clause*, then user must be a common user. If you omit the *auditing_by_clause*, then all local users in each PDB and all common users are audited. If you specify the *audit_schema_object_clause*, then the object must be a local object in the root or a common object.

audit_operation_clause

Use the *audit_operation_clause* to audit specified operations, regardless of the schema objects affected by the operations.

sql_statement_shortcut

Specify a shortcut to audit the use of specific SQL statements. Table 12-1 and Table 12-2 list the shortcuts and the SQL statements they audit.

Note:

Do not confuse SQL statement shortcuts with system privileges. For example:

- An AUDIT USER statement specifies the USER *shortcut* for auditing of all CREATE USER, ALTER USER, and DROP USER SQL statements. Auditing in this case includes an operation in which a user changes his or her own password with an ALTER USER statement.
- An AUDIT ALTER USER statement specifies the ALTER USER *system privilege* for auditing of all operations that make use of that system privilege. Auditing in this case does *not* include an operation in which a user changes his or her own password, because that operation does not require the ALTER USER system privilege.

For each audited operation, Oracle Database produces an audit record containing this information:

- The user performing the operation
- The type of operation
- The object involved in the operation
- The date and time of the operation

Oracle Database writes audit records to the audit trail, which is a database table containing audit records. You can review database activity by examining the audit trail through data dictionary views.

> **See Also:**
>
> - *Oracle Database Security Guide 11g Release 2 (11.2)* for a listing of the audit trail data dictionary views. Refer to *Oracle Database Upgrade Guide* for instructions on how to locate the Oracle Database 11*g* Release 2 (11.2) documentation.
> - *Oracle Database Reference* for detailed descriptions of the data dictionary views
> - "Auditing SQL Statements Relating to Roles: Example"

system_privilege

Specify a system privilege to audit SQL statements and other operations that are authorized by the specified system privilege.

> **Note:**
>
> Auditing the use of a system privilege containing the `ANY` keyword is more restrictive than auditing the use of the same privilege without the `ANY` keyword. For example:
>
> - `AUDIT CREATE PROCEDURE` audits the statements issued using either the `CREATE PROCEDURE` or `CREATE ANY PROCEDURE` privilege.
> - `AUDIT CREATE ANY PROCEDURE` audits only those statements issued using the `CREATE ANY PROCEDURE` privilege.

Rather than specifying many individual system privileges, you can specify the roles `CONNECT`, `RESOURCE`, and `DBA`. Doing so is equivalent to auditing all of the system privileges granted to those roles.

Oracle Database also provides three shortcuts for specifying groups of system privileges and statement options at once:

ALL

Specify `ALL` to audit all statements options shown in Table 12-1 but not the additional statement options shown in Table 12-2.

ALL STATEMENTS

Specify `ALL STATEMENTS` to audit all **top-level SQL statements** executed. Top-level SQL statements are issued directly by a user. SQL statements run from within a PL/SQL procedure or function are not considered top-level statements. Therefore, this clause does not audit the statements executed within PL/SQL procedures or functions. However, the execution of the PL/SQL procedure or function itself is audited. This clause is useful if you want to audit all the statements in a specific environment, regardless of other auditing configurations that are system wide or user specific.

ALL PRIVILEGES

Specify `ALL PRIVILEGES` to audit system privileges.

Note:

Oracle recommends that you specify individual system privileges and statement options for auditing rather than roles or shortcuts. The specific system privileges and statement options encompassed by roles and shortcuts change from one release to the next and may not be supported in future versions of Oracle Database.

See Also:

- Table 18-1 for a list of all system privileges and the operations that they authorize
- *Oracle Database Security Guide* for more information on the `CONNECT`, `RESOURCE`, and `DBA` roles
- "Auditing Query and Update SQL Statements: Example", "Auditing Deletions: Example", and "Auditing Statements Relating to Directories: Examples"

auditing_by_clause

Specify the *`auditing_by_clause`* to restrict auditing to only SQL statements issued by the specified users. If you omit this clause, then Oracle Database audits all users' statements.

IN SESSION CURRENT

Use this clause to limit auditing to the current session. Auditing will persist until the end of the session and cannot be stopped using the `NOAUDIT` statement.

audit_schema_object_clause

Use the *`audit_schema_object_clause`* to audit operations on specific schema objects.

Restriction on the *audit_schema_object_clause*

When connected to a CDB, you can specify the *`audit_schema_object_clause`*, but you cannot also specify the `CONTAINER` clause. This restriction does not limit functionality because the only allowed values for the `CONTAINER` clause are the default values. Refer to CONTAINER Clause for more information.

sql_operation

Specify the SQL operation to be audited. Table 12-3 shows the types of objects that can be audited, and for each object the SQL statements that can be audited. For example, if you choose to audit a table with the `ALTER` operation, then Oracle Database audits all `ALTER TABLE` statements issued against the table. If you choose to audit a sequence with the `SELECT` operation, then the database audits all statements that use any values of the sequence.

ALL

Specify `ALL` as a shortcut equivalent to specifying all SQL operations applicable for the type of object.

auditing_on_clause

The *auditing_on_clause* lets you specify the particular schema object to be audited.

> **See Also:**
>
> "Auditing Queries on a Table: Example", "Auditing Inserts and Updates on a Table: Example", and "Auditing Operations on a Sequence: Example"

schema

Specify the schema containing the object chosen for auditing. If you omit *schema*, then Oracle Database assumes the object is in your own schema.

object

Specify the name of the object to be audited. The object must be a table, view, sequence, stored procedure, function, package, materialized view, mining model, or library.

You can also specify a synonym for a table, view, sequence, procedure, stored function, package, materialized view, or user-defined type.

ON DEFAULT

Specify ON DEFAULT to establish the specified object options as default object options for subsequently created objects. After you have established these default auditing options, any subsequently created object is automatically audited with those options. The default auditing options for a view are always the union of the auditing options for the base tables of the view. You can see the current default auditing options by querying the ALL_DEF_AUDIT_OPTS data dictionary view.

When you change the default auditing options, the auditing options for previously created objects remain the same. You can change the auditing options for an existing object only by specifying the object in the ON clause of the AUDIT statement.

> **See Also:**
>
> "Setting Default Auditing Options: Example"

ON DIRECTORY

The ON DIRECTORY clause lets you specify the name of a directory chosen for auditing.

ON MINING MODEL

The ON MINING MODEL clause lets you specify the name of a mining model to be audited.

ON SQL TRANSLATION PROFILE

The ON SQL TRANSLATION PROFILE clause lets you specify the name of a SQL translation profile to be audited.

NETWORK

Use this clause to detect internal failures in the network layer.

> **See Also:**
>
> *Oracle Database Security Guide 11g Release 2 (11.2)* for information on network auditing. Refer to *Oracle Database Upgrade Guide* for instructions on how to locate the Oracle Database 11*g* Release 2 (11.2) documentation.

DIRECT_PATH LOAD

Use this clause to audit SQL*Loader direct path loads.

BY SESSION

In earlier releases, `BY SESSION` caused the database to write a single record for all SQL statements or operations of the same type executed on the same schema objects in the same session. Beginning with this release of Oracle Database, both `BY SESSION` and `BY ACCESS` cause Oracle Database to write one audit record for each audited statement and operation. `BY SESSION` continues to populate different values to the audit trail compared with `BY ACCESS`. Oracle recommends that you include the `BY ACCESS` clause for all `AUDIT` statements, which results in a more detailed audit record. If you specify neither clause, then `BY ACCESS` is the default.

> **Note:**
>
> This change applies only to schema object audit options, statement options and system privileges that audit SQL statements other than data definition language (DDL) statements. The database has always audited `BY ACCESS` all SQL statements and system privileges that audit a DDL statement.

BY ACCESS

Specify `BY ACCESS` if you want Oracle Database to write one record for each audited statement and operation.

> **Note:**
>
> If you specify either a SQL statement shortcut or a system privilege that audits a data definition language (DDL) statement, then the database always audits by access. In all other cases, the database honors the `BY SESSION` or `BY ACCESS` specification.

For statement options and system privileges that audit SQL statements other than DDL, you can specify either `BY SESSION` or `BY ACCESS`. `BY ACCESS` is the default.

WHENEVER [NOT] SUCCESSFUL

Specify `WHENEVER SUCCESSFUL` to audit only SQL statements and operations that succeed.

Specify `WHENEVER NOT SUCCESSFUL` to audit only SQL statements and operations that fail or result in errors.

If you omit this clause, then Oracle Database performs the audit regardless of success or failure.

CONTAINER Clause

The `CONTAINER` clause applies only when you are connected to a CDB. You can use this clause to specify the scope of the `AUDIT` statement. However, it is not necessary to specify the `CONTAINER` clause because its default values are the only allowed values.

- If you issue the `AUDIT` statement when the current container is a PDB, then you can optionally specify `CONTAINER = CURRENT`, which is the default.
- If you issue the `AUDIT` statement when the current container is the root, then you can optionally specify `CONTAINER = ALL`, which is the default.

Tables of Auditing Options

Table 12-1 SQL Statement Shortcuts for Auditing

SQL Statement Shortcut	SQL Statements and Operations Audited
ALTER SYSTEM	ALTER SYSTEM
CLUSTER	CREATE CLUSTER ALTER CLUSTER DROP CLUSTER TRUNCATE CLUSTER
CONTEXT	CREATE CONTEXT DROP CONTEXT
DATABASE LINK	CREATE DATABASE LINK ALTER DATABASE LINK DROP DATABASE LINK
DIMENSION	CREATE DIMENSION ALTER DIMENSION DROP DIMENSION
DIRECTORY	CREATE DIRECTORY DROP DIRECTORY
INDEX	CREATE INDEX ALTER INDEX ANALYZE INDEX DROP INDEX

Table 12-1 (Cont.) SQL Statement Shortcuts for Auditing

SQL Statement Shortcut	SQL Statements and Operations Audited

`MATERIALIZED VIEW`	`CREATE MATERIALIZED VIEW`
	`ALTER MATERIALIZED VIEW`
	`DROP MATERIALIZED VIEW`

`NOT EXISTS`	All SQL statements that fail because a specified object does not exist.

`OUTLINE`	`CREATE OUTLINE`
	`ALTER OUTLINE`
	`DROP OUTLINE`

`PLUGGABLE DATABASE`	`CREATE PLUGGABLE DATABASE`
	`ALTER PLUGGABLE DATABASE`
	`DROP PLUGGABLE DATABASE`

`PROCEDURE` (See note at end of table)	`CREATE FUNCTION`
	`CREATE LIBRARY`
	`CREATE PACKAGE`
	`CREATE PACKAGE BODY`
	`CREATE PROCEDURE`
	`DROP FUNCTION`
	`DROP LIBRARY`
	`DROP PACKAGE`
	`DROP PROCEDURE`

`PROFILE`	`CREATE PROFILE`
	`ALTER PROFILE`
	`DROP PROFILE`

`PUBLIC DATABASE LINK`	`CREATE PUBLIC DATABASE LINK`
	`ALTER PUBLIC DATABASE LINK`
	`DROP PUBLIC DATABASE LINK`

`PUBLIC SYNONYM`	`CREATE PUBLIC SYNONYM` `DROP PUBLIC SYNONYM`

`ROLE`	`CREATE ROLE`
	`ALTER ROLE`
	`DROP ROLE`
	`SET ROLE`

`ROLLBACK SEGMENT`	`CREATE ROLLBACK SEGMENT`
	`ALTER ROLLBACK SEGMENT`
	`DROP ROLLBACK SEGMENT`

`SEQUENCE`	`CREATE SEQUENCE` `DROP SEQUENCE`
`SESSION`	Logons

Table 12-1 (Cont.) SQL Statement Shortcuts for Auditing

SQL Statement Shortcut	SQL Statements and Operations Audited
`SYNONYM`	`CREATE SYNONYM` `DROP SYNONYM`
`SYSTEM AUDIT`	`AUDIT` *`sql_statements`* `NOAUDIT` *`sql_statements`*
`SYSTEM GRANT`	`GRANT` *`system_privileges_and_roles`* `REVOKE` *`system_privileges_and_roles`*

`TABLE`	`CREATE TABLE`
	`DROP TABLE`
	`TRUNCATE TABLE`

`TABLESPACE`	`CREATE TABLESPACE`
	`ALTER TABLESPACE`
	`DROP TABLESPACE`

`TRIGGER`	`CREATE TRIGGER`
	`ALTER TRIGGER`
	with `ENABLE` and `DISABLE` clauses
	`DROP TRIGGER`
	`ALTER TABLE`
	with `ENABLE ALL TRIGGERS` clause
	and `DISABLE ALL TRIGGERS` clause

`TYPE`	`CREATE TYPE`
	`CREATE TYPE BODY`
	`ALTER TYPE`
	`DROP TYPE`
	`DROP TYPE BODY`

`USER`	`CREATE USER`
	`ALTER USER`
	`DROP USER`
	Notes:
	• `AUDIT USER` audits these three SQL statements. Use `AUDIT ALTER USER` to audit statements that require the `ALTER USER` system privilege.
	• An `AUDIT ALTER USER` statement does not audit a user changing his or her own password, as this activity does not require the `ALTER USER` system privilege.

`VIEW`	`CREATE VIEW` `DROP VIEW`

> **Note:**
>
> Java schema objects (sources, classes, and resources) are considered the same as procedures for purposes of auditing SQL statements.

Table 12-2 Additional SQL Statement Shortcuts for Auditing

SQL Statement Shortcut	SQL Statements and Operations Audited
`ALTER SEQUENCE`	`ALTER SEQUENCE`
`ALTER TABLE`	`ALTER TABLE`
`COMMENT TABLE`	`COMMENT ON TABLE` *`table`*, *`view`*, *`materialized view`* `COMMENT ON COLUMN` *`table.column`*, *`view.column`*, *`materialized view.column`*
`DELETE TABLE`	`DELETE FROM` *`table`*, *`view`*
`EXECUTE DIRECTORY`	Execution of any program in a directory
`EXECUTE PROCEDURE`	`CALL` Execution of any procedure or function or access to any variable, library, or cursor inside a package
`GRANT DIRECTORY`	`GRANT` privilege `ON` directory `REVOKE` privilege `ON` directory
`GRANT PROCEDURE`	`GRANT` privilege `ON` procedure, function, package `REVOKE` privilege `ON` procedure, function, package
`GRANT SEQUENCE`	`GRANT` privilege `ON` sequence `REVOKE` privilege `ON` sequence
`GRANT TABLE`	`GRANT` privilege `ON` table, view, materialized view `REVOKE` privilege `ON` table, view, materialized view
`GRANT TYPE`	`GRANT` privilege `ON TYPE` `REVOKE` privilege `ON TYPE`
`INSERT TABLE`	`INSERT INTO` table, view
`LOCK TABLE`	`LOCK TABLE` table, view
`READ DIRECTORY`	Read operations on a directory
`SELECT SEQUENCE`	Any statement containing *`sequence`*`.CURRVAL` or *`sequence`*`.NEXTVAL`
`SELECT TABLE`	`SELECT FROM` table, view, materialized view
`UPDATE TABLE`	`UPDATE` table, view
`WRITE DIRECTORY`	Write operations on a directory

Table 12-3 Schema Object Auditing Options

Object	SQL Operations

Table	`ALTER`
	`AUDIT`
	`COMMENT`
	`DELETE`
	`FLASHBACK` (Note 1)
	`GRANT`
	`INDEX`
	`INSERT`
	`LOCK`
	`RENAME`
	`SELECT`
	`UPDATE`

View	`AUDIT`
	`COMMENT`
	`DELETE`
	`FLASHBACK` (Note 1)
	`GRANT`
	`INSERT`
	`LOCK`
	`RENAME`
	`SELECT`
	`UPDATE`

Sequence	`ALTER`
	`AUDIT`
	`GRANT`
	`SELECT`

Procedure, Function, Package (Note 2)	`AUDIT`
	`EXECUTE` (Notes 3 and 4)
	`GRANT`

Materialized View (Note 5)	`ALTER`
	`AUDIT`
	`COMMENT`
	`DELETE`
	`INDEX`
	`INSERT`
	`LOCK`
	`SELECT`
	`UPDATE`

Table 12-3 (Cont.) Schema Object Auditing Options

Object	SQL Operations
Mining Model	AUDIT COMMENT GRANT RENAME SELECT
Directory	AUDIT GRANT READ
Library	EXECUTE GRANT
Object Type	ALTER AUDIT GRANT

Note 1: The FLASHBACK audit object option applies only to flashback queries.

Note 2: Java schema objects (sources, classes, and resources) are considered the same as procedures, functions, and packages for purposes of auditing options.

Note 3: When you audit the EXECUTE operation on a PL/SQL stored procedure or stored function, the database considers only its ability to find the procedure or function and authorize its execution when determining the success or failure of the operation for the purposes of auditing. Therefore, if you specify the WHENEVER NOT SUCCESSFUL clause, then only invalid object errors, non-existent object errors, and authorization failures are audited; errors encountered during the execution of the procedure or function are not audited. If you specify the WHENEVER SUCCESSFUL clause, then all executions that are not blocked by invalid object errors, non-existent object errors, or authorization failures are audited, regardless of whether errors are encountered during execution.

Note 4: To audit the failure of a recursive SQL operation inside a PL/SQL stored procedure or stored function, configure auditing for the SQL operation.

Note 5: You can audit INSERT, UPDATE, and DELETE operations only on updatable materialized views.

Examples

Auditing SQL Statements Relating to Roles: Example

To choose auditing for every SQL statement that creates, alters, drops, or sets a role, regardless of whether the statement completes successfully, issue the following statement:

```
AUDIT ROLE;
```

To choose auditing for every statement that successfully creates, alters, drops, or sets a role, issue the following statement:

```
AUDIT ROLE
    WHENEVER SUCCESSFUL;
```

To choose auditing for every CREATE ROLE, ALTER ROLE, DROP ROLE, or SET ROLE statement that results in an Oracle Database error, issue the following statement:

```
AUDIT ROLE
    WHENEVER NOT SUCCESSFUL;
```

Auditing Query and Update SQL Statements: Example

To choose auditing for any statement that queries or updates any table, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE;
```

To choose auditing for statements issued by the users hr and oe that query or update a table or view, issue the following statement

```
AUDIT SELECT TABLE, UPDATE TABLE
    BY hr, oe;
```

Auditing Deletions: Example

To choose auditing for statements issued using the DELETE ANY TABLE system privilege, issue the following statement:

```
AUDIT DELETE ANY TABLE;
```

Auditing Statements Relating to Directories: Examples

To choose auditing for statements issued using the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT CREATE ANY DIRECTORY;
```

To choose auditing for CREATE DIRECTORY (and DROP DIRECTORY) statements that do not use the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT DIRECTORY;
```

To choose auditing for every statement that reads files from the bfile_dir directory, issue the following statement:

```
AUDIT READ ON DIRECTORY bfile_dir;
```

To choose auditing for every statement that reads files from any directory, issue the following statement:

```
AUDIT READ DIRECTORY;
```

Auditing Queries on a Table: Example

To choose auditing for every SQL statement that queries the employees table in the schema hr, issue the following statement:

```
AUDIT SELECT
    ON hr.employees;
```

To choose auditing for every statement that successfully queries the employees table in the schema hr, issue the following statement:

```
AUDIT SELECT
    ON hr.employees
    WHENEVER SUCCESSFUL;
```

To choose auditing for every statement that queries the `employees` table in the schema `hr` and results in an Oracle Database error, issue the following statement:

```
AUDIT SELECT
    ON hr.employees
    WHENEVER NOT SUCCESSFUL;
```

Auditing Inserts and Updates on a Table: Example

To choose auditing for every statement that inserts or updates a row in the `customers` table in the schema `oe`, issue the following statement:

```
AUDIT INSERT, UPDATE
    ON oe.customers;
```

Auditing Operations on a Sequence: Example

To choose auditing for every statement that performs any operation on the `employees_seq` sequence in the schema `hr`, issue the following statement:

```
AUDIT ALL
    ON hr.employees_seq;
```

The preceding statement uses the `ALL` shortcut to choose auditing for the following statements that operate on the sequence:

- `ALTER SEQUENCE`
- `AUDIT`
- `GRANT`
- any statement that accesses the values of the sequence using the pseudocolumns `CURRVAL` or `NEXTVAL`

Setting Default Auditing Options: Example

The following statement specifies default auditing options for objects created in the future:

```
AUDIT ALTER, GRANT, INSERT, UPDATE, DELETE
    ON DEFAULT;
```

Any objects created later are automatically configured for audit with the specified options that apply to them.

- If you create a table, then Oracle Database automatically configures audit options `ALTER`, `GRANT`, `INSERT`, `UPDATE`, or `DELETE` issued against the table.
- If you create a view, then Oracle Database automatically configures audit options `GRANT`, `INSERT`, `UPDATE`, or `DELETE` against the view.
- If you create a sequence, then Oracle Database automatically configures audit options `ALTER` or `GRANT` against the sequence.
- If you create a procedure, package, or function, then Oracle Database automatically configures audit options `ALTER` or `GRANT` against it.

AUDIT (Unified Auditing)

This section describes the AUDIT statement for **unified auditing**. This type of auditing is new beginning with Oracle Database 12*c* and provides a full set of enhanced auditing features. Refer to *Oracle Database Security Guide* for more information on unified auditing.

Purpose

Use the AUDIT statement to:

- Enable a unified audit policy for all users or for specified users
- Specify whether an audit record is created if the audited event fails, succeeds, or both
- Specify application context attributes, whose values will be recorded in audit records

Operations performed with this statement take effect in subsequent user sessions, not in the current session.

> **See Also:**
>
> - NOAUDIT (Unified Auditing)
> - CREATE AUDIT POLICY (Unified Auditing)
> - ALTER AUDIT POLICY (Unified Auditing)
> - DROP AUDIT POLICY (Unified Auditing)

Prerequisites

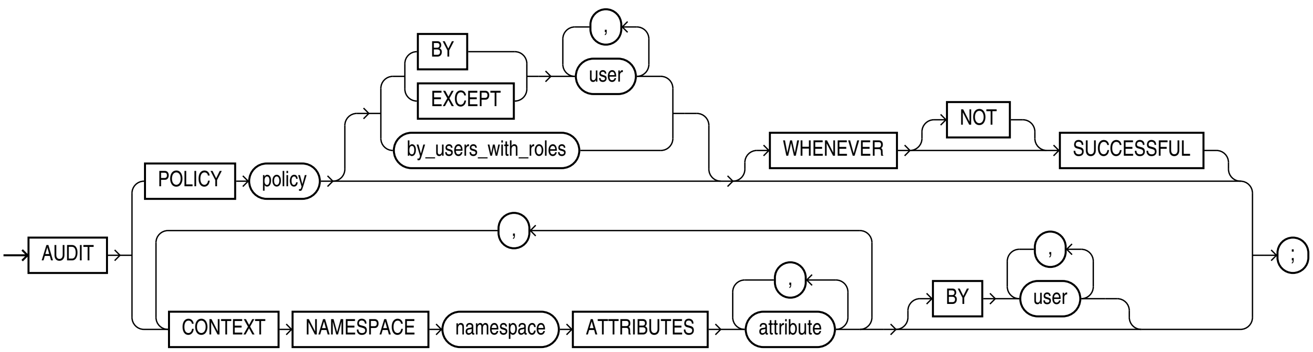
You must have the AUDIT SYSTEM system privilege or the AUDIT_ADMIN role.

If you are connected to a multitenant container database (CDB), then to enable a common unified audit policy, the current container must be the root and you must have the commonly granted AUDIT SYSTEM privilege or the AUDIT_ADMIN common role. To enable a local unified audit policy, the current container must be the container in which the audit policy was created and you must have the commonly granted AUDIT SYSTEM privilege or the AUDIT_ADMIN common role, or you must have the locally granted AUDIT SYSTEM privilege or the AUDIT_ADMIN local role in the container.

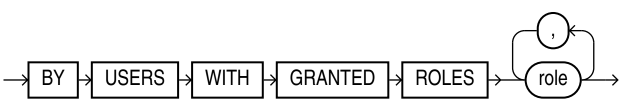
To specify the AUDIT CONTEXT ... statement when connected to a CDB, you must have the commonly granted AUDIT SYSTEM privilege or the AUDIT_ADMIN common role, or you must have the locally granted AUDIT SYSTEM privilege or the AUDIT_ADMIN local role in the current session's container.

Syntax

***unified_audit*::=**



***by_users_with_roles*::=**



Semantics

policy

Specify the name of the unified audit policy to be enabled. The policy must have been created using the `CREATE AUDIT POLICY` statement.

You can find descriptions of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view and descriptions of all *enabled* unified audit policies by querying the `AUDIT_UNIFIED_ENABLED_POLICIES` view.

When you enable a unified audit policy, all SQL statements and operations that satisfy either a system privilege or action or role audit option specified in the enabled policy will be audited —that is, a unified audit record will be created in the `UNIFIED_AUDIT_TRAIL` view. If a single SQL statement or operation satisfies multiple enabled policies, then only one unified audit record will be created and all satisfied audit policy names will appear in a comma-separated list in the `UNIFIED_AUDIT_POLICIES` column of the `UNIFIED_AUDIT_TRAIL` view.

> **See Also:**
>
> - CREATE AUDIT POLICY (Unified Auditing)
> - *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES`, `AUDIT_UNIFIED_ENABLED_POLICIES`, and `UNIFIED_AUDIT_TRAIL` views

BY | EXCEPT

Specify the BY clause to enable *policy* for only the specified users.

Specify the EXCEPT clause to enable *policy* for all users except the specified users.

If you omit the BY and EXCEPT clauses and the *by_users_with_roles* clause, then Oracle Database enables *policy* for all users.

If *policy* is a common unified audit policy, then *user* must be a common user. If *policy* is a local unified audit policy, then *user* must be a common user or a local user in the container to which you are connected.

Notes on the BY and EXCEPT Clauses

The following notes apply to the BY and EXCEPT clauses:

- If multiple AUDIT ... BY ... statements are specified for the same unified audit policy, then the policy is enabled for the union of the users specified in each statement.
- If multiple AUDIT ... EXCEPT ... statements are specified for the same unified audit policy, then only the most recently specified statement takes effect. That is, the policy is enabled for all users except the users specified in the most recent AUDIT ... EXCEPT ... statement.
- If a policy is enabled using the BY clause and you would like to instead enable it using the EXCEPT clause, then you must first use the NOAUDIT ... BY ... statement to disable the policy for all users for whom the policy is currently enabled, and then enable the policy with the AUDIT ... EXCEPT ... statement.
- If a policy is enabled using the EXCEPT clause and you would like to instead enable it using the BY clause, then you must first use the NOAUDIT statement to disable the audit policy. Note that you cannot specify the EXCEPT clause with the NOAUDIT statement. You can then enable the policy with the AUDIT ... BY ... statement.

Restriction on the BY and EXCEPT Clauses

You cannot specify an AUDIT ... BY ... statement and an AUDIT ... EXCEPT ... statement for the same unified audit policy. If you attempt to do so, then an error occurs.

by_users_with_roles

Specify this clause to enable *policy* for users who have been directly or indirectly granted the specified roles. If you subsequently grant one of the roles to an additional user or to a role which is directly or indirectly granted to a user, then the policy automatically applies to that user. If you subsequently revoke one of the roles from a user or from a role which was directly or indirectly granted to a role or a user, then the policy no longer applies to that user.

When you are connected to a CDB, if *policy* is a common unified audit policy, then *role* must be a common role. If *policy* is a local unified audit policy, then *role* must be a common role or a local role in the container to which you are connected.

Enabling a Local Audit Policy on Roles

Local audit policy can be enabled on local roles as well as on common roles. When a local audit policy is enabled on a common role, it generates audit records when a common role is granted to user locally or commonly in the container.

Enabling a Common Audit Policy on Roles

Common audit policy can only be enabled on common roles. When a common audit policy is enabled on a common role, it generates audit records when a common role is granted to an user commonly or locally in the ROOT container.

WHENEVER [NOT] SUCCESSFUL

Specify `WHENEVER SUCCESSFUL` to audit only SQL statements and operations that succeed.

Specify `WHENEVER NOT SUCCESSFUL` to audit only SQL statements and operations that fail or result in errors.

If you omit this clause, then Oracle Database performs the audit regardless of success or failure.

CONTEXT Clause

Specify the `CONTEXT` clause to include the values of context attributes in audit records.

- For *`namespace`*, specify the context namespace.
- For *`attribute`*, specify one or more context attributes whose values you want to include in audit records.
- Use the optional `BY` *`user`* clause to include the values of the context attributes only in audit records for events executed by the specified users. If you omit the `BY` clause, then the values of the context attributes are included in all audit records.

If you specify the `CONTEXT` clause when the current container is the root of a CDB, then the values of context attributes will be included in audit records only for events executed in the root. If you specify the optional `BY` clause, then *`user`* must be a common user.

If you specify the `CONTEXT` clause when the current container is a pluggable database (PDB), then the values of context attributes will be included in audit records only for events executed in that PDB. If you specify the optional `BY` clause, then *`user`* must be a common user or a local user in that PDB.

You can find the application context attributes that are configured to be captured in the audit trail by querying the `AUDIT_UNIFIED_CONTEXTS` view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_CONTEXTS` view

Examples

The following examples enable unified audit policies that were created in the `CREATE AUDIT POLICY` "Examples".

Enabling a Unified Audit Policy for All Users: Example

The following statement enables unified audit policy `table_pol` for all users:

```
AUDIT POLICY table_pol;
```

The following statement verifies that `table_pol` is enabled for all users:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'TABLE_POL';

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
TABLE_POL    BY           ALL USERS
```

Enabling a Unified Audit Policy for Specific Users: Examples

The following statement enables unified audit policy `dml_pol` for only users `hr` and `sh`:

```
AUDIT POLICY dml_pol BY hr, sh;
```

The following statement verifies that `dml_pol` is enabled for only users `hr` and `sh`:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL'
  ORDER BY user_name;

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
DML_POL      BY           HR
DML_POL      BY           SH
```

The following statement enables unified audit policy `read_dir_pol` for all users except `hr`:

```
AUDIT POLICY read_dir_pol EXCEPT hr;
```

The following statement verifies that `read_dir_pol` is enabled for all users except `hr`:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'READ_DIR_POL';

POLICY_NAME   ENABLED_OPTION  ENTITY_NAME
------------  -----------  ---------
READ_DIR_POL  EXCEPT       HR
```

The following statement enables unified audit policy `security_pol` for user `hr` and audits only the SQL statements and operations that fail:

```
AUDIT POLICY security_pol BY hr WHENEVER NOT SUCCESSFUL;
```

The following statement verifies that `security_pol` is enabled for only user `hr` and that only the SQL statements and operations that fail will be audited:

```
SELECT policy_name, enabled_option, entity_name, success, failure
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'SECURITY_POL';

POLICY_NAME   ENABLED_OPTION           ENTITY_NAME   SUCCESS  FAILURE
------------  --------------------  ----------  -------  -------
SECURITY_POL  BY                    HR          NO       YES
```

Including Values of Context Attributes in Audit Records: Example

The following statement instructs the database to include the values of namespace `USERENV` attributes `CURRENT_USER` and `DB_NAME` in all audit records for user `hr`:

```
AUDIT CONTEXT NAMESPACE userenv
  ATTRIBUTES current_user, db_name
  BY hr;
```

CALL

Purpose

Use the `CALL` statement to execute a **routine** (a standalone procedure or function, or a procedure or function defined within a type or package) from within SQL.

> **Note:**
>
> The restrictions on user-defined function expressions specified in "Function Expressions " apply to the `CALL` statement as well.

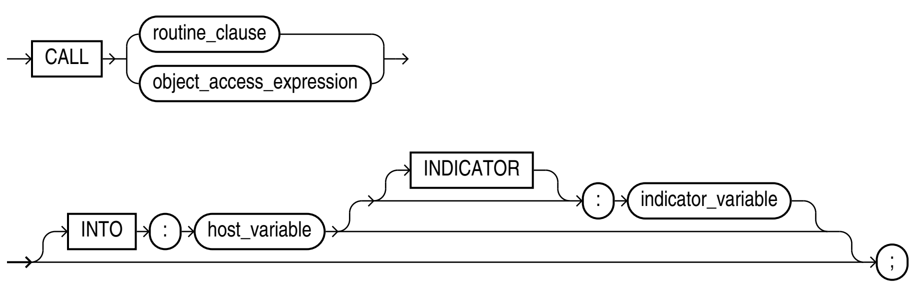
> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information on creating such routine

Prerequisites

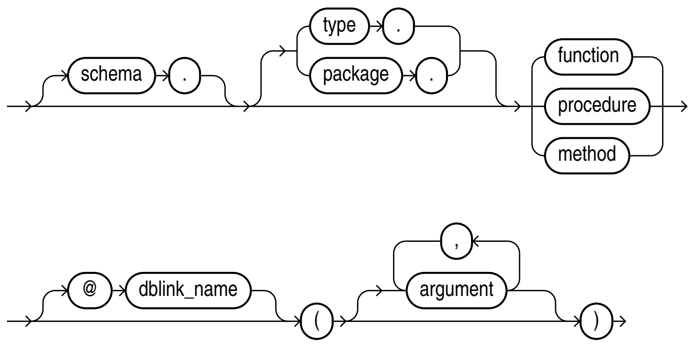
You must have `EXECUTE` privilege on the standalone routine or on the type or package in which the routine is defined.

Syntax

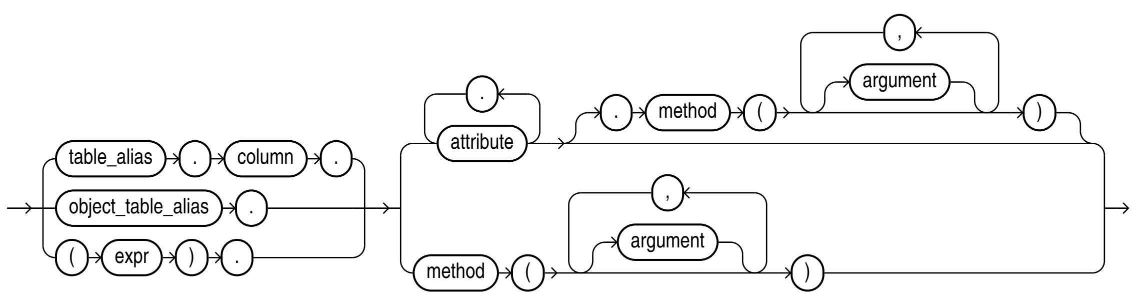
***call*::=**



***routine_clause*::=**



***object_access_expression*::=**



Semantics

You can execute a routine in two ways. You can issue a call to the routine itself by name, by using the `routine_clause`, or you can invoke a routine inside the type of an expression, by using an `object_access_expression`.

routine_clause

Specify the name of the function or procedure being called, or a synonym that resolves to a function or procedure.

When you call a member function or procedure of a type, if the first argument (`SELF`) is a null `IN OUT` argument, then Oracle Database returns an error. If `SELF` is a null `IN` argument, then the database returns null. In both cases, the function or procedure is not invoked.

Restriction on Functions

If the routine is a function, then the `INTO` clause is required.

schema

Specify the schema in which the standalone routine, or the package or type containing the routine, resides. If you do not specify `schema`, then Oracle Database assumes the routine is in your own schema.

type or *package*

Specify the type or package in which the routine is defined.

@dblink

In a distributed database system, specify the name of the database containing the standalone routine, or the package or function containing the routine. If you omit *dblink*, then Oracle Database looks in your local database.

> **See Also:**
>
> "Calling a Procedure: Example" for an example of calling a routine directly

object_access_expression

If you have an expression of an object type, such as a type constructor or a bind variable, then you can use this form of expression to call a routine defined within the type. In this context, the *object_access_expression* is limited to method invocations.

> **See Also:**
>
> "Object Access Expressions " for syntax and semantics of this form of expression, and "Calling a Procedure Using an Expression of an Object Type: Example" for an example of calling a routine using an expression of an object type

argument

Specify one or more arguments to the routine, if the routine takes arguments. You can use positional, named, or mixed notation for *argument*. For example, all of the following notations are correct:

```
CALL my_procedure(arg1 => 3, arg2 => 4)

CALL my_procedure(3, 4)

CALL my_procedure(3, arg2 => 4)
```

Restrictions on Applying Arguments to Routines

The *argument* is subject to the following restrictions:

- The data types of the parameters passed by the CALL statement must be SQL data types. They cannot be PL/SQL-only data types such as BOOLEAN.
- An *argument* cannot be a pseudocolumn or either of the object reference functions VALUE or REF.
- Any *argument* that is an IN OUT or OUT argument of the routine must correspond to a host variable expression.
- The number of arguments, including any return argument, is limited to 1000.
- You cannot bind arguments of character and raw data types (CHAR, VARCHAR2, NCHAR, NVARCHAR2, RAW, LONG RAW) that are larger than 4K.

INTO :*host_variable*

The `INTO` clause applies only to calls to functions. Specify which host variable will store the return value of the function.

:*indicator_variable*

Specify the value or condition of the host variable.

> **See Also:**
>
> *Pro*C/C++ Programmer's Guide* for more information on host variables and indicator variables

Examples

Calling a Procedure: Example

The following statement removes the Entertainment department (created in "Inserting Sequence Values: Example") using uses the `remove_dept` procedure. See *Oracle Database PL/SQL Language Reference* for the example that creates this procedure.

```
CALL emp_mgmt.remove_dept(162);
```

Calling a Procedure Using an Expression of an Object Type: Example

The following examples show how call a procedure by using an expression of an object type in the `CALL` statement. The example uses the `warehouse_typ` object type in the order entry sample schema `OE`:

```
ALTER TYPE warehouse_typ
      ADD MEMBER FUNCTION ret_name
      RETURN VARCHAR2
      CASCADE;

CREATE OR REPLACE TYPE BODY warehouse_typ
      AS MEMBER FUNCTION ret_name
      RETURN VARCHAR2
      IS
         BEGIN
            RETURN self.warehouse_name;
         END;
      END;
/
VARIABLE x VARCHAR2(25);

CALL warehouse_typ(456, 'Warehouse 456', 2236).ret_name()
   INTO :x;

PRINT x;
X
--------------------------------
Warehouse 456
```

The next example shows how to use an external function to achieve the same thing:

```
CREATE OR REPLACE FUNCTION ret_warehouse_typ(x warehouse_typ)
  RETURN warehouse_typ
  IS
    BEGIN
      RETURN x;
    END;
/
CALL ret_warehouse_typ(warehouse_typ(234, 'Warehouse 234',
   2235)).ret_name()
   INTO :x;

PRINT x;

X
--------------------------------
Warehouse 234
```

COMMENT

Purpose

Use the `COMMENT` statement to add to the data dictionary a comment about a table or table column, unified audit policy, edition, indextype, materialized view, mining model, operator, or view.

To drop a comment from the database, set it to the empty string ' '.

> **See Also:**
>
> - "Comments " for more information on associating comments with SQL statements and schema objects
> - *Oracle Database Reference* for information on the data dictionary views that display comments

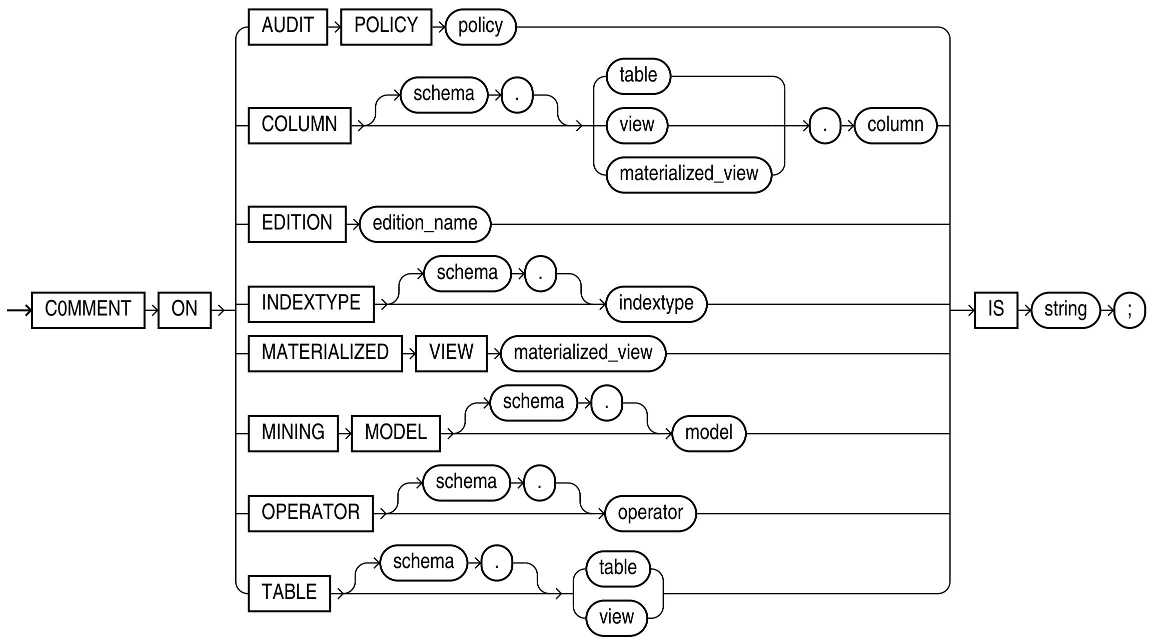
Prerequisites

The object about which you are adding a comment must be in your own schema or:

- To add a comment to a table, view, or materialized view, you must have `COMMENT ANY TABLE` system privilege.
- To add a comment to a unified audit policy, you must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.
- To add a comment to an edition, you must have the `CREATE ANY EDITION` system privilege, granted either directly or through a role.
- To add a comment to an indextype, you must have the `CREATE ANY INDEXTYPE` system privilege.
- To add a comment to a mining model, you must have the `COMMENT ANY MINING MODEL` system privilege.
- To add a comment to an operator, you must have the `CREATE ANY OPERATOR` system privilege.

Syntax

***comment*::=**



Semantics

AUDIT POLICY Clause

Specify the name of the unified audit policy to be commented.

You can view the comments on a particular unified audit policy by querying the `AUDIT_UNIFIED_POLICY_COMMENTS` data dictionary view.

COLUMN Clause

Specify the name of the column of a table, view, or materialized view to be commented. If you omit *`schema`*, then Oracle Database assumes the table, view, or materialized view is in your own schema.

You can view the comments on a particular table or column by querying the data dictionary views `USER_TAB_COMMENTS`, `DBA_TAB_COMMENTS`, or `ALL_TAB_COMMENTS` or `USER_COL_COMMENTS`, `DBA_COL_COMMENTS`, or `ALL_COL_COMMENTS`.

EDITION Clause

Specify the name of an existing edition to be commented.

You can query the data dictionary view `ALL_EDITION_COMMENTS` to view comments associated with editions that are accessible to the current user. You can query `DBA_EDITION_COMMENTS` to view comments associated with all editions in the database.

TABLE Clause

Specify the schema and name of the table or materialized view to be commented. If you omit *schema*, then Oracle Database assumes the table or materialized view is in your own schema.

> **Note:**
>
> In earlier releases, you could use this clause to create a comment on a materialized view. You should now use the `COMMENT ON MATERIALIZED VIEW` clause for materialized views.

INDEXTYPE Clause

Specify the name of the indextype to be commented. If you omit *schema*, then Oracle Database assumes the indextype is in your own schema.

You can view the comments on a particular indextype by querying the data dictionary views `USER_INDEXTYPE_COMMENTS`, `DBA_INDEXTYPE_COMMENTS`, or `ALL_INDEXTYPE_COMMENTS`.

MATERIALIZED VIEW Clause

Specify the name of the materialized view to be commented. If you omit *schema*, then Oracle Database assumes the materialized view is in your own schema.

You can view the comments on a particular materialized view by querying the data dictionary views `USER_MVIEW_COMMENTS`, `DBA_MVIEW_COMMENTS`, or `ALL_MVIEW_COMMENTS`.

MINING MODEL

Specify the name of the mining model to be commented.

You can view the comments on a particular mining model by querying the `COMMENTS` column of the data dictionary views `USER_MINING_MODELS`, `DBA_MINING_MODELS`, or `ALL_MINING_MODELS`.

OPERATOR Clause

Specify the name of the operator to be commented. If you omit *schema*, then Oracle Database assumes the operator is in your own schema.

You can view the comments on a particular operator by querying the data dictionary views `USER_OPERATOR_COMMENTS`, `DBA_OPERATOR_COMMENTS`, or `ALL_OPERATOR_COMMENTS`.

IS '*string*'

Specify the text of the comment. Refer to "Text Literals " for a syntax description of *'string'*.

Examples

Creating Comments: Example

To insert an explanatory remark on the `job_id` column of the `employees` table, you might issue the following statement:

```
COMMENT ON COLUMN employees.job_id
   IS 'abbreviated job title';
```

To drop this comment from the database, issue the following statement:

```
COMMENT ON COLUMN employees.job_id IS '';
```

13 SQL Statements: COMMIT to CREATE JAVA

This chapter contains the following SQL statements:

- COMMIT
- CREATE ANALYTIC VIEW
- CREATE ATTRIBUTE DIMENSION
- CREATE AUDIT POLICY (Unified Auditing)
- CREATE CLUSTER
- CREATE CONTEXT
- CREATE CONTROLFILE
- CREATE DATABASE
- CREATE DATABASE LINK
- CREATE DIMENSION
- CREATE DIRECTORY
- CREATE DISKGROUP
- CREATE EDITION
- CREATE FLASHBACK ARCHIVE
- CREATE FUNCTION
- CREATE HIERARCHY
- CREATE INDEX
- CREATE INDEXTYPE
- CREATE INMEMORY JOIN GROUP
- CREATE JAVA

COMMIT

Purpose

Use the `COMMIT` statement to end your current transaction and make permanent all changes performed in the transaction. A **transaction** is a sequence of SQL statements that Oracle Database treats as a single unit. This statement also erases all savepoints in the transaction and releases transaction locks.

Until you commit a transaction:

- You can see any changes you have made during the transaction by querying the modified tables, but other users cannot see the changes. After you commit the transaction, the changes are visible to other users' statements that execute after the commit.

- You can roll back (undo) any changes made during the transaction with the `ROLLBACK` statement (see ROLLBACK).

Oracle Database issues an implicit `COMMIT` under the following circumstances:

- Before any syntactically valid data definition language (DDL) statement, even if the statement results in an error
- After any data definition language (DDL) statement that completes without an error

You can also use this statement to:

- Commit an in-doubt distributed transaction manually
- Terminate a read-only transaction begun by a `SET TRANSACTION` statement

Oracle recommends that you explicitly end every transaction in your application programs with a `COMMIT` or `ROLLBACK` statement, including the last transaction, before disconnecting from Oracle Database. If you do not explicitly commit the transaction and the program terminates abnormally, then the last uncommitted transaction is automatically rolled back.

A normal exit from most Oracle utilities and tools causes the current transaction to be committed. A normal exit from an Oracle precompiler program does not commit the transaction and relies on Oracle Database to roll back the current transaction.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on transactions
> - SET TRANSACTION for more information on specifying characteristics of a transaction

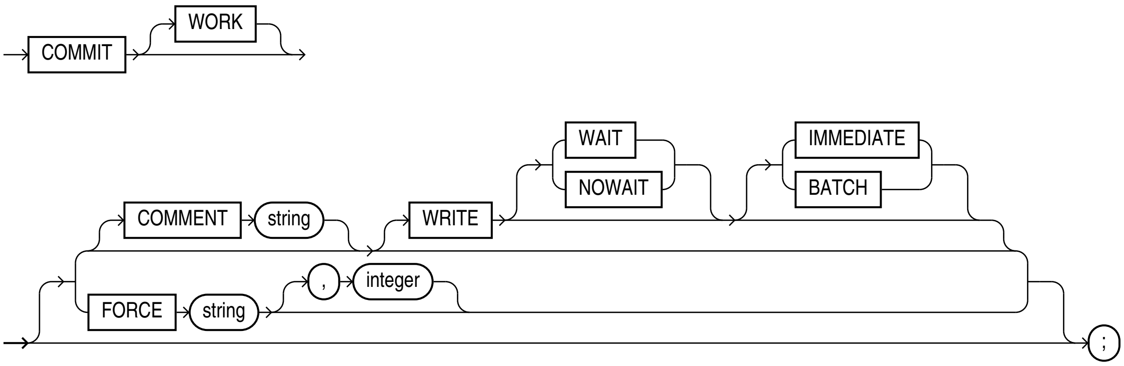
Prerequisites

You need no privileges to commit your current transaction.

To manually commit a distributed in-doubt transaction that you originally committed, you must have `FORCE TRANSACTION` system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have `FORCE ANY TRANSACTION` system privilege.

Syntax

***commit*::=**



Semantics

COMMIT

All clauses after the `COMMIT` keyword are optional. If you specify only `COMMIT`, then the default is `COMMIT WORK WRITE WAIT IMMEDIATE`.

WORK

The `WORK` keyword is supported for compliance with standard SQL. The statements `COMMIT` and `COMMIT WORK` are equivalent.

COMMENT Clause

This clause is supported for backward compatibility. Oracle recommends that you use named transactions instead of commit comments.

> **See Also:**
>
> SET TRANSACTION and *Oracle Database Concepts* for more information on named transactions

Specify a comment to be associated with the current transaction. The '*`text`*' is a quoted literal of up to 255 bytes that Oracle Database stores in the data dictionary view `DBA_2PC_PENDING` along with the transaction ID if a distributed transaction becomes in doubt. This comment can help you diagnose the failure of a distributed transaction.

> **See Also:**
>
> COMMENT for more information on adding comments to SQL statements

WRITE Clause

Use this clause to specify the priority with which the redo information generated by the commit operation is written to the redo log. This clause can improve performance by reducing latency, thus eliminating the wait for an I/O to the redo log. Use this clause to improve response time in environments with stringent response time requirements where the following conditions apply:

- The volume of update transactions is large, requiring that the redo log be written to disk frequently.
- The application can tolerate the loss of an asynchronously committed transaction.
- The latency contributed by waiting for the redo log write to occur contributes significantly to overall response time.

You can specify the `WAIT` | `NOWAIT` and `IMMEDIATE` | `BATCH` clauses in any order.

> **Note:**
>
> If you omit this clause, then the behavior of the commit operation is controlled by the `COMMIT_LOGGING` and `COMMIT_WAIT` initialization parameters, if they have been set.

WAIT | NOWAIT

Use these clauses to specify when control returns to the user.

- The `WAIT` parameter ensures that the commit will return only after the corresponding redo is persistent in the online redo log. Whether in `BATCH` or `IMMEDIATE` mode, when the client receives a successful return from this `COMMIT` statement, the transaction has been committed to durable media. A crash occurring after a successful write to the log can prevent the success message from returning to the client. In this case the client cannot tell whether or not the transaction committed.
- The `NOWAIT` parameter causes the commit to return to the client whether or not the write to the redo log has completed. This behavior can increase transaction throughput. With the `WAIT` parameter, if the commit message is received, then you can be sure that no data has been lost.

> **Note:**
>
> With `NOWAIT`, a crash occurring after the commit message is received, but before the redo log record(s) are written, can falsely indicate to a transaction that its changes are persistent.

If you omit this clause, then the transaction commits with the `WAIT` behavior.

IMMEDIATE | BATCH

Use these clauses to specify when the redo is written to the log.

- The `IMMEDIATE` parameter causes the log writer process (LGWR) to write the transaction's redo information to the log. This operation option forces a disk I/O, so it can reduce transaction throughput.
- The `BATCH` parameter causes the redo to be buffered to the redo log, along with other concurrently executing transactions. When sufficient redo information is collected, a disk write of the redo log is initiated. This behavior is called "group commit", as redo for multiple transactions is written to the log in a single I/O operation.

If you omit this clause, then the transaction commits with the `IMMEDIATE` behavior.

> **See Also:**
>
> *Oracle Database Concepts* for more information on asynchronous commit

FORCE Clause

In a distributed database system, the `FORCE` *`string`* [, *`integer`*] clause lets you manually commit an in-doubt distributed transaction. The transaction is identified by the '*`string`*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view `DBA_2PC_PENDING`. You can use *`integer`* to specifically assign the transaction a system change number (SCN). If you omit *`integer`*, then the transaction is committed using the current SCN.

> **Note:**
>
> A `COMMIT` statement with a `FORCE` clause commits only the specified transactions. Such a statement does not affect your current transaction.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on these topics

Examples

Committing an Insert: Example

This statement inserts a row into the `hr.regions` table and commits this change:

```
INSERT INTO regions VALUES (5, 'Antarctica');

COMMIT WORK;
```

To commit the same insert operation and instruct the database to buffer the change to the redo log, without initiating disk I/O, use the following `COMMIT` statement:

```
COMMIT WRITE BATCH;
```

Commenting on COMMIT: Example

The following statement commits the current transaction and associates a comment with it:

```
COMMIT
    COMMENT 'In-doubt transaction Code 36, Call (415) 555-2637';
```

If a network or machine failure prevents this distributed transaction from committing properly, then Oracle Database stores the comment in the data dictionary along with the transaction ID. The comment indicates the part of the application in which the failure occurred and provides information for contacting the administrator of the database where the transaction was committed.

Forcing an In-Doubt Transaction: Example

The following statement manually commits a hypothetical in-doubt distributed transaction. Query the `V$CORRUPT_XID_LIST` data dictionary view to find the transaction IDs of corrupt transactions. You must have DBA privileges to view the `V$CORRUPT_XID_LIST` and to issue this statement.

```
COMMIT FORCE '22.57.53';
```

CREATE ANALYTIC VIEW

Purpose

Use the `CREATE ANALYTIC VIEW` statement to create an analytic view. An analytic view specifies the source of its fact data and defines measures that describe calculations or other analytic operations to perform on the data. An analytic view also specifies the attribute dimensions and hierarchies that define the rows of the analytic view.

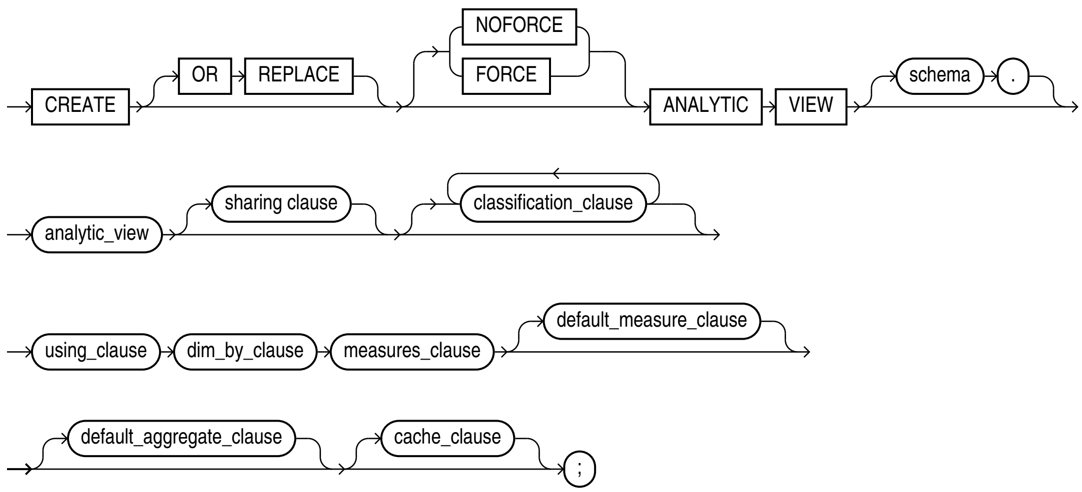
> **Tip:**
>
> You can view and run SQL scripts that create analytic views at the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html. The website has scripts and tutorials that demonstrate the creation and use of analytic views.

Prerequisites

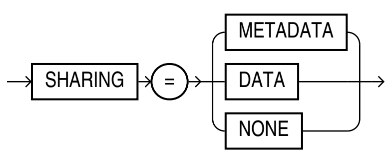
To create an analytic view in your own schema, you must have the `CREATE ANALYTIC VIEW` system privilege. To create an analytic view in another user's schema, you must have the `CREATE ANY ANALYTIC VIEW` system privilege.

Syntax

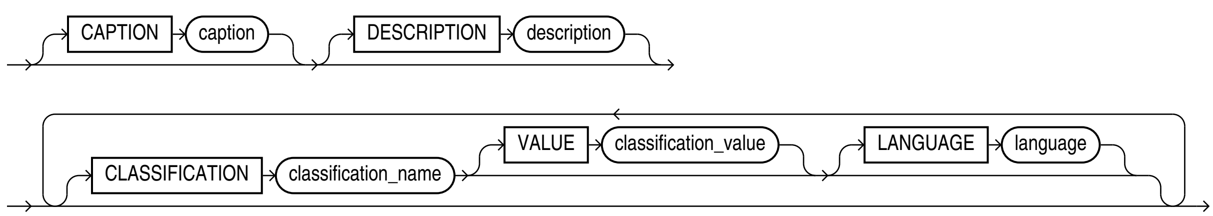
***create_analytic_view*::=**



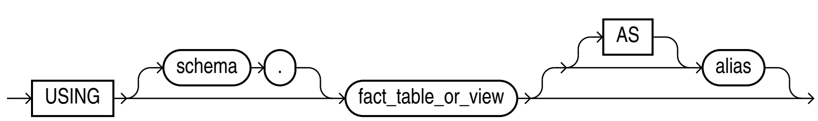
***sharing_clause*::=**



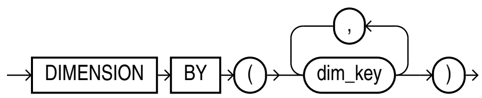
***classification_clause*::=**



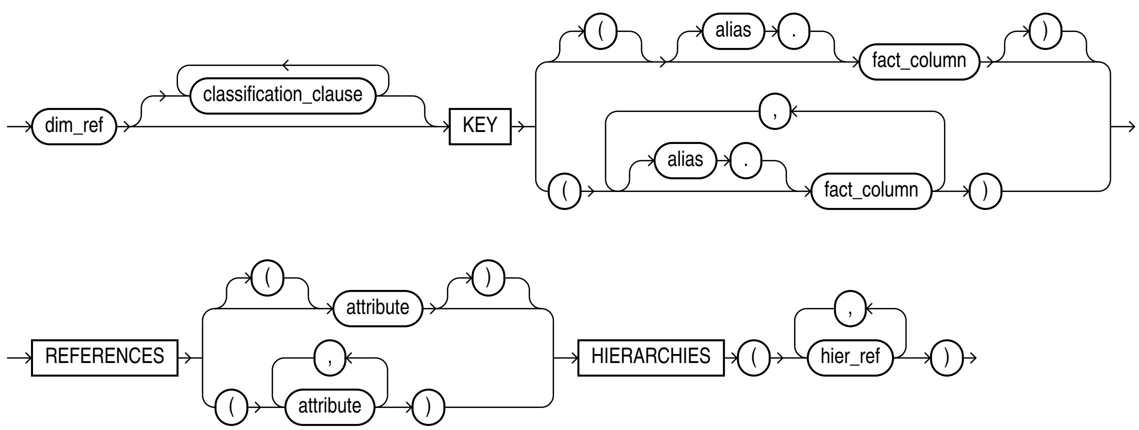
***using_clause*::=**



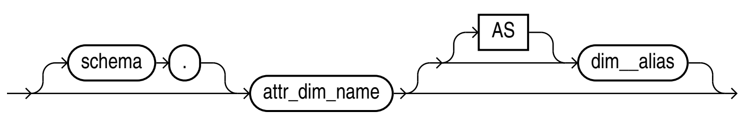
***dim_by_clause*::=**



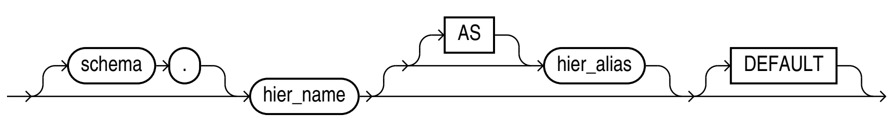
***dim_key*::=**



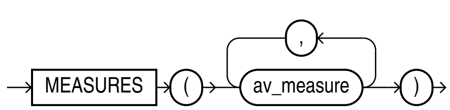
***dim_ref*::=**



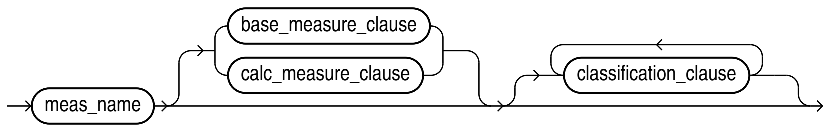
***hier_ref*::=**



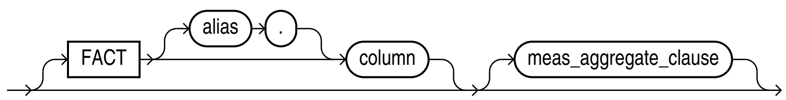
***measures_clause*::=**



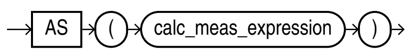
***av_measure*::=**



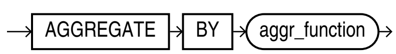
***base_measure_clause*::=**



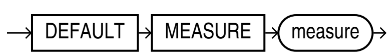
***calc_measure_clause*::=**



***meas_aggregate_clause*::=**

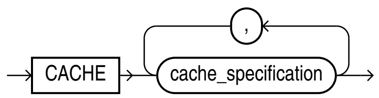


***default_measure_clause*::=**

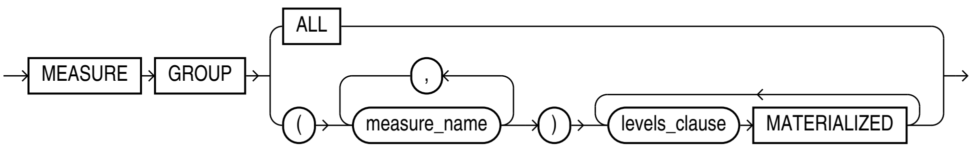


***default_aggregate_clause*::=**

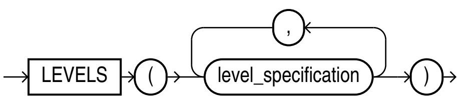
***cache_clause*::=**



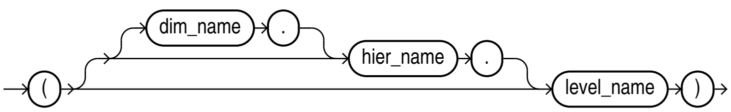
***cache_specification*::=**



***levels_clause*::=**



***level_specification*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing definition of an analytic view with a different definition.

FORCE and NOFORCE

Specify `FORCE` to force the creation of the analytic view even if it does not successfully compile. If you specify `NOFORCE`, then the analytic view must compile successfully, otherwise an error occurs. The default is `NOFORCE`.

schema

Specify the schema in which to create the analytic view. If you do not specify a schema, then Oracle Database creates the analytic view in your own schema.

analytic_view

Specify a name for the analytic view.

sharing_clause

Specify whether to create the analytic view as an application common object. Specifying `METADATA` shares the analytic view's metadata, but its data is unique to each container. Specifying `DATA` shares the analytic view object; its data is the same for all containers in the application container and the data is stored only in the application root. Specifying `NONE` excludes the analytic view from being shared.

classification_clause

Use the classification clause to specify values for the `CAPTION` or `DESCRIPTION` classifications and to specify user-defined classifications. Classifications provide descriptive metadata that applications may use to provide information about analytic views and their components.

You may specify any number of classifications for the same object. A classification can have a maximum length of 4000 bytes.

For the `CAPTION` and `DESCRIPTION` classifications, you may use the DDL shortcuts `CAPTION 'caption'` and `DESCRIPTION 'description'` or the full classification syntax.

You may vary the classification values by language. To specify a language for the `CAPTION` or `DESCRIPTION` classification, you must use the full syntax. If you do not specify a language, then the language value for the classification is `NULL`. The language value must either be `NULL` or a valid `NLS_LANGUAGE` value.

using_clause

Specify a fact table or view. External and remote tables are permitted. You may specify a table or view in another schema. You can specify an alias for the table or view.

dim_by_clause

Specify the attribute dimensions of the analytic view.

dim_key

Specify an attribute dimension, columns of the fact table, columns of the attribute dimension, and hierarchies that are related in the analytic view.

With the `KEY` keyword, specify one or more columns in the fact table.

With the `REFERENCES` keyword, specify attributes of the attribute dimensions that the analytic view is dimensioned by. The attributes must be a level key.

With the `HIERARCHIES` keyword, specify the hierarchies in the analytic view that use the attribute dimension.

dim_ref

Specify an attribute dimension. You can specify an alias for an attribute dimension, which is required if you use the same dimension more than once or if you use multiple dimensions with the same name from different schemas.

hier_ref

Specify a hierarchy. You can specify an alias for a hierarchy. You can specify one of the hierarchies in the list as the default. If you do not specify a default, the first hierarchy in the list is the default.

measures_clause

Specify the measures for the analytic view.

av_measure

Define a measure using either a single fact column or an expression over measures in this analytic view. A measure can be either a base measure or a calculated measure.

base_measure_clause

Define a base measure by optionally specifying a fact column or a *meas_aggregate_clause*, or both. If you do not specify a fact column, then the analytic view uses the column of the fact table that has the same name as the measure. If a column by the same name does not exist, an error is raised.

calc_measure_clause

Define a calculated measure by specifying a calculated measure expression. The expression may reference other measures in the analytic view, but may not reference fact columns. Calculated measures do not have an aggregate clause because they're computed over the aggregated base measures.

For the syntax and descriptions of calculated measure expressions, see Calculated Measure Expressions.

default_measure_clause

Specify a measure to use as the default measure for the analytic view. If you do not specify a measure, the first measure defined is the default.

meas_aggregate_clausè

Specify a default aggregation operator for a base measure. If you do not specify an aggregation operator, then the operator specified by the *default_aggregate_clause* is used.

default_aggregate_clause

Specify a default aggregation for all base measures in the analytic view. If you do not specify a default aggregation, then the default value is `SUM`.

cache_clause

Specify a cache clause to improve query response time when an appropriate materialized view is available. You can specify one or more cache specifications.

cache_specification

Specify one or more measure groups for a cache clause. To include all measure groups, specify `ALL`. Each measure group can contain one or more measures and one or more level groupings. A level grouping can contain one or more level specifications.

level_specification

Specify one or more levels for a level grouping of a measure group for a cache specification. Specify only one level per hierarchy. A materialized view must exist that contains the aggregated values for the hierarchy level.

Examples

The following is a description of the SALES_FACT table:

```
desc SALES_FACT
Name                 Null? Type
------------------ ----- -------------
MONTH_ID                   VARCHAR2(10)
CATEGORY_ID                NUMBER(6)
STATE_PROVINCE_ID          VARCHAR2(120)
UNITS                      NUMBER(6)
SALES                      NUMBER(12,2)
```

The following example creates the SALES_AV analytic view using the SALES_FACT table:

```
CREATE OR REPLACE ANALYTIC VIEW sales_av
USING sales_fact
DIMENSION BY
  (time_attr_dim                        -- An attribute dimension of time
data
    KEY month_id REFERENCES month_id
    HIERARCHIES (
      time_hier DEFAULT),
   product_attr_dim                     -- An attribute dimension of
product data
    KEY category_id REFERENCES category_id
    HIERARCHIES (
      product_hier DEFAULT),
   geography_attr_dim                   -- An attribute dimension of store
data
    KEY state_province_id
    REFERENCES state_province_id HIERARCHIES (
      geography_hier DEFAULT)
   )
MEASURES
 (sales FACT sales,                     -- A base measure
  units FACT units,                     -- A base measure
  sales_prior_period AS                 -- Calculated measures
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1)),
  sales_year_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  chg_sales_year_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  pct_chg_sales_year_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL year)),
  sales_qtr_ago AS
    (LAG(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter)),
  chg_sales_qtr_ago AS
    (LAG_DIFF(sales) OVER (HIERARCHY time_hier OFFSET 1
```

```
     ACROSS ANCESTOR AT LEVEL quarter)),
  pct_chg_sales_qtr_ago AS
    (LAG_DIFF_PERCENT(sales) OVER (HIERARCHY time_hier OFFSET 1
     ACROSS ANCESTOR AT LEVEL quarter))
  )
DEFAULT MEASURE SALES;
```

CREATE ATTRIBUTE DIMENSION

Purpose

Use the `CREATE ATTRIBUTE DIMENSION` statement to create an attribute dimension. An attribute dimension specifies dimension members for one or more analytic view hierarchies. It specifies the data source it is using and the members it includes. It specifies levels for its members and determines attribute relationships between levels.

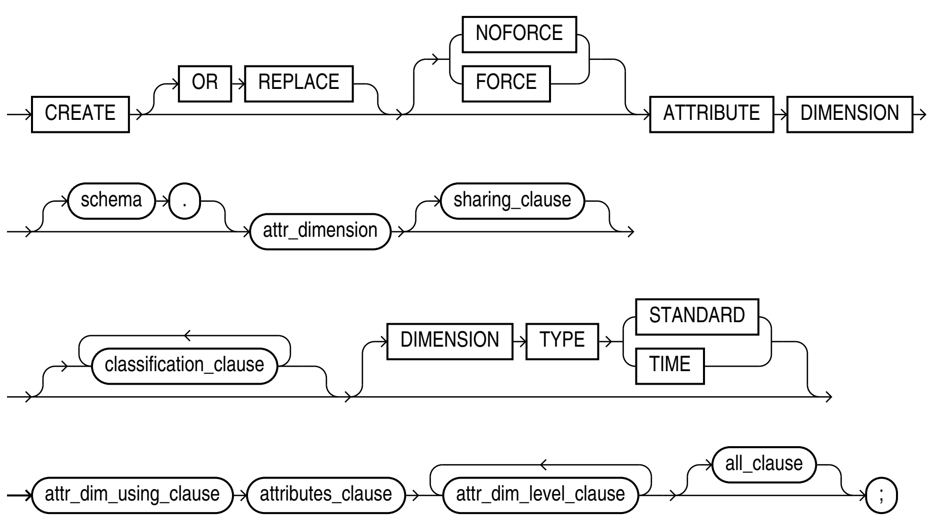
> **Tip:**
>
> You can view and run SQL scripts that create attribute dimensions at the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html. The website has scripts and tutorials that demonstrate the creation and use of analytic views.

Prerequisites

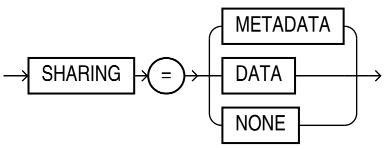
To create an attribute dimension in your own schema, you must have the `CREATE ATTRIBUTE DIMENSION` system privilege. To create an attribute dimension in another user's schema, you must have the `CREATE ANY ATTRIBUTE DIMENSION` system privilege.

Syntax

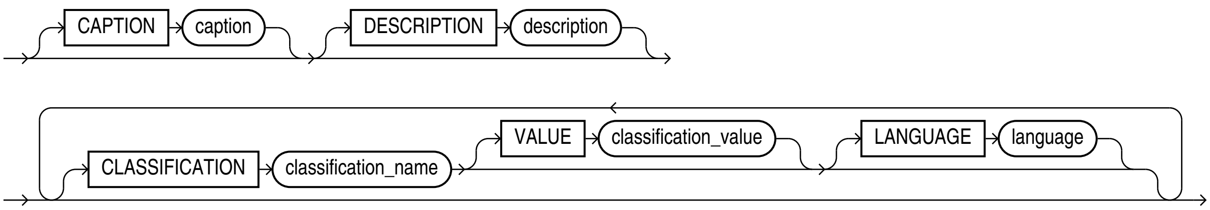
***create_attribute_dimension*::=**



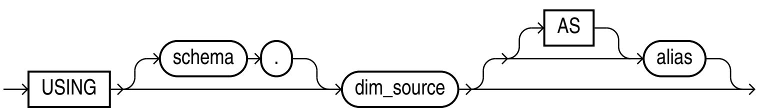
***sharing_clause*::=**



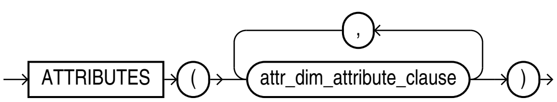
***classification_clause*::=**



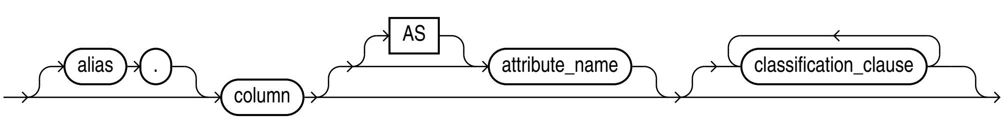
***attr_dim_using_clause*::=**



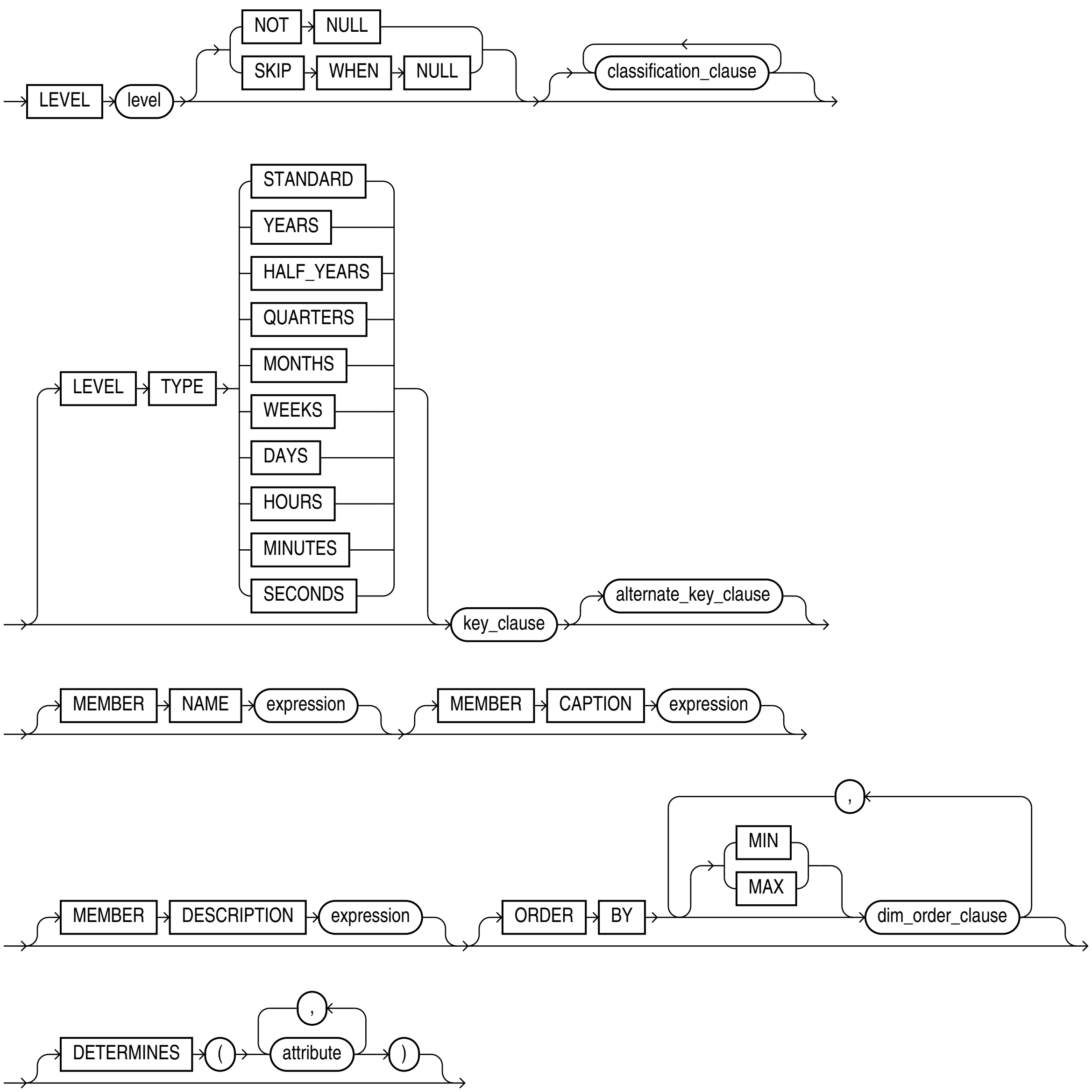
***attributes_clause*::=**



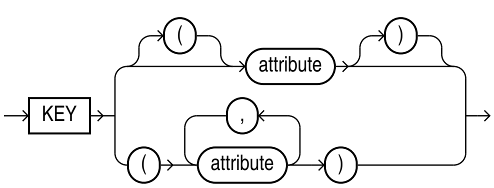
***attr_dim_attributes_clause*::=**



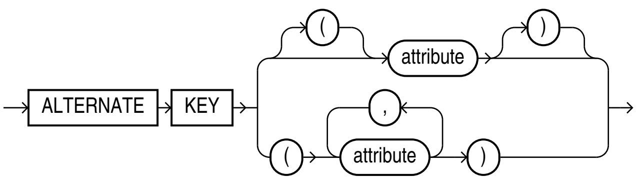
***attr_dim_level_clause*::=**



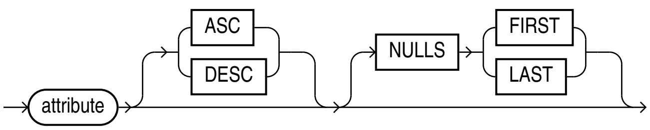
***key_clause*::=**



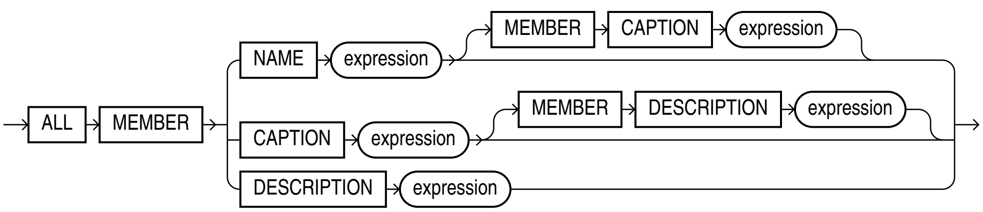
***alternate_key_clause*::=**



***dim_order_clause*::=**



***all_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing definition of an attribute dimension with a different definition.

FORCE and NOFORCE

Specify `FORCE` to force the creation of the attribute dimension even if it does not successfully compile. If you specify `NOFORCE`, then the attribute dimension must compile successfully, otherwise an error occurs. The default is `NOFORCE`.

schema

Specify the schema in which to create the attribute dimension. If you do not specify a schema, then Oracle Database creates the attribute dimension in your own schema.

attr_dimension

Specify a name for the attribute dimension.

sharing_clause

Specify whether to create the attribute dimension as an application common object. Specifying `METADATA` shares the attribute dimension's metadata, but its data is unique to each container. Specifying `DATA` shares the attribute dimension object; its data is the same for all containers in the application container and the data is stored only in the application root. Specifying `NONE` excludes the attribute dimension from being shared.

classification_clause

Use the classification clause to specify values for the `CAPTION` or `DESCRIPTION` classifications and to specify user-defined classifications. Classifications provide descriptive metadata that applications may use to provide information about analytic views and their components.

You may specify any number of classifications for the same object. A classification can have a maximum length of 4000 bytes.

For the `CAPTION` and `DESCRIPTION` classifications, you may use the DDL shortcuts `CAPTION '`*`caption`*`'` and `DESCRIPTION '`*`description`*`'` or the full classification syntax.

You may vary the classification values by language. To specify a language for the `CAPTION` or `DESCRIPTION` classification, you must use the full syntax. If you do not specify a language, then the language value for the classification is `NULL`. The language value must either be `NULL` or a valid `NLS_LANGUAGE` value.

DIMENSION TYPE

An attribute dimension may be either a `STANDARD` or a `TIME` type. A `STANDARD` type attribute dimension has `STANDARD` type levels. Each level of a `TIME` type attribute dimension is one of the time types. The default `DIMENSION TYPE` is `STANDARD`.

attr_dim_using_clause

Specify a table or view. You may specify an alias for the table or view by using the `AS` keyword.

attributes_clause

Specify one or more *`attr_dim_attribute_clause`* clauses.

attr_dim_attribute_clause

Specify a column from the *`attr_dim_using_clause`* source. The attribute has the name of the column unless you specify an alias using the `AS` keyword. You may specify classifications for each attribute.

attr_dim_level_clause

Specify a level in the attribute dimension. A level specifies key and optional alternate key attributes that provide the members of the level.

If the key attribute has no `NULL` values, then you may specify `NOT NULL`, which is the default. If it does have one or more `NULL` values, then specify `SKIP WHEN NULL`.

LEVEL TYPE

A `STANDARD` type attribute dimension has `STANDARD` type levels. You do not need to specify a `LEVEL TYPE` for a `STANDARD` type attribute dimension.

In a `TIME` type attribute dimension, you must specify a level type. The type of the level may be one of the time types. You must specify a time type even if the values of the level members are not of that type. For example, you may have a SEASON level with values that are the names of seasons. In defining the level, you must specify any one of the time level types, such as QUARTERS. An application may use the level type designations for whatever purpose it chooses.

DETERMINES

With the DETERMINES keyword, you may specify other attributes of the attribute dimension that this level determines. If an attribute has only one value for each value of another attribute, then the value of the first attribute determines the value of the other attribute. For example, the QUARTER_ID attribute has only one value for each value of the MONTH_ID attribute, so you can include the the QUARTER_ID attribute in the DETERMINES phrase of the MONTHS level.

key_clause

Specify one or more attributes as the key for the level.

alternate_key_clause

Specify one or more attributes as the alternate key for the level.

dim_order_clause

Specify the ordering of the members of the level.

all_clause

Optionally specify MEMBER NAME, MEMBER CAPTION, and MEMBER DESCRIPTION values for the implicit ALL level. By default, the MEMBER NAME value is ALL.

Examples

The following example describes the TIME_DIM table:

```
desc TIME_DIM

Name                  Null?    Type
------------------ -----    -------------
MONTH_ID                       VARCHAR2(10)
CATEGORY_ID                    NUMBER(6)
STATE_PROVINCE_ID              VARCHAR2(120)
UNITS                          NUMBER(6)
SALES                          NUMBER(12,2)
YEAR_ID           NOT NULL VARCHAR2(30)
YEAR_NAME         NOT NULL VARCHAR2(40)
YEAR_END_DATE                  DATE
QUARTER_ID        NOT NULL VARCHAR2(30)
QUARTER_NAME      NOT NULL VARCHAR2(40)
QUARTER_END_DATE               DATE
QUARTER_OF_YEAR                NUMBER
MONTH_ID          NOT NULL VARCHAR2(30)
MONTH_NAME        NOT NULL VARCHAR2(40)
MONTH_END_DATE                 DATE
MONTH_OF_YEAR                  NUMBER
MONTH_LONG_NAME                VARCHAR2(30)
SEASON                         VARCHAR2(10)
```

```
SEASON_ORDER                NUMBER(38)
MONTH_OF_QUARTER            NUMBER(38)
```

The following example creates a `TIME` type attribute dimension, using columns from the TIME_DIM table:

```
CREATE OR REPLACE ATTRIBUTE DIMENSION time_attr_dim
DIMENSION TYPE TIME
USING time_dim
ATTRIBUTES
 (year_id
   CLASSIFICATION caption VALUE 'YEAR_ID'
   CLASSIFICATION description VALUE 'YEAR ID',
  year_name
    CLASSIFICATION caption VALUE 'YEAR_NAME'
    CLASSIFICATION description VALUE 'Year',
  year_end_date
    CLASSIFICATION caption VALUE 'YEAR_END_DATE'
    CLASSIFICATION description VALUE 'Year End Date',
  quarter_id
    CLASSIFICATION caption VALUE 'QUARTER_ID'
    CLASSIFICATION description VALUE 'QUARTER ID',
  quarter_name
    CLASSIFICATION caption VALUE 'QUARTER_NAME'
    CLASSIFICATION description VALUE 'Quarter',
  quarter_end_date
    CLASSIFICATION caption VALUE 'QUARTER_END_DATE'
    CLASSIFICATION description VALUE 'Quarter End Date',
  quarter_of_year
    CLASSIFICATION caption VALUE 'QUARTER_OF_YEAR'
    CLASSIFICATION description VALUE 'Quarter of Year',
  month_id
    CLASSIFICATION caption VALUE 'MONTH_ID'
    CLASSIFICATION description VALUE 'MONTH ID',
  month_name
    CLASSIFICATION caption VALUE 'MONTH_NAME'
    CLASSIFICATION description VALUE 'Month',
  month_long_name
    CLASSIFICATION caption VALUE 'MONTH_LONG_NAME'
    CLASSIFICATION description VALUE 'Month Long Name',
  month_end_date
    CLASSIFICATION caption VALUE 'MONTH_END_DATE'
    CLASSIFICATION description VALUE 'Month End Date',
  month_of_quarter
    CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
    CLASSIFICATION description VALUE 'Month of Quarter',
  month_of_year
    CLASSIFICATION caption VALUE 'MONTH_OF_YEAR'
    CLASSIFICATION description VALUE 'Month of Year',
  season
    CLASSIFICATION caption VALUE 'SEASON'
    CLASSIFICATION description VALUE 'Season',
  season_order
    CLASSIFICATION caption VALUE 'SEASON_ORDER'
```

```
    CLASSIFICATION description VALUE 'Season Order')
LEVEL month
  LEVEL TYPE MONTHS
  CLASSIFICATION caption VALUE 'MONTH'
  CLASSIFICATION description VALUE 'Month'
  KEY month_id
  MEMBER NAME month_name
  MEMBER CAPTION month_name
  MEMBER DESCRIPTION month_long_name
  ORDER BY month_end_date
  DETERMINES (month_end_date,
    quarter_id,
    season,
    season_order,
    month_of_year,
    month_of_quarter)
LEVEL quarter
  LEVEL TYPE QUARTERS
  CLASSIFICATION caption VALUE 'QUARTER'
  CLASSIFICATION description VALUE 'Quarter'
  KEY quarter_id
  MEMBER NAME quarter_name
  MEMBER CAPTION quarter_name
  MEMBER DESCRIPTION quarter_name
  ORDER BY quarter_end_date
  DETERMINES (quarter_end_date,
    quarter_of_year,
    year_id)
LEVEL year
  LEVEL TYPE YEARS
  CLASSIFICATION caption VALUE 'YEAR'
  CLASSIFICATION description VALUE 'Year'
  KEY year_id
  MEMBER NAME year_name
  MEMBER CAPTION year_name
  MEMBER DESCRIPTION year_name
  ORDER BY year_end_date
  DETERMINES (year_end_date)
LEVEL season
  LEVEL TYPE QUARTERS
  CLASSIFICATION caption VALUE 'SEASON'
  CLASSIFICATION description VALUE 'Season'
  KEY season
  MEMBER NAME season
  MEMBER CAPTION season
  MEMBER DESCRIPTION season
LEVEL month_of_quarter
  LEVEL TYPE MONTHS
  CLASSIFICATION caption VALUE 'MONTH_OF_QUARTER'
  CLASSIFICATION description VALUE 'Month of Quarter'
  KEY month_of_quarter;
```

The following example describes the PRODUCT_DIM table:

```
desc PRODUCT_DIM

Name            Null?    Type
--------------- -------- -------------
DEPARTMENT_ID   NOT NULL NUMBER
DEPARTMENT_NAME NOT NULL VARCHAR2(100)
CATEGORY_ID     NOT NULL NUMBER
CATEGORY_NAME   NOT NULL VARCHAR2(100)
```

The following example creates a `STANDARD` type attribute dimension, using columns from the PRODUCT_DIM table:

```
CREATE OR REPLACE ATTRIBUTE DIMENSION product_attr_dim
USING product_dim
ATTRIBUTES
 (department_id,
  department_name,
  category_id,
  category_name)
LEVEL DEPARTMENT
  KEY department_id
  ALTERNATE KEY department_name
  MEMBER NAME department_name
  MEMBER CAPTION department_name
  ORDER BY department_name
LEVEL CATEGORY
  KEY category_id
  ALTERNATE KEY category_name
  MEMBER NAME category_name
  MEMBER CAPTION category_name
  ORDER BY category_name
  DETERMINES(department_id)
ALL MEMBER NAME 'ALL PRODUCTS';
```

The following example describes the GEOGRAPHY_DIM table:

```
desc GEOGRAPHY_DIM

Name                Null?    Type
----------------    -------- -------------
DEPARTMENT_ID       NOT NULL NUMBER
DEPARTMENT_NAME     NOT NULL VARCHAR2(100)
CATEGORY_ID         NOT NULL NUMBER
CATEGORY_NAME       NOT NULL VARCHAR2(100)
REGION_ID           NOT NULL VARCHAR2(120)
REGION_NAME         NOT NULL VARCHAR2(100)
COUNTRY_ID          NOT NULL VARCHAR2(2)
COUNTRY_NAME        NOT NULL VARCHAR2(120)
STATE_PROVINCE_ID   NOT NULL VARCHAR2(120)
STATE_PROVINCE_NAME NOT NULL VARCHAR2(400)
```

The following example creates an STANDARD type attribute dimension, using columns from the GEOGRAPHY_DIM table:

```
CREATE OR REPLACE ATTRIBUTE DIMENSION geography_attr_dim
USING geography_dim
ATTRIBUTES
 (region_id,
  region_name,
  country_id,
  country_name,
  state_province_id,
  state_province_name)
LEVEL REGION
  KEY region_id
  ALTERNATE KEY region_name
  MEMBER NAME region_name
  MEMBER CAPTION region_name
  ORDER BY region_name
LEVEL COUNTRY
  KEY country_id
  ALTERNATE KEY country_name
  MEMBER NAME country_name
  MEMBER CAPTION country_name
  ORDER BY country_name
  DETERMINES(region_id)
LEVEL STATE_PROVINCE
  KEY state_province_id
  ALTERNATE KEY state_province_name
  MEMBER NAME state_province_name
  MEMBER CAPTION state_province_name
  ORDER BY state_province_name
  DETERMINES(country_id)
ALL MEMBER NAME 'ALL CUSTOMERS';
```

CREATE AUDIT POLICY (Unified Auditing)

This section describes the CREATE AUDIT POLICY statement for **unified auditing**. This type of auditing is new beginning with Oracle Database 12*c* and provides a full set of enhanced auditing features. Refer to *Oracle Database Security Guide* for more information on unified auditing.

Purpose

Use the CREATE AUDIT POLICY statement to create a unified audit policy.

See Also:

- ALTER AUDIT POLICY (Unified Auditing)
- DROP AUDIT POLICY (Unified Auditing)
- AUDIT (Unified Auditing)
- NOAUDIT (Unified Auditing)

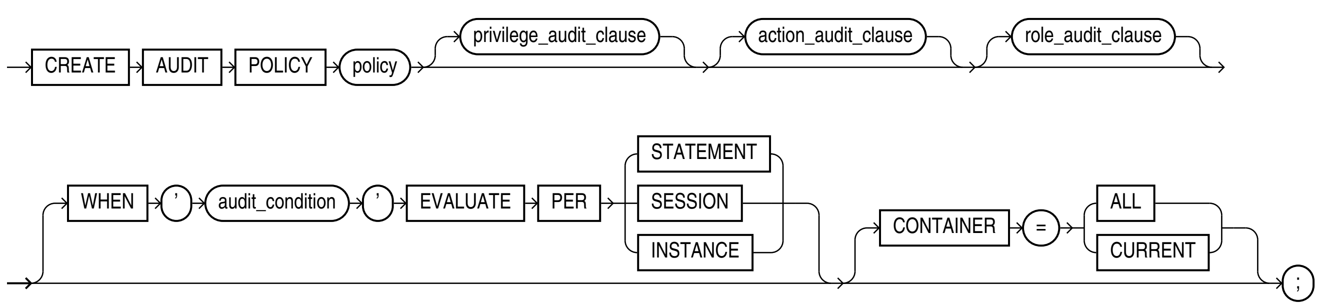
Prerequisites

You must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To create a common unified audit policy, you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role. To create a local unified audit policy, you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the container to which you are connected.

Syntax

***create_audit_policy*::=**

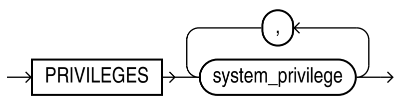


Note:

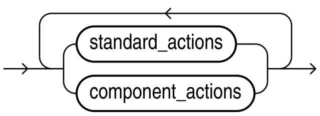
You must specify at least one of the clauses *`privilege_audit_clause`*, *`action_audit_clause`*, or *`role_audit_clause`*.

(*privilege_audit_clause*::=, *action_audit_clause*::=, *role_audit_clause*::=)

***privilege_audit_clause*::=**

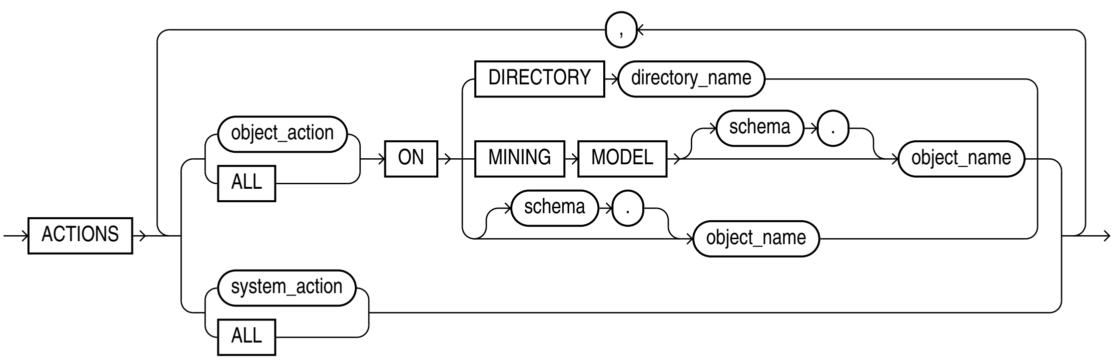


***action_audit_clause*::=**

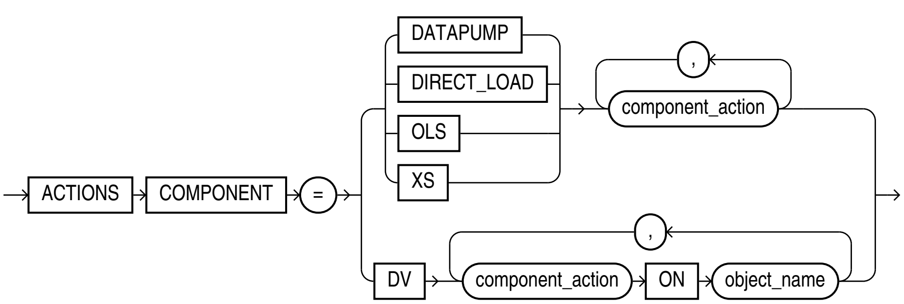


> **Note:**
>
> You can specify only the `standard_actions` clause, only the `component_actions` clause, or both clauses in either order, but you can specify each clause at most once.

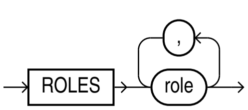
***standard_actions*::=**



***component_actions*::=**



***role_audit_clause*::=**



Semantics

policy

Specify the name of the unified audit policy to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

You can find the names of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES` view

privilege_audit_clause

Use this clause to audit one or more system privileges. For *`system_privilege`*, specify a valid system privilege. To view all valid system privileges, query the `NAME` column of the `SYSTEM_PRIVILEGE_MAP` view.

Only those SQL statements are audited, that make successful use of system privileges. If a statement does not make use of a system privilege, it does not get audited with the *`privilege_audit_clause`*.

Restriction on Auditing System Privileges

You cannot audit the following system privileges: `INHERIT ANY PRIVILEGES`, `SYSASM`, `SYSBACKUP`, `SYSDBA`, `SYSDG`, `SYSKM`, and `SYSOPER`.

action_audit_clause

Use this clause to specify one or more actions to be audited. Use the *`standard_actions`* clause to audit actions on standard RDBMS objects and to audit standard RDBMS system actions for the database. Use the *`component_actions`* clause to audit actions for components.

standard_actions

Use this clause to audit actions on standard RDBMS objects and to audit standard RDBMS system actions for the database.

_object_action_ ON

Use this clause to audit an action on the specified object. For *`object_action`*, specify the action to be audited. Table 13-1 lists the actions that can be audited on each type of object.

ALL ON

Use this clause to audit all actions on the specified object. All of the actions listed in Table 13-1 for the type of object that you specify in the `ON` clause will be audited.

ON Clause

Use the ON clause to specify the object to be audited. Directories and data mining models are identified separately because they reside in separate namespaces. To audit actions on a directory, specify ON DIRECTORY *directory_name*. To audit actions on a data mining model, specify ON MINING MODEL *object_name*. To audit actions on the other types of objects listed in Table 13-1, specify ON *object_name*. If you do not qualify *object_name* with *schema*, then the database assumes the object is in your own schema.

Table 13-1 Unified Auditing Objects and Actions

Type of Object	Actions
Directory	AUDIT, GRANT, READ
Function	AUDIT, EXECUTE (Notes 1 and 2), GRANT
Java Schema Objects (Source, Class, Resource)	AUDIT, EXECUTE, GRANT
Library	EXECUTE, GRANT
Materialized Views	ALTER, AUDIT, COMMENT, DELETE, INDEX, INSERT, LOCK, SELECT, UPDATE
Mining Model	AUDIT, COMMENT, GRANT, RENAME, SELECT
Object Type	ALTER, AUDIT, GRANT
Package	AUDIT, EXECUTE, GRANT
Procedure	AUDIT, EXECUTE (Notes 1 and 2), GRANT
Sequence	ALTER, AUDIT, GRANT, SELECT
Table	ALTER, AUDIT, COMMENT, DELETE, FLASHBACK, GRANT, INDEX, INSERT, LOCK, RENAME, SELECT, UPDATE
View	AUDIT, DELETE, FLASHBACK, GRANT, INSERT, LOCK, RENAME, SELECT, UPDATE

Note 1: When you audit the EXECUTE operation on a PL/SQL stored procedure or stored function, the database considers only its ability to find the procedure or function and authorize its execution when determining the success or failure of the operation for the purposes of auditing. Therefore, if you specify the WHENEVER NOT SUCCESSFUL clause, then only invalid object errors, non-existent object errors, and authorization failures are audited; errors encountered during the execution of the procedure or function are not audited. If you specify the WHENEVER SUCCESSFUL clause, then all executions that are not blocked by invalid object errors, non-existent object errors, or authorization failures are audited, regardless of whether errors are encountered during execution.

Note 2: To audit the failure of a recursive SQL operation inside a PL/SQL stored procedure or stored function, configure auditing for the SQL operation.

system_action

Use this clause to audit a system action for the database. To view the valid values for *system_action*, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is 'Standard'.

ALL

Use this clause to audit all system actions for the database.

component_actions

Use this clause to audit actions for the following components: Oracle Data Pump, Oracle SQL*Loader Direct Path Load, Oracle Label Security, Oracle Database Real Application Security, and Oracle Database Vault.

DATAPUMP

Use this clause to audit actions for Oracle Data Pump. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Data Pump, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Datapump. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Datapump';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Data Pump.

DIRECT_LOAD

Use this clause to audit actions for Oracle SQL*Loader Direct Path Load. For *component_action*, specify the action to be audited. To view the valid actions for Oracle SQL*Loader Direct Path Load, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Direct path API. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Direct path API';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle SQL*Loader Direct Path Load.

OLS

Use this clause to audit actions for Oracle Label Security. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Label Security, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is Label Security. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Label Security';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Label Security.

XS

Use this clause to audit actions for Oracle Database Real Application Security. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Database Real Application Security, query the NAME column of the AUDITABLE_SYSTEM_ACTIONS view where COMPONENT is XS. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'XS';
```

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Database Real Application Security.

DV

Use this clause to audit actions for Oracle Database Vault. For *component_action*, specify the action to be audited. To view the valid actions for Oracle Database Vault, query the `NAME` column of the `AUDITABLE_SYSTEM_ACTIONS` view where `COMPONENT` is `Database Vault`. For example:

```
SELECT name FROM auditable_system_actions WHERE component = 'Database Vault';
```

For *`object_name`*, specify the name of the Database Vault object to be audited.

Refer to *Oracle Database Security Guide* for complete information on auditing Oracle Database Vault.

role_audit_clause

Use this clause to specify one or more roles to be audited. When you audit a role, Oracle Database audits all system privileges that are granted directly to the role. SQL statements that require the system privileges in order to succeed are audited. For *`role`*, specify either a user-defined (local or external) or predefined role. For a list of predefined roles, refer to *Oracle Database Security Guide*.

WHEN Clause

Use this clause to control when the unified audit policy is enforced.

audit_condition

Specify a condition that determines if the unified audit policy is enforced. If *`audit_condition`* evaluates to `TRUE`, then the policy is enforced. If `FALSE`, then the policy is not enforced.

The *`audit_condition`* can have a maximum length of 4000 characters. It can contain expressions, as well as the following functions and conditions:

- Numeric functions: `BITAND`, `CEIL`, `FLOOR`, `POWER`
- Character functions returning character values: `CONCAT`, `LOWER`, `UPPER`
- Character functions returning number values: `INSTR`, `LENGTH`
- Environment and identifier functions: `SYS_CONTEXT`, `UID`
- Comparison conditions: `=`, `!=`, `<>`, `<`, `>`, `<=`, `>=`
- Logical conditions: `AND`, `OR`
- Null conditions: `IS [NOT] NULL`
- `[NOT] BETWEEN` condition
- `[NOT] IN` condition

The *`audit_condition`* must be enclosed in single quotation marks. If the *`audit_condition`* contains a single quotation mark, then specify two single quotation marks instead. For example, to specify the following condition:

```
SYS_CONTEXT('USERENV', 'CLIENT_IDENTIFIER') = 'myclient'
```

Specify the following for '*`audit_condition`*':

```
'SYS_CONTEXT(''USERENV'', ''CLIENT_IDENTIFIER'') = ''myclient'''
```

EVALUATE PER STATEMENT

Specify this clause to evaluate the *audit_condition* for each auditable statement. If the *audit_condition* evaluates to `TRUE`, then the unified audit policy is enforced for the statement. If `FALSE`, then the unified audit policy is not enforced for the statement.

EVALUATE PER SESSION

Specify this clause to evaluate the *audit_condition* once during the session. The *audit_condition* is evaluated for the first auditable statement that is executed during the session. If the *audit_condition* evaluates to `TRUE`, then the unified audit policy is enforced for all applicable statements for the rest of the session. If `FALSE`, then the unified audit policy is not enforced for all applicable statements for the rest of the session.

EVALUATE PER INSTANCE

Specify this clause to evaluate the *audit_condition* once during the lifetime of the instance. The *audit_condition* is evaluated for the first auditable statement that is executed during the instance lifetime. If the *audit_condition* evaluates to `TRUE`, then the unified audit policy is enforced for all applicable statements for the rest of the lifetime of the instance. If `FALSE`, then the unified audit policy is not enforced for all applicable statements for the rest of the lifetime of the instance.

CONTAINER Clause

Use the `CONTAINER` clause to specify the scope of the unified audit policy.

- Specify `CONTAINER = ALL` to create a **common unified audit policy**. This type of policy is available to all pluggable databases (PDBs) in the CDB. The current container must be the root. If you specify the `ACTIONS` *object_action* `ON` or `ACTIONS ALL ON` clause, then you must specify a common object or an application common object.
- Specify `CONTAINER = CURRENT` to create a **local unified audit policy** in the current container. The current container can be the root or a PDB.

If you omit this clause, then `CONTAINER = CURRENT` is the default.

> **Note:**
>
> You cannot alter the scope of a unified audit policy after it has been created.

Examples

Auditing System Privileges: Example

The following statement creates unified audit policy `table_pol`, which audits the system privileges `CREATE ANY TABLE` and `DROP ANY TABLE`:

```
CREATE AUDIT POLICY table_pol
  PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE;
```

The following statement verifies that `table_pol` now appears in the `AUDIT_UNIFIED_POLICIES` view:

```
SELECT *
  FROM audit_unified_policies
  WHERE policy_name = 'TABLE_POL';
```

Auditing Actions on Objects: Examples

The following statement creates unified audit policy `dml_pol`, which audits `DELETE`, `INSERT`, and `UPDATE` actions on table `hr.employees`, and all auditable actions on table `hr.departments`:

```
CREATE AUDIT POLICY dml_pol
  ACTIONS DELETE on hr.employees,
          INSERT on hr.employees,
          UPDATE on hr.employees,
          ALL on hr.departments;
```

The following statement creates unified audit policy `read_dir_pol`, which audits `READ` actions on directory `bfile_dir` (created in "Creating a Directory: Examples"):

```
CREATE AUDIT POLICY read_dir_pol
  ACTIONS READ ON DIRECTORY bfile_dir;
```

Auditing System Actions: Examples

The following query displays the standard RDBMS system actions that can be audited for the database:

```
SELECT name FROM auditable_system_actions
  WHERE component = 'Standard'
  ORDER BY name;

NAME
----
ADMINISTER KEY MANAGEMENT
ALL
ALTER ASSEMBLY
ALTER AUDIT POLICY
ALTER CLUSTER
...
```

The following statement creates unified audit policy `security_pol`, which audits the system action `ADMINISTER KEY MANAGEMENT`:

```
CREATE AUDIT POLICY security_pol
  ACTIONS ADMINISTER KEY MANAGEMENT;
```

The following statement creates unified audit policy `dir_pol`, which audits all read, write, and execute operations on any directory:

```
CREATE AUDIT POLICY dir_pol
  ACTIONS READ DIRECTORY, WRITE DIRECTORY, EXECUTE DIRECTORY;
```

The following statement creates unified audit policy `all_actions_pol`, which audits all standard RDBMS system actions for the database:

```
CREATE AUDIT POLICY all_actions_pol
  ACTIONS ALL;
```

Auditing Component Actions: Example

The following query displays the actions that can be audited for Oracle Data Pump:

```
SELECT name FROM auditable_system_actions
  WHERE component = 'Datapump';

NAME
----
EXPORT
IMPORT
ALL
```

The following statement creates unified audit policy `dp_actions_pol`, which audits `IMPORT` actions for Oracle Data Pump:

```
CREATE AUDIT POLICY dp_actions_pol
  ACTIONS COMPONENT = datapump IMPORT;
```

Auditing Roles: Example

The following statement creates unified audit policy `java_pol`, which audits the predefined roles `java_admin` and `java_deploy`:

```
CREATE AUDIT POLICY java_pol
  ROLES java_admin, java_deploy;
```

Auditing System Privileges, Actions, and Roles: Example

The following statement creates unified audit policy `hr_admin_pol`, which audits multiple system privileges, actions, and roles:

```
CREATE AUDIT POLICY hr_admin_pol
  PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE
  ACTIONS DELETE on hr.employees,
          INSERT on hr.employees,
          UPDATE on hr.employees,
          ALL on hr.departments,
          LOCK TABLE
  ROLES audit_admin, audit_viewer;
```

Controlling When a Unified Audit Policy is Enforced: Examples

The following statement creates unified audit policy `order_updates_pol`, which audits `UPDATE` actions on table `oe.orders`. This policy is enforced only when the auditable statement is issued by an external user. The audit condition is checked once per session.

```
CREATE AUDIT POLICY order_updates_pol
  ACTIONS UPDATE ON oe.orders
  WHEN 'SYS_CONTEXT(''USERENV'', ''IDENTIFICATION_TYPE'') = ''EXTERNAL'''
  EVALUATE PER SESSION;
```

The following statement creates unified audit policy `emp_updates_pol`, which audits `DELETE`, `INSERT`, and `UPDATE` actions on table `hr.employees`. This policy is enforced only when the auditable statement is issued by a user who does not have a UID of 100, 105, or 107. The audit condition is checked for each auditable statement.

```
CREATE AUDIT POLICY emp_updates_pol
  ACTIONS DELETE on hr.employees,
          INSERT on hr.employees,
          UPDATE on hr.employees
  WHEN 'UID NOT IN (100, 105, 107)'
  EVALUATE PER STATEMENT;
```

Creating a Local Unified Audit Policy: Example

The following statement creates local unified audit policy `local_table_pol`, which audits the system privileges `CREATE ANY TABLE` and `DROP ANY TABLE` in the current container::

```
CREATE AUDIT POLICY local_table_pol
  PRIVILEGES CREATE ANY TABLE, DROP ANY TABLE
  CONTAINER = CURRENT;
```

Creating a Common Unified Audit Policy: Example

The following statement creates common unified audit policy `common_role1_pol`, which audits the common role `c##role1` (created in `CREATE ROLE` "Examples") across the entire CDB:

```
CREATE AUDIT POLICY common_role1_pol
  ROLES c##role1
  CONTAINER = ALL;
```

CREATE CLUSTER

Purpose

Use the `CREATE CLUSTER` statement to create a cluster. A **cluster** is a schema object that contains data from one or more tables.

- An **indexed cluster** must contain more than one table, and all of the tables in the cluster have one or more columns in common. Oracle Database stores together all the rows from all the tables that share the same cluster key.
- In a **hash cluster**, which can contain one or more tables, Oracle Database stores together rows that have the same hash key value.

For information on existing clusters, query the `USER_CLUSTERS`, `ALL_CLUSTERS`, and `DBA_CLUSTERS` data dictionary views.

> **See Also:**
>
> - *Oracle Database Concepts* for general information on clusters
> - *Oracle Database SQL Tuning Guide* for suggestions on when to use clusters
> - *Oracle Database Reference* for information on the data dictionary views

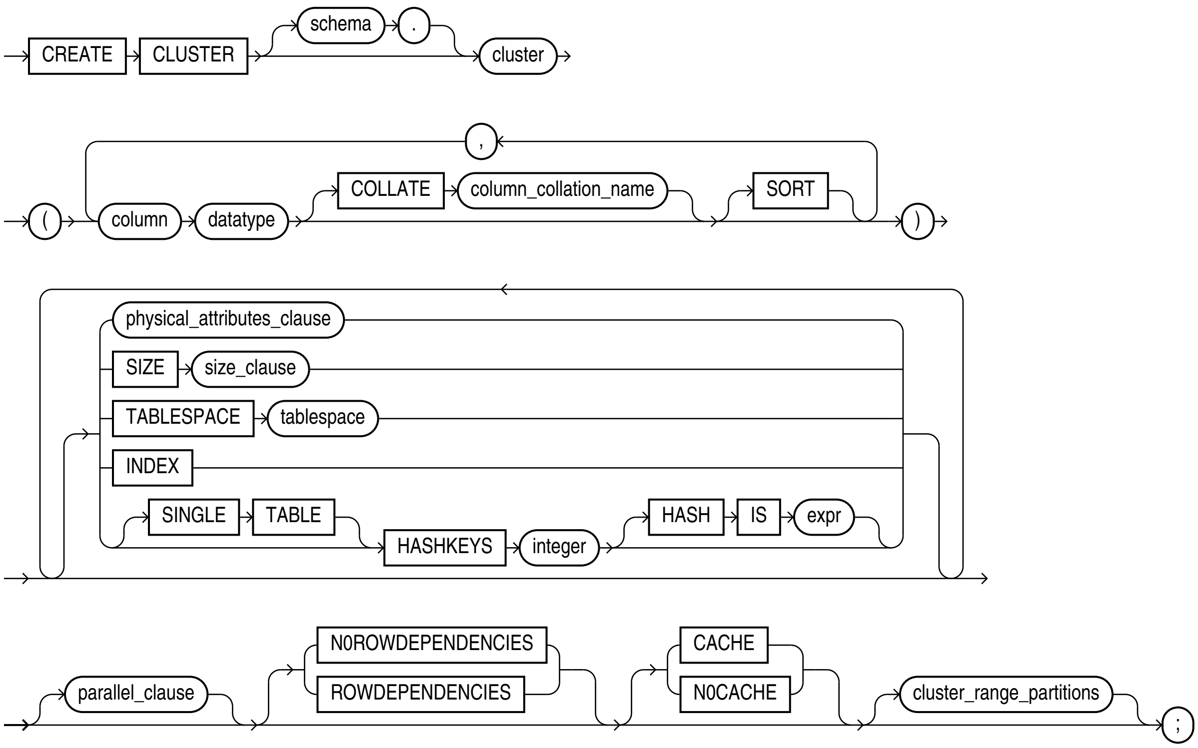
Prerequisites

To create a cluster in your own schema, you must have `CREATE CLUSTER` system privilege. To create a cluster in another user's schema, you must have `CREATE ANY CLUSTER` system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or the `UNLIMITED TABLESPACE` system privilege.

Oracle Database does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language (DML) statements cannot be issued against cluster tables in an indexed cluster until you create a cluster index with a `CREATE INDEX` statement.

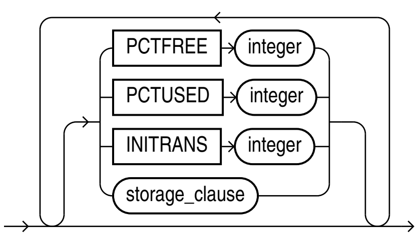
Syntax

***create_cluster*::=**



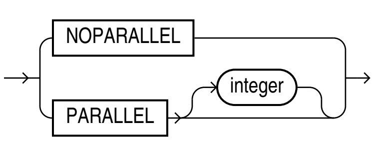
(*physical_attributes_clause*::=, *size_clause*::=, *cluster_range_partitions*::=)

***physical_attributes_clause*::=**

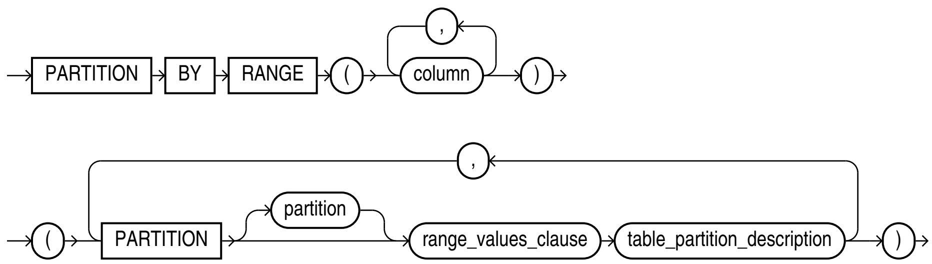


(*storage_clause*::=)

***parallel_clause*::=**



***cluster_range_partitions*::=**



(*range_values_clause*::=, *table_partition_description*::=)

Semantics

schema

Specify the schema to contain the cluster. If you omit *schema*, then Oracle Database creates the cluster in your current schema.

cluster

Specify is the name of the cluster to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

After you create a cluster, you add tables to it. A cluster can contain a maximum of 32 tables. Object tables and tables containing LOB columns or columns of the `Any*` Oracle-supplied types cannot be part of a cluster. After you create a cluster and add tables to it, the cluster is transparent. You can access clustered tables with SQL statements just as you can access nonclustered tables.

> **See Also:**
>
> CREATE TABLE for information on adding tables to a cluster, "Creating a Cluster: Example", and "Adding Tables to a Cluster: Example"

column

Specify one or more names of columns in the cluster key. You can specify up to 16 cluster key columns. These columns must correspond in both data type and size to columns in each of the clustered tables, although they need not correspond in name.

You cannot specify integrity constraints as part of the definition of a cluster key column. Instead, you can associate integrity constraints with the tables that belong to the cluster.

> **See Also:**
>
> "Cluster Keys: Example"

datatype

Specify the data type of each cluster key column.

Restrictions on Cluster Data Types

Cluster data types are subject to the following restrictions:

- You cannot specify a cluster key column of data type `LONG`, `LONG RAW`, `REF`, nested table, varray, `BLOB`, `CLOB`, `BFILE`, the `Any*` Oracle-supplied types, or user-defined object type.
- You can specify a column of type `ROWID`, but Oracle Database does not guarantee that the values in such columns are valid rowids.

> **See Also:**
>
> "Data Types " for information on data types

COLLATE

Use this clause to specify the data-bound collation for character data type columns in the cluster key.

For *`column_collation_name`*, specify the collation as follows:

- When creating an indexed cluster or a sorted hash cluster, you can specify one of the following collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.
- When creating a hash cluster that is not sorted, you can specify any valid named collation or pseudo-collation.

If you omit this clause, then columns in the cluster key inherit the effective schema default collation of the schema containing the cluster. Refer to the DEFAULT_COLLATION clause of `ALTER SESSION` for more information on the effective schema default collation.

The collations of cluster key columns must match the collations of the corresponding columns in the tables created in the cluster.

You can specify the `COLLATE` clause only if the `COMPATIBLE` initialization parameter is set to `12.2` or greater, and the `MAX_STRING_SIZE` initialization parameter is set to `EXTENDED`.

To change the collation of a cluster key column, you must recreate the cluster.

SORT

The `SORT` keyword is valid only if you are creating a hash cluster. Table rows are hashed into buckets on cluster key columns without `SORT`, and then sorted in each bucket on the columns with this clause. This may improve response time during subsequent operations on the clustered data.

All columns without the `SORT` clause must come before all columns with the `SORT` clause in the `CREATE CLUSTER` statement.

Restriction on Sorted Hash Clusters

Row dependency is not supported for sorted hash clusters.

> **See Also:**
>
> - See "HASHKEYS Clause" for information on creating a hash cluster.
> - *Managing Hash Clusters* for more information.

physical_attributes_clause

The `physical_attributes_clause` lets you specify the storage characteristics of the cluster. Each table in the cluster uses these storage characteristics as well. If you do not specify values for these parameters, then Oracle Database uses the following defaults:

- `PCTFREE`: 10
- `PCTUSED`: 40
- `INITRANS`: 2 or the default value of the tablespace to contain the cluster, whichever is greater

> **See Also:**
>
> *physical_attributes_clause* and *storage_clause* for a complete description of these clauses

SIZE

Specify the amount of space in bytes reserved to store all rows with the same cluster key value or the same hash value. This space determines the maximum number of cluster or hash values stored in a data block. If `SIZE` is not a divisor of the data block size, then Oracle Database uses the next largest divisor. If `SIZE` is larger than the data block size, then the database uses the operating system block size, reserving at least one data block for each cluster or hash value.

The database also considers the length of the cluster key when determining how much space to reserve for the rows having a cluster key value. Larger cluster keys require larger sizes. To see the actual size, query the `KEY_SIZE` column of the `USER_CLUSTERS` data dictionary view. (This value does not apply to hash clusters, because hash values are not actually stored in the cluster.)

If you omit this parameter, then the database reserves one data block for each cluster key value or hash value.

TABLESPACE

Specify the tablespace in which the cluster is to be created.

INDEX Clause

Specify `INDEX` to create an **indexed cluster**. In an indexed cluster, Oracle Database stores together rows having the same cluster key value. Each distinct cluster key value is stored only once in each data block, regardless of the number of tables and rows in which it occurs. If you specify neither `INDEX` nor `HASHKEYS`, then Oracle Database creates an indexed cluster by default.

After you create an indexed cluster, you must create an index on the cluster key before you can issue any data manipulation language (DML) statements against a table in the cluster. This index is called the **cluster index**.

You cannot create a cluster index for a hash cluster, and you need not create an index on a hash cluster key.

> **See Also:**
>
> CREATE INDEX for information on creating a cluster index and *Oracle Database Concepts* for general information in indexed clusters

HASHKEYS Clause

Specify the `HASHKEYS` clause to create a **hash cluster** and specify the number of hash values for the hash cluster. In a hash cluster, Oracle Database stores together rows that have the same hash key value. The hash value for a row is the value returned by the hash function of the cluster.

Oracle Database rounds up the `HASHKEYS` value to the nearest prime number to obtain the actual number of hash values. The minimum value for this parameter is 2. If you omit both the `INDEX` clause and the `HASHKEYS` parameter, then the database creates an indexed cluster by default.

When you create a hash cluster, the database immediately allocates space for the cluster based on the values of the `SIZE` and `HASHKEYS` parameters.

> **See Also:**
>
> *Oracle Database Concepts* for more information on how Oracle Database allocates space for clusters and "Hash Clusters: Examples"

SINGLE TABLE

`SINGLE TABLE` indicates that the cluster is a type of hash cluster containing only one table. This clause can provide faster access to rows in the table.

Restriction on Single-table Clusters

Only one table can be present in the cluster at a time. However, you can drop the table and create a different table in the same cluster.

> **See Also:**
>
> "Single-Table Hash Clusters: Example"

HASH IS *expr*

Specify an expression to be used as the hash function for the hash cluster. The expression:

- Must evaluate to a positive value
- Must contain at least one column, with referenced columns of any data type as long as the entire expression evaluates to a number of scale 0. For example: *number_column* * LENGTH(*varchar2_column*)
- Cannot reference user-defined PL/SQL functions
- Cannot reference the pseudocolumns LEVEL or ROWNUM
- Cannot reference the user-related functions USERENV, UID, or USER or the datetime functions CURRENT_DATE, CURRENT_TIMESTAMP, DBTIMEZONE, EXTRACT (datetime), FROM_TZ, LOCALTIMESTAMP, NUMTODSINTERVAL, NUMTOYMINTERVAL, SESSIONTIMEZONE, SYSDATE, SYSTIMESTAMP, TO_DSINTERVAL, TO_TIMESTAMP, TO_DATE, TO_TIMESTAMP_TZ, TO_YMINTERVAL, and TZ_OFFSET.
- Cannot evaluate to a constant
- Cannot be a scalar subquery expression
- Cannot contain columns qualified with a schema or object name (other than the cluster name)

If you omit the HASH IS clause, then Oracle Database uses an internal hash function for the hash cluster.

For information on existing hash functions, query the USER_, ALL_, and DBA_CLUSTER_HASH_EXPRESSIONS data dictionary tables.

The cluster key of a hash column can have one or more columns of any data type. Hash clusters with composite cluster keys or cluster keys made up of noninteger columns must use the internal hash function.

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views

parallel_clause

The *parallel_clause* lets you parallelize the creation of the cluster.

For complete information on this clause, refer to *parallel_clause* in the documentation on CREATE TABLE.

NOROWDEPENDENCIES | ROWDEPENDENCIES

This clause has the same behavior for a cluster that it has for a table. Refer to "NOROWDEPENDENCIES | ROWDEPENDENCIES" in `CREATE TABLE` for information.

CACHE | NOCACHE

CACHE

Specify `CACHE` if you want the blocks retrieved for this cluster to be placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables.

NOCACHE

Specify `NOCACHE` if you want the blocks retrieved for this cluster to be placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the default behavior.

`NOCACHE` has no effect on clusters for which you specify `KEEP` in the *`storage_clause`*.

cluster_range_partitions

Specify the *`cluster_range_partitions`* clause to create a range-partitioned hash cluster. If you specify this clause, then you must also specify the `HASHKEYS` clause.

Use the *`cluster_range_partitions`* clause to partition the cluster on ranges of values from the column list. When you add a table to a range-partitioned hash cluster, it is automatically partitioned on the same columns, with the same number of partitions, and on the same partition bounds as the cluster. Oracle Database assigns system-generated names to the table partitions.

The *`cluster_range_partitions`* clause has the same semantics as the *`range_partitions`* clause of `CREATE TABLE`, except that here you cannot specify the `INTERVAL` clause. For complete information, refer to *range_partitions* in the documentation on `CREATE TABLE`.

> **See Also:**
>
> "Range-Partitioned Hash Clusters: Example"

Examples

Creating a Cluster: Example

The following statement creates a cluster named `personnel` with the cluster key column `department`, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
   (department NUMBER(4))
SIZE 512
STORAGE (initial 100K next 50K);
```

Cluster Keys: Example

The following statement creates the cluster index on the cluster key of `personnel`:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can add tables to the index and perform DML operations on those tables.

Adding Tables to a Cluster: Example

The following statements create some departmental tables from the sample `hr.employees` table and add them to the personnel cluster created in the earlier example:

```
CREATE TABLE dept_10
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE department_id = 10;

CREATE TABLE dept_20
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE department_id = 20;
```

Hash Clusters: Examples

The following statement creates a hash cluster named `language` with the cluster key column `cust_language`, a maximum of 10 hash key values, each of which is allocated 512 bytes, and storage parameter values:

```
CREATE CLUSTER language (cust_language VARCHAR2(3))
   SIZE 512 HASHKEYS 10
   STORAGE (INITIAL 100k next 50k);
```

Because the preceding statement omits the `HASH IS` clause, Oracle Database uses the internal hash function for the cluster.

The following statement creates a hash cluster named `address` with the cluster key made up of the columns `postal_code` and `country_id`, and uses a SQL expression containing these columns for the hash function:

```
CREATE CLUSTER address
   (postal_code NUMBER, country_id CHAR(2))
   HASHKEYS 20
   HASH IS MOD(postal_code + country_id, 101);
```

Single-Table Hash Clusters: Example

The following statement creates a single-table hash cluster named `cust_orders` with the cluster key `customer_id` and a maximum of 100 hash key values, each of which is allocated 512 bytes:

```
CREATE CLUSTER cust_orders (customer_id NUMBER(6))
   SIZE 512 SINGLE TABLE HASHKEYS 100;
```

Range-Partitioned Hash Clusters: Example

The following statement creates a range-partitioned hash cluster named `sales` with five range partitions based on the amount sold. The cluster key is made up of the columns `amount_sold` and `prod_id`. The cluster uses the hash function `(amount_sold * 10 + prod_id)` and has a maximum of 100000 hash key values, each of which is allocated 300 bytes.

```
CREATE CLUSTER sales (amount_sold NUMBER, prod_id NUMBER)
  HASHKEYS 100000
  HASH IS (amount_sold * 10 + prod_id)
```

```
SIZE 300
TABLESPACE example
PARTITION BY RANGE (amount_sold)
  (PARTITION p1 VALUES LESS THAN (2001),
   PARTITION p2 VALUES LESS THAN (4001),
   PARTITION p3 VALUES LESS THAN (6001),
   PARTITION p4 VALUES LESS THAN (8001),
   PARTITION p5 VALUES LESS THAN (MAXVALUE));
```

CREATE CONTEXT

Purpose

Use the `CREATE CONTEXT` statement to:

- Create a namespace for a **context** (a set of application-defined attributes that validates and secures an application)
- Associate the namespace with the externally created package that sets the context

You can use the `DBMS_SESSION.SET_CONTEXT` procedure in your designated package to set or reset the attributes of the context.

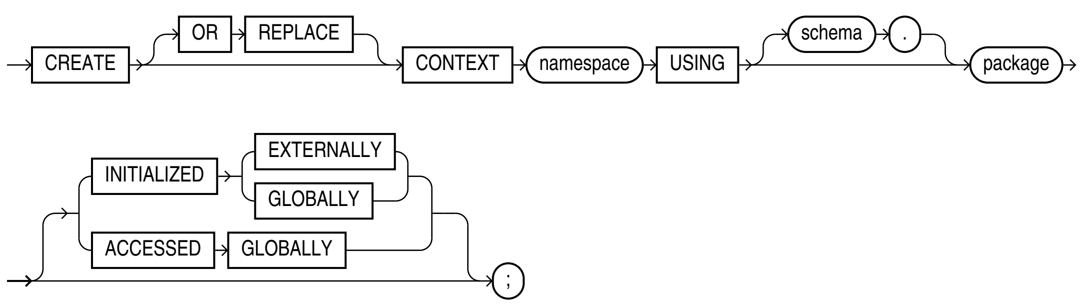
> **See Also:**
>
> - *Oracle Database Security Guide* for a discussion of contexts
> - *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_SESSION.SET_CONTEXT` procedure

Prerequisites

To create a context namespace, you must have `CREATE ANY CONTEXT` system privilege.

Syntax

***create_context*::=**



Semantics

OR REPLACE

Specify OR REPLACE to redefine an existing context namespace using a different package.

namespace

Specify the name of the context namespace to create or modify. The name must satisfy the requirements listed in "Database Object Naming Rules ". Context namespaces are always stored in the schema SYS.

> **See Also:**
>
> "Database Object Naming Rules " for guidelines on naming a context namespace

schema

Specify the schema owning *package*. If you omit *schema*, then Oracle Database uses the current schema.

package

Specify the PL/SQL package that sets or resets the context attributes under the namespace for a user session.

To provide some design flexibility, Oracle Database does not verify the existence of the schema or the validity of the package at the time you create the context.

INITIALIZED Clause

The INITIALIZED clause lets you specify an entity other than Oracle Database that can initialize the context namespace.

EXTERNALLY

EXTERNALLY indicates that the namespace can be initialized using an OCI interface when establishing a session.

> **See Also:**
>
> *Oracle Call Interface Programmer's Guide* for information on using OCI to establish a session

GLOBALLY

GLOBALLY indicates that the namespace can be initialized by the LDAP directory when a global user connects to the database.

After the session is established, only the designated PL/SQL package can issue commands to write to any attributes inside the namespace.

> **See Also:**
>
> *Oracle Database Security Guide* for information on establishing globally initialized contexts

ACCESSED GLOBALLY

This clause indicates that any application context set in `namespace` is accessible throughout the entire instance. This setting lets multiple sessions share application attributes.

Examples

Creating an Application Context: Example

This example uses a PL/SQL package `emp_mgmt`, which validates and secures a human resources application. See *Oracle Database PL/SQL Language Reference* for the example that creates that package. The following statement creates the context namespace `hr_context` and associates it with the package `emp_mgmt`:

```
CREATE CONTEXT hr_context USING emp_mgmt;
```

You can control data access based on this context using the `SYS_CONTEXT` function. For example, the `emp_mgmt` package has defined an attribute `department_id` as a particular department identifier. You can secure the base table `employees` by creating a view that restricts access based on the value of `department_id`, as follows:

```
CREATE VIEW hr_org_secure_view AS
   SELECT * FROM employees
   WHERE department_id = SYS_CONTEXT('hr_context', 'department_id');
```

> **See Also:**
>
> SYS_CONTEXT and *Oracle Database Security Guide* for more information on using application contexts to retrieve user information

CREATE CONTROLFILE

> **Note:**
>
> Oracle recommends that you perform a full backup of all files in the database before using this statement. For more information, see *Oracle Database Backup and Recovery User's Guide*.

Purpose

The `CREATE CONTROLFILE` statement should be used in only a few cases. Use this statement to re-create a control file if all control files being used by the database are lost **and** no backup control file exists. You can also use this statement to change the maximum number of redo log file groups, redo log file members, archived redo log files, data files, or instances that can concurrently have the database mounted and open.

To change the name of the database, Oracle recommends that you use the `DBNEWID` utility rather than the `CREATE CONTROLFILE` statement. `DBNEWID` is preferable because no `OPEN RESETLOGS` operation is required after changing the database name.

See Also:

- *Oracle Database Utilities* for more information about the `DBNEWID` utility
- `ALTER DATABASE` "BACKUP CONTROLFILE Clause" for information creating a script based on an existing database control file

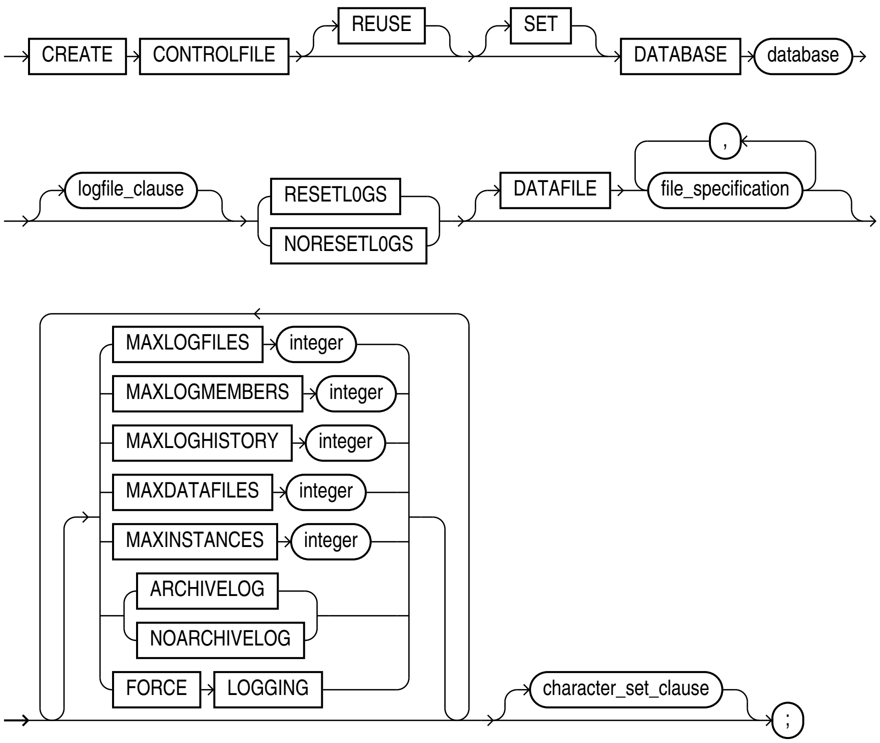
Prerequisites

To create a control file, you must have the `SYSDBA` or `SYSBACKUP` system privilege.

The database must not be mounted by any instance. After successfully creating the control file, Oracle mounts the database in the mode specified by the `CLUSTER_DATABASE` parameter. The DBA must then perform media recovery before opening the database. If you are using the database with Oracle Real Application Clusters (Oracle RAC), then you must then shut down and remount the database in `SHARED` mode (by setting the value of the `CLUSTER_DATABASE` initialization parameter to `TRUE`) before other instances can start up.

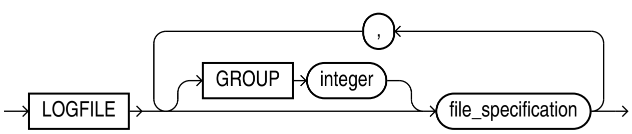
Syntax

***create_controlfile*::=**



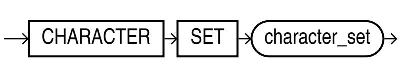
(*storage_clause*::=)

***logfile_clause*::=**



(*file_specification*::=)

***character_set_clause*::=**



Semantics

When you issue a `CREATE CONTROLFILE` statement, Oracle Database creates a new control file based on the information you specify in the statement. The control file resides in the location specified in the `CONTROL_FILES` initialization parameter. If that parameter does not have a value, then the database creates an Oracle-managed control file in the default control file destination, which is one of the following (in order of precedence):

1. One or more control files as specified in the `DB_CREATE_ONLINE_LOG_DEST_n` initialization parameter. The file in the first directory is the primary control file. When `DB_CREATE_ONLINE_LOG_DEST_n` is specified, the database does not create a control file in `DB_CREATE_FILE_DEST` or in `DB_RECOVERY_FILE_DEST` (the fast recovery area).
2. If no value is specified for `DB_CREATE_ONLINE_LOG_DEST_n`, but values are set for both the `DB_CREATE_FILE_DEST` and `DB_RECOVERY_FILE_DEST`, then the database creates one control file in each location. The location specified in `DB_CREATE_FILE_DEST` is the primary control file.
3. If a value is specified only for `DB_CREATE_FILE_DEST`, then the database creates one control file in that location.
4. If a value is specified only for `DB_RECOVERY_FILE_DEST`, then the database creates one control file in that location.

If no values are set for any of these parameters, then the database creates a control file in the default location for the operating system on which the database is running. This control file is not an Oracle Managed File.

If you omit any clauses, then Oracle Database uses the default values rather than the values for the previous control file. After successfully creating the control file, Oracle Database mounts the database in the mode specified by the initialization parameter `CLUSTER_DATABASE`. If that parameter is not set, then the default value is `FALSE`, and the database is mounted in `EXCLUSIVE` mode. Oracle recommends that you then shut down the instance and take a full backup of all files in the database.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide*

REUSE

Specify `REUSE` to indicate that existing control files identified by the initialization parameter `CONTROL_FILES` can be reused, overwriting any information they may currently contain. If you omit this clause and any of these control files already exists, then Oracle Database returns an error.

DATABASE Clause

Specify the name of the database. The value of this parameter must be the existing database name established by the previous `CREATE DATABASE` statement or `CREATE CONTROLFILE` statement.

SET DATABASE Clause

Use `SET DATABASE` to change the name of the database. The name of a database can be as long as eight bytes.

When you specify this clause, you must also specify `RESETLOGS`. If you want to rename the database and retain your existing log files, then after issuing this `CREATE CONTROLFILE` statement you must complete a full database recovery using an `ALTER DATABASE RECOVER USING BACKUP CONTROLFILE` statement.

logfile_clause

Use the *`logfile_clause`* to specify the redo log files for your database. You must list all members of all redo log file groups.

Use the *`redo_log_file_spec`* form of *`file_specification`* (see *file_specification*) to list regular redo log files in an operating system file system or to list Oracle ASM disk group redo log files. When using a form of *`ASM_filename`*, you cannot specify the *`autoextend_clause`* of the *`redo_log_file_spec`*.

If you specify `RESETLOGS` in this clause, then you must use one of the file creation forms of *`ASM_filename`*. If you specify `NORESETLOGS`, then you must specify one of the reference forms of *`ASM_filename`*.

> **See Also:**
>
> *ASM_filename* for information on the different forms of syntax and *Oracle Automatic Storage Management Administrator's Guide* for general information about using Oracle ASM

GROUP *integer*

Specify the logfile group number. If you specify `GROUP` values, then Oracle Database verifies these values with the `GROUP` values when the database was last open.

If you omit this clause, then the database creates logfiles using system default values. In addition, if either the `DB_CREATE_ONLINE_LOG_DEST_`*n* or `DB_CREATE_FILE_DEST` initialization parameter has been set, and if you have specified `RESETLOGS`, then the database creates two logs in the default logfile destination specified in the `DB_CREATE_ONLINE_LOG_DEST_`*n* parameter, and if it is not set, then in the `DB_CREATE_FILE_DEST` parameter.

> **See Also:**
>
> *file_specification* for a full description of this clause

RESETLOGS

Specify `RESETLOGS` if you want Oracle Database to ignore the contents of the files listed in the `LOGFILE` clause. These files do not have to exist. You must specify this clause if you have specified the `SET DATABASE` clause.

Each *`redo_log_file_spec`* in the `LOGFILE` clause must specify the `SIZE` parameter. The database assigns all online redo log file groups to thread 1 and enables this thread for public use by any instance. After using this clause, you must open the database using the `RESETLOGS` clause of the `ALTER DATABASE` statement.

NORESETLOGS

Specify NORESETLOGS if you want Oracle Database to use all files in the LOGFILE clause as they were when the database was last open. These files must exist and must be the current online redo log files rather than restored backups. The database reassigns the redo log file groups to the threads to which they were previously assigned and reenables the threads as they were previously enabled.

You cannot specify NORESETLOGS if you have specified the SET DATABASE clause to change the name of the database. Refer to "SET DATABASE Clause" for more information.

DATAFILE Clause

Specify the data files of the database. You must list all data files. These files must all exist, although they may be restored backups that require media recovery.

Do not include in the DATAFILE clause any data files in read-only tablespaces. You can add these types of files to the database later. Also, do not include in this clause any temporary data files (temp files).

Use the *datafile_tempfile_spec* form of *file_specification* (see *file_specification*) to list regular data files and temp files in an operating system file system or to list Oracle ASM disk group files. When using a form of *ASM_filename*, you must use one of the reference forms of *ASM_filename*. Refer to *ASM_filename* for information on the different forms of syntax.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for general information about using Oracle ASM

Restriction on DATAFILE

You cannot specify the *autoextend_clause* of *file_specification* in this DATAFILE clause.

MAXLOGFILES Clause

Specify the maximum number of online redo log file groups that can ever be created for the database. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The default and maximum values depend on your operating system. The value that you specify should not be less than the greatest GROUP value for any redo log file group.

MAXLOGMEMBERS Clause

Specify the maximum number of members, or identical copies, for a redo log file group. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY Clause

This parameter is useful only if you are using Oracle Database in ARCHIVELOG mode. Specify your current estimate of the maximum number of archived redo log file groups needed for automatic media recovery of the database. The database uses this value to determine how much space to allocate in the control file for the names of archived redo log files.

The minimum value is 0. The default value is a multiple of the `MAXINSTANCES` value and depends on your operating system. The maximum value is limited only by the maximum size of the control file. The database will continue to add additional space to the appropriate section of the control file as needed, so that you do not need to re-create the control file if your your original configuration is no longer adequate. As a result, the actual value of this parameter can eventually exceed the value you specify.

MAXDATAFILES Clause

Specify the initial sizing of the data files section of the control file at `CREATE DATABASE` or `CREATE CONTROLFILE` time. An attempt to add a file whose number is greater than `MAXDATAFILES`, but less than or equal to `DB_FILES`, causes the control file to expand automatically so that the data files section can accommodate more files.

The number of data files accessible to your instance is also limited by the initialization parameter `DB_FILES`.

MAXINSTANCES Clause

Specify the maximum number of instances that can simultaneously have the database mounted and open. This value takes precedence over the value of the initialization parameter `INSTANCES`. The minimum value is 1. The maximum and default values depend on your operating system.

ARCHIVELOG | NOARCHIVELOG

Specify `ARCHIVELOG` to archive the contents of redo log files before reusing them. This clause prepares for the possibility of media recovery as well as instance or system failure recovery.

If you omit both the `ARCHIVELOG` clause and `NOARCHIVELOG` clause, then Oracle Database chooses `NOARCHIVELOG` mode by default. After creating the control file, you can change between `ARCHIVELOG` mode and `NOARCHIVELOG` mode with the `ALTER DATABASE` statement.

FORCE LOGGING

Use this clause to put the database into `FORCE LOGGING` mode after control file creation. When the database is in this mode, Oracle Database logs all changes in the database except changes to temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any `NOLOGGING` or `FORCE LOGGING` settings you specify for individual tablespaces and any `NOLOGGING` settings you specify for individual database objects. If you omit this clause, then the database will not be in `FORCE LOGGING` mode after the control file is created.

> **Note:**
>
> `FORCE LOGGING` mode can have performance effects. Refer to *Oracle Database Administrator's Guide* for information on when to use this setting.

character_set_clause

If you specify a character set, then Oracle Database reconstructs character set information in the control file. If media recovery of the database is subsequently required, then this information will be available before the database is open, so that tablespace names can be correctly interpreted during recovery. This clause is required only if you are using a character set other than the default, which depends on your operating system. Oracle Database prints the current database character set to the alert log in `$ORACLE_HOME/log` during startup.

If you are re-creating your control file and you are using Recovery Manager for tablespace recovery, and if you specify a different character set from the one stored in the data dictionary, then tablespace recovery will not succeed. However, at database open, the control file character set will be updated with the correct character set from the data dictionary.

You cannot modify the character set of the database with this clause.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information on tablespace recovery

Examples

Creating a Controlfile: Example

This statement re-creates a control file. In this statement, database `demo` was created with the WE8DEC character set. The example uses the word *`path`* where you would normally insert the path on your system to the appropriate Oracle Database directories.

```
STARTUP NOMOUNT

CREATE CONTROLFILE REUSE DATABASE "demo" NORESETLOGS NOARCHIVELOG
    MAXLOGFILES 32
    MAXLOGMEMBERS 2
    MAXDATAFILES 32
    MAXINSTANCES 1
    MAXLOGHISTORY 449
LOGFILE
  GROUP 1 '/path/oracle/dbs/t_log1.f'  SIZE 500K,
  GROUP 2 '/path/oracle/dbs/t_log2.f'  SIZE 500K
# STANDBY LOGFILE
DATAFILE
  '/path/oracle/dbs/t_db1.f',
  '/path/oracle/dbs/dbu19i.dbf',
  '/path/oracle/dbs/tbs_11.f',
  '/path/oracle/dbs/smundo.dbf',
  '/path/oracle/dbs/demo.dbf'
CHARACTER SET WE8DEC
;
```

CREATE DATABASE

> **Note:**
>
> This statement prepares a database for initial use and erases any data currently in the specified files. Use this statement only when you understand its ramifications.

> **Note:**
>
> In this release of Oracle Database and in subsequent releases, several enhancements are being made to ensure the security of default database user accounts. You can find a security checklist for this release in *Oracle Database Security Guide*. Oracle recommends that you read this checklist and configure your database accordingly.

Purpose

Use the `CREATE DATABASE` statement to create a database, making it available for general use.

This statement erases all data in any specified data files that already exist in order to prepare them for initial database use. If you use the statement on an existing database, then all data in the data files is lost.

After creating the database, this statement mounts it in either exclusive or parallel mode, depending on the value of the `CLUSTER_DATABASE` initialization parameter and opens it, making it available for normal use. You can then create tablespaces for the database.

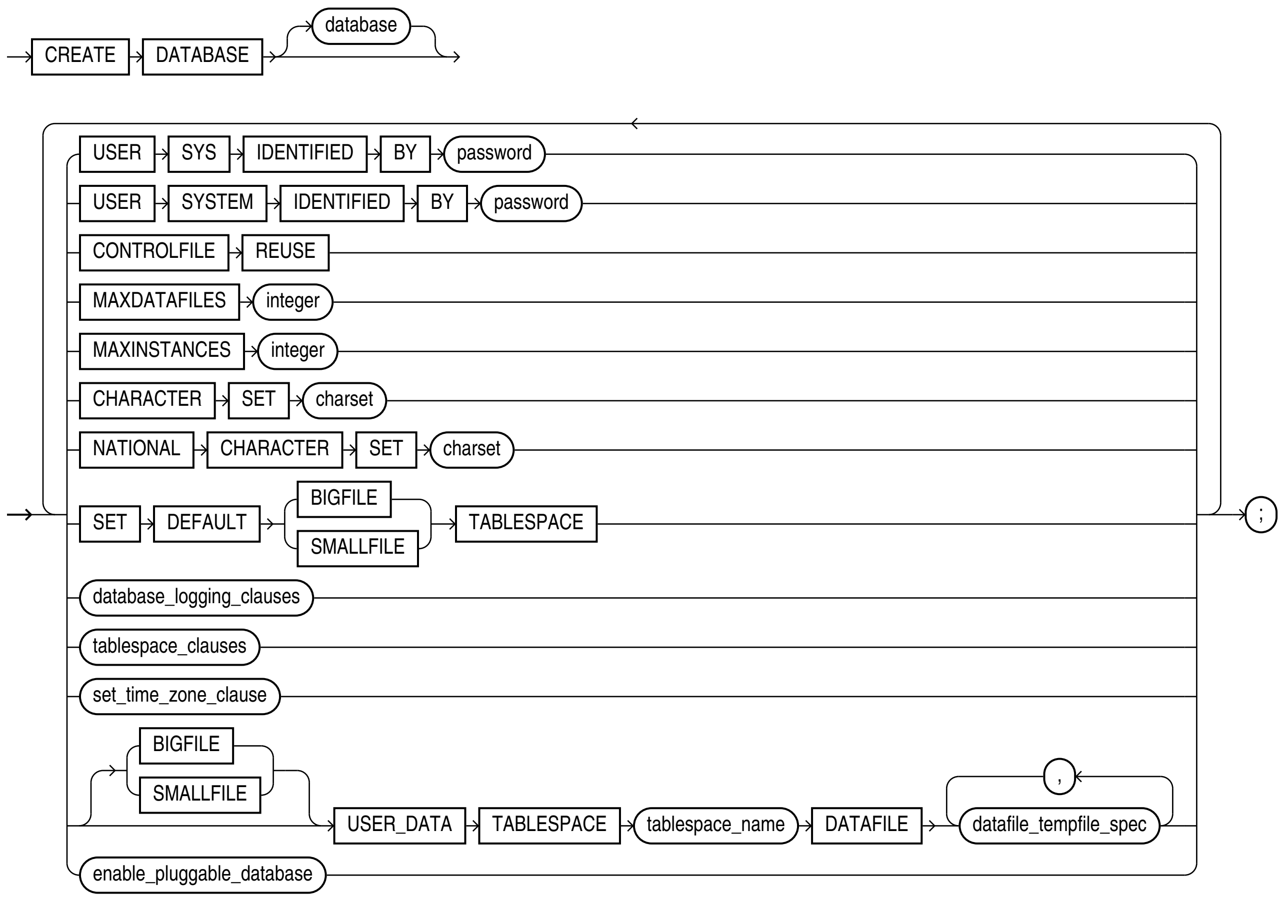
> **See Also:**
>
> - ALTER DATABASE for information on modifying a database
> - *Oracle Database Java Developer's Guide* for information on creating an Oracle Java virtual machine
> - CREATE TABLESPACE for information on creating tablespaces

Prerequisites

To create a database, you must have the `SYSDBA` system privilege. An initialization parameter file with the name of the database to be created must be available, and you must be in `STARTUP NOMOUNT` mode.

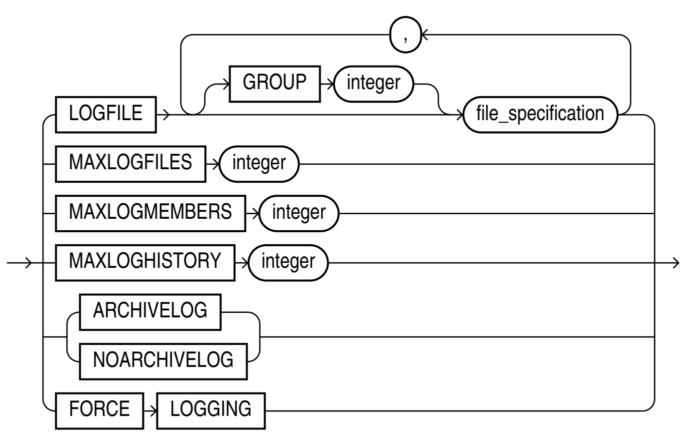
Syntax

***create_database*::=**



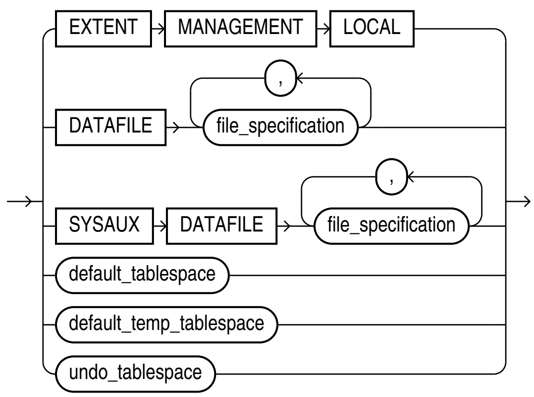
(*database_logging_clauses*::=, *tablespace_clauses*::=, *set_time_zone_clause*::=, *datafile_tempfile_spec*::=, *enable_pluggable_database*::=)

***database_logging_clauses*::=**



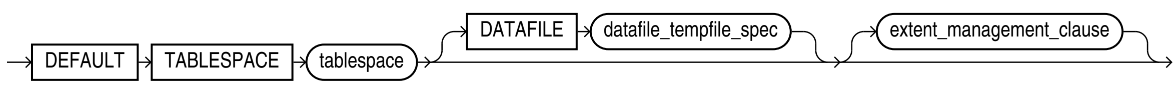
(*file_specification::=*)

***tablespace_clauses*::=**

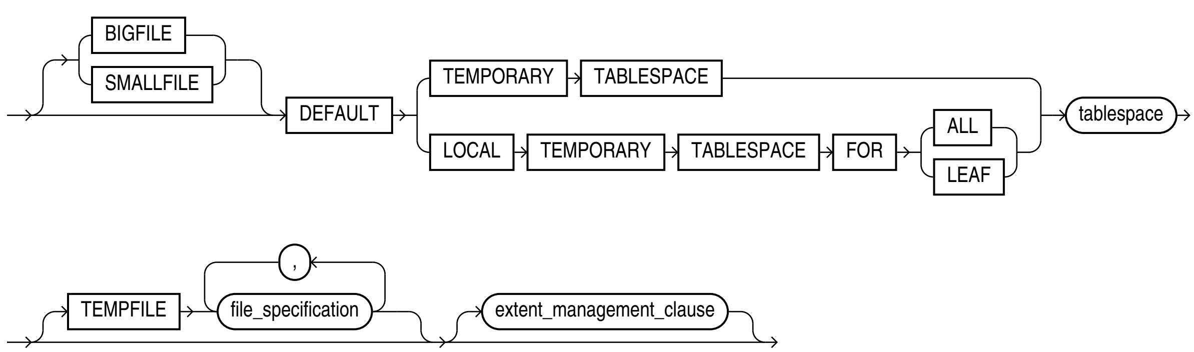


(*file_specification::=*, *default_tablespace::=*, *default_temp_tablespace::=*, *undo_tablespace::=*, *undo_tablespace::=*)

***default_tablespace*::=**

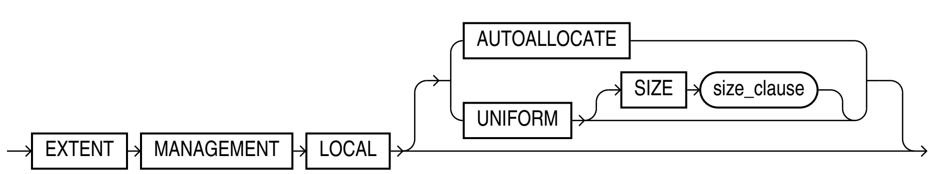


***default_temp_tablespace*::=**



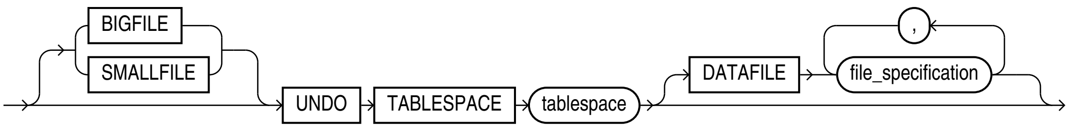
(*file_specification::=*)

***extent_management_clause*::=**



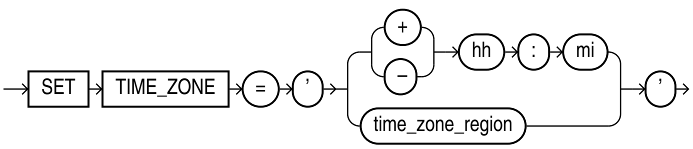
(*size_clause*::=)

***undo_tablespace*::=**

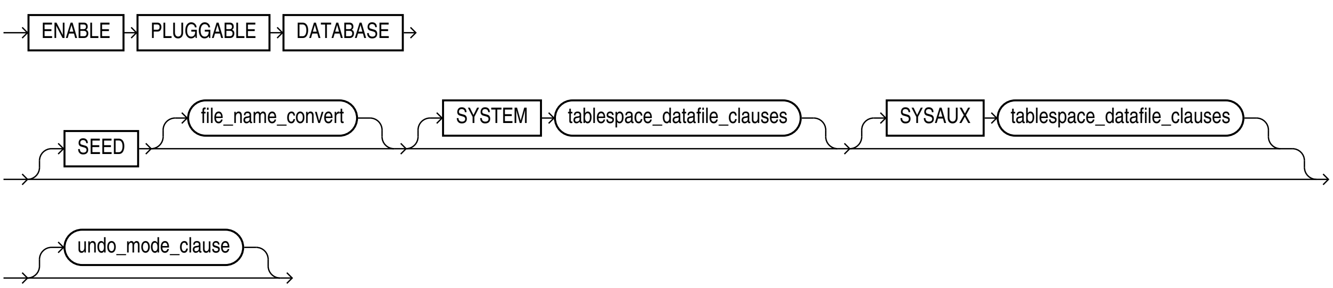


(*file_specification*::=)

***set_time_zone_clause*::=**

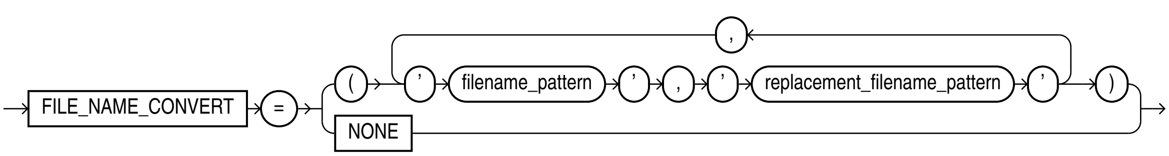


***enable_pluggable_database*::=**

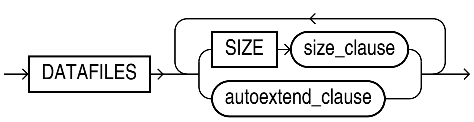


(*tablespace_datafile_clauses*::=, *undo_mode_clause*::=)

***file_name_convert*::=**

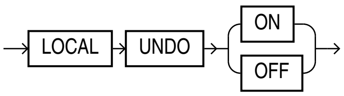


***tablespace_datafile_clauses*::=**



(*size_clause*::=, *autoextend_clause*::=)

***undo_mode_clause*::=**



Semantics

database

Specify the name of the database to be created. The name must match the value of the `DB_NAME` initialization parameter. The name can be up to 8 bytes long and can contain only ASCII characters. The following characters are valid in a database name: alphanumeric characters, underscore (_), number sign (#), and dollar sign ($). No other characters are valid. The database name must start with an alphabetic character. Oracle Database writes this name into the control file. If you subsequently issue an `ALTER DATABASE` statement that explicitly specifies a database name, then Oracle Database verifies that name with the name in the control file.

The database name is case insensitive and is stored in uppercase ASCII characters. If you specify the database name as a quoted identifier, then the quotation marks are silently ignored.

> **Note:**
>
> You cannot use special characters from European or Asian character sets in a database name. For example, characters with umlauts are not allowed.

If you omit the database name from a `CREATE DATABASE` statement, then Oracle Database uses the name specified by the initialization parameter `DB_NAME`. The `DB_NAME` initialization parameter must be set in the database initialization parameter file, and if you specify a different name from the value of that parameter, then the database returns an error. Refer to "Database Object Naming Rules " for additional rules to which database names should adhere.

USER SYS ..., USER SYSTEM ...

Use these clauses to establish passwords for the `SYS` and `SYSTEM` users. These clauses are not mandatory in this release. However, if you specify either clause, then you must specify both clauses.

If you do not specify these clauses, then Oracle Database creates default passwords `change_on_install` for user `SYS` and `manager` for user `SYSTEM`. You can subsequently change these passwords using the `ALTER USER` statement. You can also use `ALTER USER` to add password management attributes after database creation.

> **See Also:**
>
> ALTER USER

CONTROLFILE REUSE Clause

Specify `CONTROLFILE REUSE` to reuse existing control files identified by the initialization parameter `CONTROL_FILES`, overwriting any information they currently contain. Normally you use this clause only when you are re-creating a database, rather than creating one for the first time. When you create a database for the first time, Oracle Database creates a control file in the default destination, which is dependent on the value or several initialization parameters. See `CREATE CONTROLFILE`, "Semantics".

You cannot use this clause if you also specify a parameter value that requires that the control file be larger than the existing files. These parameters are `MAXLOGFILES`, `MAXLOGMEMBERS`, `MAXLOGHISTORY`, `MAXDATAFILES`, and `MAXINSTANCES`.

If you omit this clause and any of the files specified by `CONTROL_FILES` already exist, then the database returns an error.

MAXDATAFILES Clause

Specify the initial sizing of the data files section of the control file at `CREATE DATABASE` or `CREATE CONTROLFILE` time. An attempt to add a file whose number is greater than `MAXDATAFILES`, but less than or equal to `DB_FILES`, causes the Oracle Database control file to expand automatically so that the data files section can accommodate more files.

The number of data files accessible to your instance is also limited by the initialization parameter `DB_FILES`.

MAXINSTANCES Clause

Specify the maximum number of instances that can simultaneously have this database mounted and open. This value takes precedence over the value of initialization parameter `INSTANCES`. The minimum value is 1. The maximum value is 1055. The default depends on your operating system.

CHARACTER SET Clause

Specify the character set the database uses to store data. The supported character sets and default value of this parameter depend on your operating system.

Restriction on CHARACTER SET

You cannot specify the `AL16UTF16` character set as the database character set.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about choosing a character set

NATIONAL CHARACTER SET Clause

Specify the national character set used to store data in columns specifically defined as `NCHAR`, `NCLOB`, or `NVARCHAR2`. Valid values are `AL16UTF16` and `UTF8`. The default is `AL16UTF16`.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support

SET DEFAULT TABLESPACE Clause

Use this clause to determine the default type of subsequently created tablespaces and of the `SYSTEM` and `SYSAUX` tablespaces. Specify either `BIGFILE` or `SMALLFILE` to set the default type of subsequently created tablespaces as a bigfile or smallfile tablespace, respectively.

- A **bigfile tablespace** contains only one data file or temp file, which can contain up to approximately 4 billion (2^{32}) blocks. The maximum size of the single data file or temp file is 128 terabytes (TB) for a tablespace with 32K blocks and 32TB for a tablespace with 8K blocks.
- A **smallfile tablespace** is a traditional Oracle tablespace, which can contain 1022 data files or temp files, each of which can contain up to approximately 4 million (2^{22}) blocks.

If you omit this clause, then Oracle Database creates smallfile tablespaces by default.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information about bigfile tablespaces
> - "Setting the Default Type of Tablespaces: Example" for an example using this syntax

database_logging_clauses

Use the *`database_logging_clauses`* to determine how Oracle Database will handle redo log files for this database.

LOGFILE Clause

Specify one or more files to be used as redo log files. Use the *`redo_log_file_spec`* form of *`file_specification`* to create regular redo log files in an operating system file system or to create Oracle ASM disk group redo log files. When using a form of *`ASM_filename`*, you cannot specify the *`autoextend_clause`* of *`redo_log_file_spec`*.

The *`redo_log_file_spec`* clause specifies a redo log file group containing one or more redo log file members (copies). All redo log files specified in a `CREATE DATABASE` statement are added to redo log thread number 1.

> **See Also:**
>
> *file_specification* for a full description of this clause

If you omit the LOGFILE clause, then Oracle Database creates an Oracle-managed log file member in the default destination, which is one of the following locations (in order of precedence):

- If DB_CREATE_ONLINE_LOG_DEST_*n* is set, then the database creates a log file member in each directory specified, up to the value of the MAXLOGMEMBERS initialization parameter.
- If the DB_CREATE_ONLINE_LOG_DEST_*n* parameter is not set, but both the DB_CREATE_FILE_DEST and DB_RECOVERY_FILE_DEST initialization parameters are set, then the database creates one Oracle-managed log file member in each of those locations. The log file in the DB_CREATE_FILE_DEST destination is the first member.
- If only the DB_CREATE_FILE_DEST initialization parameter is specified, then Oracle Database creates a log file member in that location.
- If only the DB_RECOVERY_FILE_DEST initialization parameter is specified, then Oracle Database creates a log file member in that location.

In all these cases, the parameter settings must correctly specify operating system filenames or creation form Oracle ASM filenames, as appropriate.

If no values are set for any of these parameters, then the database creates a log file in the default location for the operating system on which the database is running. This log file is not an Oracle Managed File.

GROUP *integer*

Specify the number that identifies the redo log file group. The value of *integer* can range from 1 to the value of the MAXLOGFILES parameter. A database must have at least two redo log file groups. You cannot specify multiple redo log file groups having the same GROUP value. If you omit this parameter, then Oracle Database generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

MAXLOGFILES Clause

Specify the maximum number of redo log file groups that can ever be created for the database. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The default, minimum, and maximum values depend on your operating system.

MAXLOGMEMBERS Clause

Specify the maximum number of members, or copies, for a redo log file group. Oracle Database uses this value to determine how much space to allocate in the control file for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY Clause

This parameter is useful only if you are using Oracle Database in ARCHIVELOG mode with Oracle Real Application Clusters (Oracle RAC). Specify the maximum number of archived redo log files for automatic media recovery of Oracle RAC. The database uses this value to determine how much space to allocate in the control file for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the `MAXINSTANCES` value and depends on your operating system. The maximum value is limited only by the maximum size of the control file.

ARCHIVELOG

Specify `ARCHIVELOG` if you want the contents of a redo log file group to be archived before the group can be reused. This clause prepares for the possibility of media recovery.

NOARCHIVELOG

Specify `NOARCHIVELOG` if the contents of a redo log file group need not be archived before the group can be reused. This clause does not allow for the possibility of media recovery.

The default is `NOARCHIVELOG` mode. After creating the database, you can change between `ARCHIVELOG` mode and `NOARCHIVELOG` mode with the `ALTER DATABASE` statement.

FORCE LOGGING

Use this clause to put the database into `FORCE LOGGING` mode. Oracle Database will log all changes in the database except for changes in temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any `NOLOGGING` or `FORCE LOGGING` settings you specify for individual tablespaces and any `NOLOGGING` settings you specify for individual database objects.

`FORCE LOGGING` mode is persistent across instances of the database. If you shut down and restart the database, then the database is still in `FORCE LOGGING` mode. However, if you re-create the control file, then Oracle Database will take the database out of `FORCE LOGGING` mode unless you specify `FORCE LOGGING` in the `CREATE CONTROLFILE` statement.

> **Note:**
>
> `FORCE LOGGING` mode can have performance effects. Refer to *Oracle Database Administrator's Guide* for information on when to use this setting.

> **See Also:**
>
> CREATE CONTROLFILE

tablespace_clauses

Use the tablespace clauses to configure the `SYSTEM` and `SYSAUX` tablespaces and to specify a default temporary tablespace and an undo tablespace.

extent_management_clause

Use this clause to create a locally managed `SYSTEM` tablespace. If you omit this clause, then the `SYSTEM` tablespace will be dictionary managed.

> **Note:**
>
> When you create a locally managed `SYSTEM` tablespace, you cannot change it to be dictionary managed, nor can you create any other dictionary-managed tablespaces in this database.

If you specify this clause, then the database must have a default temporary tablespace, because a locally managed `SYSTEM` tablespace cannot store temporary segments.

- If you specify `EXTENT MANAGEMENT LOCAL` but you do not specify the `DATAFILE` clause, then you can omit the *`default_temp_tablespace`* clause. Oracle Database will create a default temporary tablespace called `TEMP` with one data file of size 10M with autoextend disabled.
- If you specify both `EXTENT MANAGEMENT LOCAL` and the `DATAFILE` clause, then you must also specify the *`default_temp_tablespace`* clause and explicitly specify a temp file for that temporary tablespace.

If you have opened the instance in automatic undo mode, similar requirements exist for the database undo tablespace:

- If you specify `EXTENT MANAGEMENT LOCAL` but you do not specify the `DATAFILE` clause, then you can omit the *`undo_tablespace`* clause. Oracle Database will create an undo tablespace named `SYS_UNDOTBS`.
- If you specify both `EXTENT MANAGEMENT LOCAL` and the `DATAFILE` clause, then you must also specify the *`undo_tablespace`* clause and explicitly specify a data file for that tablespace.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on locally managed and dictionary-managed tablespaces

DATAFILE Clause

Specify one or more files to be used as data files. All these files become part of the `SYSTEM` tablespace. Use the data *`file_tempfile_spec`* form of *`file_specification`* to create regular data files and temp files in an operating system file system or to create Oracle ASM disk group files.

> **Note:**
>
> This clause is optional, as is the `DATAFILE` clause of the *`undo_tablespace`* clause. Therefore, to avoid ambiguity, if your intention is to specify a data file for the `SYSTEM` tablespace with this clause, then do *not* specify it immediately after an *`undo_tablespace`* clause that does not include the optional `DATAFILE` clause. If you do so, then Oracle Database will interpret the `DATAFILE` clause to be part of the *`undo_tablespace`* clause.

The syntax for specifying data files for the `SYSTEM` tablespace is the same as that for specifying data files during tablespace creation using the `CREATE TABLESPACE` statement, whether you are storing files using Oracle ASM or in a file system.

> **See Also:**
>
> CREATE TABLESPACE for information on specifying data files

If you are running the database in automatic undo mode and you specify a data file name for the `SYSTEM` tablespace, then Oracle Database expects to generate data files for all tablespaces. Oracle Database does this automatically if you are using Oracle Managed Files—you have set a value for the `DB_CREATE_FILE_DEST` initialization parameter. However, if you are not using Oracle Managed Files and you specify this clause, then you must also specify the *`undo_tablespace`* clause and the *`default_temp_tablespace`* clause.

If you omit this clause, then:

- If the `DB_CREATE_FILE_DEST` initialization parameter is set, then Oracle Database creates a 100 MB Oracle-managed data file with a system-generated name in the default file destination specified in the parameter.
- If the `DB_CREATE_FILE_DEST` initialization parameter is not set, then Oracle Database creates one data file whose name and size depend on your operating system.

> **See Also:**
>
> *file_specification* for syntax

SYSAUX Clause

Oracle Database creates both the `SYSTEM` and `SYSAUX` tablespaces as part of every database. Use this clause if you are not using Oracle Managed Files and you want to specify one or more data files for the `SYSAUX` tablespace.

You must specify this clause if you have specified one or more data files for the `SYSTEM` tablespace using the `DATAFILE` clause. If you are using Oracle Managed Files and you omit this clause, then the database creates the SYSAUX data files in the default location set up for Oracle Managed Files.

If you have enabled Oracle Managed Files and you omit the SYSAUX clause, then the database creates the SYSAUX tablespace as an online, permanent, locally managed tablespace with one data file of 100 MB, with logging enabled and automatic segment-space management.

The syntax for specifying data files for the SYSAUX tablespace is the same as that for specifying data files during tablespace creation using the CREATE TABLESPACE statement, whether you are storing files using Oracle ASM or in a file system.

> **See Also:**
>
> - CREATE TABLESPACE for information on creating the SYSAUX tablespace during database upgrade and for information on specifying data files in a tablespace
> - *Oracle Database Administrator's Guide* for more information on creating the SYSAUX tablespace

default_tablespace

Specify this clause to create a default permanent tablespace for the database. Oracle Database creates a smallfile tablespace and subsequently will assign to this tablespace any non-SYSTEM users for whom you do not specify a different permanent tablespace. If you do not specify this clause, then the SYSTEM tablespace is the default permanent tablespace for non-SYSTEM users.

The DATAFILE clause and *extent_management_clause* have the same semantics they have in a CREATE TABLESPACE statement. Refer to "DATAFILE | TEMPFILE Clause" and *extent_management_clause* for information on these clauses.

default_temp_tablespace

Use this clause to create a default shared temporary tablespace or a default local temporary tablespace. Oracle Database will assign to these temporary tablespaces any users for whom you do not specify different temporary tablespaces.

- Specify DEFAULT TEMPORARY TABLESPACE to create a default shared temporary tablespace for the database. Shared temporary tablespaces were available in prior releases of Oracle Database and were called "temporary tablespaces." Elsewhere in this guide, the term "temporary tablespace" refers to a shared temporary tablespace unless specified otherwise. If you do not specify this clause, and if the database does not create a default shared temporary tablespace automatically in the process of creating a locally managed SYSTEM tablespace, then the SYSTEM tablespace is the default shared temporary tablespace.
- Starting with Oracle Database 12*c* Release 2 (12.2), you can specify DEFAULT LOCAL TEMPORARY TABLESPACE to create a default local temporary tablespace. Local temporary tablespaces are useful for Oracle Real Application Clusters and Oracle Flex Clusters. They store a separate, nonshared temp file for each database instance, which can improve I/O performance. A local temporary tablespace must be a BIGFILE tablespace.

  – Specify FOR ALL to instruct the database to create separate, nonshared temp files for all HUB and LEAF nodes.
  – Specify FOR LEAF to instruct the database to create separate nonshared temp files for only LEAF nodes. If you specify this clause, then HUB nodes will use the default shared temporary tablespace. For SQL operations that span both HUB and LEAF nodes, HUB nodes will use the default shared temporary tablespace and LEAF nodes will use the default local temporary tablespace.

  If you do not create a local temporary tablespace, then HUB and LEAF nodes will use the default shared temporary tablespace.

Specify BIGFILE or SMALLFILE to determine whether the default temporary tablespace is a bigfile or smallfile tablespace. These clauses have the same semantics as in the "SET DEFAULT TABLESPACE Clause ".

The TEMPFILE clause part of this clause is optional if you have enabled Oracle Managed Files by setting the DB_CREATE_FILE_DEST initialization parameter. If you have not specified a value for this parameter, then the TEMPFILE clause is required. If you have specified BIGFILE, then you can specify only one temp file in this clause.

The syntax for specifying temp files for the default temporary tablespace is the same as that for specifying temp files during temporary tablespace creation using the CREATE TABLESPACE statement, whether you are storing files using Oracle ASM or in a file system.

The *extent_management_clause* clause has the same semantics in CREATE DATABASE and CREATE TABLESPACE statements. For complete information, refer to the CREATE TABLESPACE ... *extent_management_clause* .

See Also:

CREATE TABLESPACE for information on specifying temp files

Note:

On some operating systems, Oracle does not allocate space for a temp file until the temp file blocks are actually accessed. This delay in space allocation results in faster creation and resizing of temp files, but it requires that sufficient disk space is available when the temp files are later used. To avoid potential problems, before you create or resize a temp file, ensure that the available disk space exceeds the size of the new temp file or the increased size of a resized temp file. The excess space should allow for anticipated increases in disk space use by unrelated operations as well. Then proceed with the creation or resizing operation.

Restrictions on Default Temporary Tablespaces

Default temporary tablespaces are subject to the following restrictions:

- You cannot specify the SYSTEM tablespace in this clause.
- The default temporary tablespace must have a standard block size.

undo_tablespace

If you have opened the instance in automatic undo mode (the `UNDO_MANAGEMENT` initialization parameter is set to `AUTO`, which is the default), then you can specify the *`undo_tablespace`* to create a tablespace to be used for undo data. Oracle strongly recommends that you use automatic undo mode. However, if you want undo space management to be handled by way of rollback segments, then you must omit this clause. You can also omit this clause if you have set a value for the `UNDO_TABLESPACE` initialization parameter. If that parameter has been set, and if you specify this clause, then *`tablespace`* must be the same as that parameter value.

- Specify `BIGFILE` if you want the undo tablespace to be a bigfile tablespace. A **bigfile tablespace** contains only one data file, which can be up to 8 exabytes (8 million terabytes) in size.
- Specify `SMALLFILE` if you want the undo tablespace to be a smallfile tablespace. A **smallfile tablespace** is a traditional Oracle Database tablespace, which can contain 1022 data files or temp files, each of which can contain up to approximately 4 million (2^{22}) blocks.
- The `DATAFILE` clause part of this clause is optional if you have enabled Oracle Managed Files by setting the `DB_CREATE_FILE_DEST` initialization parameter. If you have not specified a value for this parameter, then the `DATAFILE` clause is required. If you have specified `BIGFILE`, then you can specify only one data file in this clause.

The syntax for specifying data files for the undo tablespace is the same as that for specifying data files during tablespace creation using the `CREATE TABLESPACE` statement, whether you are storing files using Oracle ASM or in a file system.

> **See Also:**
>
> CREATE TABLESPACE for information on specifying data files

If you specify this clause, then Oracle Database creates an undo tablespace named *`tablespace`*, creates the specified data file(s) as part of the undo tablespace, and assigns this tablespace as the undo tablespace of the instance. Oracle Database will manage undo data using this undo tablespace. The `DATAFILE` clause of this clause has the same behavior as described in "DATAFILE Clause".

If you have specified a value for the `UNDO_TABLESPACE` initialization parameter in your initialization parameter file before mounting the database, then you must specify the same name in this clause. If these names differ, then Oracle Database will return an error when you open the database.

If you omit this clause, then Oracle Database creates a default database with a default smallfile undo tablespace named `SYS_UNDOTBS` and assigns this default tablespace as the undo tablespace of the instance. This undo tablespace allocates disk space from the default files used by the `CREATE DATABASE` statement, and it has an initial extent of 10M. Oracle Database handles the system-generated data file as described in "DATAFILE Clause". If Oracle Database is unable to create the undo tablespace, then the entire `CREATE DATABASE` operation fails.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces
> - CREATE TABLESPACE for information on creating an undo tablespace after database creation

set_time_zone_clause

Use the `SET TIME_ZONE` clause to set the time zone of the database. You can specify the time zone in two ways:

- By specifying a displacement from UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The valid range of *`hh:mi`* is -12:00 to +14:00.
- By specifying a time zone region. To see a listing of valid time zone region names, query the `TZNAME` column of the `V$TIMEZONE_NAMES` dynamic performance view.

> **Note:**
>
> Oracle recommends that you set the database time zone to UTC (`0:00`). Doing so can improve performance, especially across databases, as no conversion of time zones will be required.

> **See Also:**
>
> *Oracle Database Reference* for information on the dynamic performance views

Oracle Database normalizes all `TIMESTAMP WITH LOCAL TIME ZONE` data to the time zone of the database when the data is stored on disk. If you do not specify the `SET TIME_ZONE` clause, then the database uses the operating system time zone of the server. If the operating system time zone is not a valid Oracle Database time zone, then the database time zone defaults to UTC.

USER_DATA TABLESPACE Clause

This clause lets you create a tablespace that is used for storing user data and database options such as Oracle XML DB.

- If you specify this clause when creating a multitenant container database (CDB), then the tablespace is created as part of the seed. Pluggable databases (PDBs) subsequently created using the seed will include this tablespace and its data file. The tablespace and data file specified in this clause are not used by the root.
- If you specify this clause when creating a non-CDB, then the tablespace is created as part of the non-CDB.

Specify `BIGFILE` or `SMALLFILE` to determine whether the tablespace is a bigfile or smallfile tablespace. If you omit these clauses, then Oracle Database creates a tablespace of the type that you specify with the `SET DEFAULT TABLESPACE` clause. If you do not specify the `SET DEFAULT TABLESPACE` clause, then Oracle Database creates a smallfile tablespace. These clauses have the same semantics as in the "SET DEFAULT TABLESPACE Clause ".

Use the *`datafile_tempfile_spec`* clause to specify one or more data files for the tablespace. Refer to datafile_tempfile_spec for the full semantics of this clause.

enable_pluggable_database

Specify this clause to create a CDB. Before issuing the `CREATE DATABASE` statement, you must set the `ENABLE_PLUGGABLE_DATABASE` initialization parameter to `TRUE`. The `CREATE DATABASE` statement will create a CDB that contains a root and a seed container. You can then create PDBs in the CDB by using the `CREATE PLUGGABLE DATABASE` statement. If you omit the *`enable_pluggable_database`* clause, then a non-CDB is created and it can never contain any containers.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for the complete steps for creating a CDB
> - CREATE PLUGGABLE DATABASE
> - "Creating a CDB: Example"

file_name_convert

Use the *`file_name_convert`* clause to determine how the database generates the names of files (such as data files and wallet files) associated with the seed by using the names of files associated with the root.

- For *`filename_pattern`*, specify a string found in file names associated with the root.
- For *`replacement_filename_pattern`*, specify a replacement string.

Oracle Database will replace *`filename_pattern`* with *`replacement_filename_pattern`* when generating the names of files associated with the seed.

File name patterns cannot match files or directories managed by Oracle Managed Files.

You can specify `FILE_NAME_CONVERT = NONE`, which is the same as omitting this clause. If you omit this clause, then the database first attempts to use Oracle Managed Files to generate seed file names. If you are not using Oracle Managed Files, then the database uses the `PDB_FILE_NAME_CONVERT` initialization parameter to generate file names. If this parameter is not set, then an error occurs.

tablespace_datafile_clauses

Use these clauses to specify attributes for all data files comprising the `SYSTEM` and `SYSAUX` tablespaces in the seed PDB. If you do not specify `SIZE` *`size_clause`*, then the data file size for a given tablespace will be set to a predetermined fraction of the size of the corresponding root data file. If you do not specify the *`autoextend_clause`*, then those values are inherited from the root.

Refer to *size_clause* and *autoextend_clause* for the full semantics of these clauses.

undo_mode_clause

This clause lets you specify local undo mode or shared undo mode for the CDB.

- Use `LOCAL UNDO ON` to specify local undo mode for the CDB. In this mode, every container in the CDB uses local undo.
- Use `LOCAL UNDO OFF` to specify shared undo mode for the CDB. In this mode, there is one active undo tablespace for a single-instance CDB, or for an Oracle RAC CDB, there is one active undo tablespace for each instance.

If you omit this clause, then the default is `LOCAL UNDO OFF`.

Examples

Creating a Database: Example

The following statement creates a database and fully specifies each argument:

```
CREATE DATABASE sample
   CONTROLFILE REUSE
   LOGFILE
      GROUP 1 ('diskx:log1.log', 'disky:log1.log') SIZE 50K,
      GROUP 2 ('diskx:log2.log', 'disky:log2.log') SIZE 50K
   MAXLOGFILES 5
   MAXLOGHISTORY 100
   MAXDATAFILES 10
   MAXINSTANCES 2
   ARCHIVELOG
   CHARACTER SET AL32UTF8
   NATIONAL CHARACTER SET AL16UTF16
   DATAFILE
      'disk1:df1.dbf' AUTOEXTEND ON,
      'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
   DEFAULT TEMPORARY TABLESPACE temp_ts
   UNDO TABLESPACE undo_ts
   SET TIME_ZONE = '+02:00';
```

This example assumes that you have enabled Oracle Managed Files by specifying a value for the `DB_CREATE_FILE_DEST` parameter in your initialization parameter file. Therefore no file specification is needed for the `DEFAULT TEMPORARY TABLESPACE` and `UNDO TABLESPACE` clauses.

Creating a CDB: Example

The following statement creates a CDB `newcdb`. The `ENABLE PLUGGABLE DATABASE` clause indicates that a CDB is being created. The CDB will contain a root (`CDB$ROOT`) and a seed (`PDB$SEED`). The `FILE_NAME_CONVERT` clause specifies that names of files for the seed will be generated by replacing `/u01/app/oracle/oradata/newcdb` in the names of files associated with the root with `/u01/app/oracle/oradata/pdbseed`.

```
CREATE DATABASE newcdb
  USER SYS IDENTIFIED BY sys_password
  USER SYSTEM IDENTIFIED BY system_password
  LOGFILE GROUP 1 ('/u01/logs/my/redo01a.log','/u02/logs/my/redo01b.log')
             SIZE 100M BLOCKSIZE 512,
          GROUP 2 ('/u01/logs/my/redo02a.log','/u02/logs/my/redo02b.log')
             SIZE 100M BLOCKSIZE 512,
          GROUP 3 ('/u01/logs/my/redo03a.log','/u02/logs/my/redo03b.log')
```

```
           SIZE 100M BLOCKSIZE 512
MAXLOGHISTORY 1
MAXLOGFILES 16
MAXLOGMEMBERS 3
MAXDATAFILES 1024
CHARACTER SET AL32UTF8
NATIONAL CHARACTER SET AL16UTF16
EXTENT MANAGEMENT LOCAL
DATAFILE '/u01/app/oracle/oradata/newcdb/system01.dbf'
  SIZE 700M REUSE AUTOEXTEND ON NEXT 10240K MAXSIZE UNLIMITED
SYSAUX DATAFILE '/u01/app/oracle/oradata/newcdb/sysaux01.dbf'
  SIZE 550M REUSE AUTOEXTEND ON NEXT 10240K MAXSIZE UNLIMITED
DEFAULT TABLESPACE deftbs
  DATAFILE '/u01/app/oracle/oradata/newcdb/deftbs01.dbf'
  SIZE 500M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED
DEFAULT TEMPORARY TABLESPACE tempts1
  TEMPFILE '/u01/app/oracle/oradata/newcdb/temp01.dbf'
  SIZE 20M REUSE AUTOEXTEND ON NEXT 640K MAXSIZE UNLIMITED
UNDO TABLESPACE undotbs1
  DATAFILE '/u01/app/oracle/oradata/newcdb/undotbs01.dbf'
  SIZE 200M REUSE AUTOEXTEND ON NEXT 5120K MAXSIZE UNLIMITED
ENABLE PLUGGABLE DATABASE
  SEED
  FILE_NAME_CONVERT = ('/u01/app/oracle/oradata/newcdb/',
                       '/u01/app/oracle/oradata/pdbseed/')
  SYSTEM DATAFILES SIZE 125M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
  SYSAUX DATAFILES SIZE 100M
USER_DATA TABLESPACE usertbs
  DATAFILE '/u01/app/oracle/oradata/pdbseed/usertbs01.dbf'
  SIZE 200M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

CREATE DATABASE LINK

Purpose

Use the `CREATE DATABASE LINK` statement to create a database link. A **database link** is a schema object in one database that enables you to access objects on another database. The other database need not be an Oracle Database system. However, to access non-Oracle systems you must use Oracle Heterogeneous Services.

After you have created a database link, you can use it in SQL statements to refer to tables, views, and PL/SQL objects in the other database by appending `@`*`dblink`* to the table, view, or PL/SQL object name. You can query a table or view in the other database with the `SELECT` statement. You can also access remote tables and views using any `INSERT`, `UPDATE`, `DELETE`, or `LOCK TABLE` statement.

See Also:

- *Oracle Database Development Guide* for information about accessing remote tables or views with PL/SQL functions, procedures, packages, and data types
- *Oracle Database Administrator's Guide* for information on distributed database systems
- *Oracle Database Reference* for descriptions of existing database links in the `ALL_DB_LINKS`, `DBA_DB_LINKS`, and `USER_DB_LINKS` data dictionary views and for information on monitoring the performance of existing links through the `V$DBLINK` dynamic performance view
- ALTER DATABASE LINK for information on altering a database link when the password of a connection or authentication user changes.
- DROP DATABASE LINK for information on dropping existing database links
- INSERT , UPDATE , DELETE , and LOCK TABLE for using links in DML operations

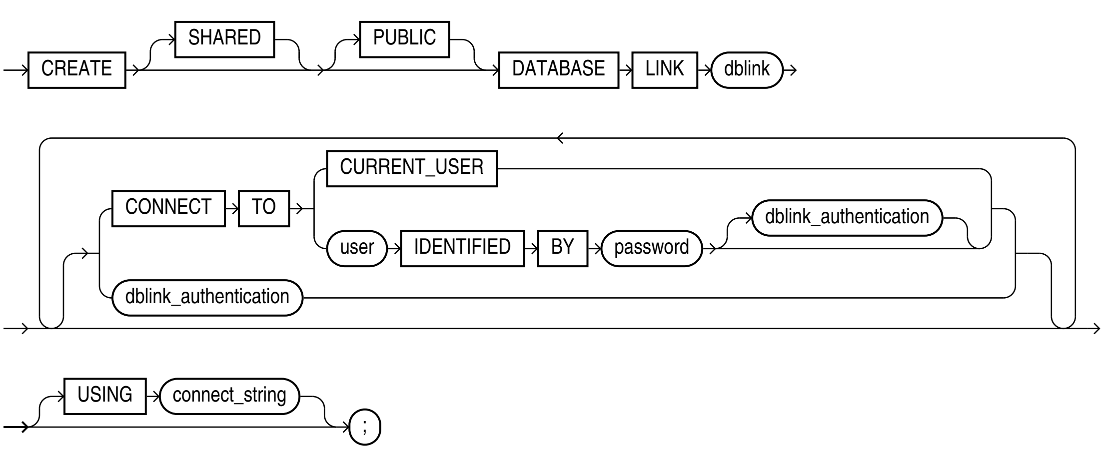
Prerequisites

To create a private database link, you must have the `CREATE DATABASE LINK` system privilege. To create a public database link, you must have the `CREATE PUBLIC DATABASE LINK` system privilege. Also, you must have the `CREATE SESSION` system privilege on the remote Oracle Database.

Oracle Net must be installed on both the local and remote Oracle Databases.

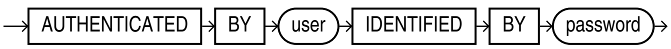
Syntax

***create_database_link*::=**



(*dblink*::=)

***dblink_authentication*::=**



Semantics

SHARED

Specify `SHARED` to create a database link that can be shared by multiple sessions using a single network connection from the source database to the target database. In a shared server configuration, shared database links can keep the number of connections into the remote database from becoming too large. Shared links are typically also public database links. However, a shared private database link can be useful when many clients access the same local schema, and therefore use the same private database link.

In a shared database link, multiple sessions in the source database share the same connection to the target database. Once a session is established on the target database, that session is disassociated from the connection, to make the connection available to another session on the source database. To prevent an unauthorized session from attempting to connect through the database link, when you specify `SHARED` you must also specify the *`dblink_authentication`* clause for the users authorized to use the database link.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about shared database links

PUBLIC

Specify `PUBLIC` to create a public database link visible to all users. If you omit this clause, then the database link is private and is available only to you.

The data accessible on the remote database depends on the identity the database link uses when connecting to the remote database:

- If you specify `CONNECT TO` *`user`* `IDENTIFIED BY` *`password`*, then the database link connects with the specified user and password.
- If you specify `CONNECT TO CURRENT_USER`, then the database link connects with the user in effect based on the scope in which the link is used.
- If you omit both of those clauses, then the database link connects to the remote database as the locally connected user.

> **See Also:**
>
> "Defining a Public Database Link: Example"

dblink

Specify the complete or partial name of the database link. If you specify only the database name, then Oracle Database implicitly appends the database domain of the local database.

Use only ASCII characters for *dblink*. Multibyte characters are not supported. The database link name is case insensitive and is stored in uppercase ASCII characters. If you specify the database name as a quoted identifier, then the quotation marks are silently ignored.

If the value of the GLOBAL_NAMES initialization parameter is TRUE, then the database link must have the same name as the database to which it connects. If the value of GLOBAL_NAMES is FALSE, and if you have changed the global name of the database, then you can specify the global name.

The maximum number of database links that can be open in one session or one instance of an Oracle RAC configuration depends on the value of the OPEN_LINKS and OPEN_LINKS_PER_INSTANCE initialization parameters.

Restriction on Creating Database Links

You cannot create a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema. Periods are permitted in names of database links, so Oracle Database interprets the entire name, such as ralph.linktosales, as the name of a database link in your schema rather than as a database link named linktosales in the schema ralph.

> **See Also:**
>
> - "References to Objects in Remote Databases " for guidelines for naming database links
> - *Oracle Database Reference* for information on the GLOBAL_NAMES, OPEN_LINKS, and OPEN_LINKS_PER_INSTANCE initialization parameters
> - "RENAME GLOBAL_NAME Clause" (an ALTER DATABASE clause) for information on changing the database global name

CONNECT TO Clause

The CONNECT TO clause lets you specify the user and credentials, if any, to be used to connect to the remote database.

CURRENT_USER Clause

Specify CURRENT_USER to create a **current user database link**. The current user must be a global user with a valid account on the remote database.

If the database link is used directly rather than from within a stored object, then the current user is the same as the connected user.

When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, CURRENT_USER is the name of the user that owns the stored object, and not the name of the user that called the object. For example, if the database link appears inside procedure `scott.p` (created by `scott`), and user `jane` calls procedure `scott.p`, then the current user is `scott`.

However, if the stored object is an invoker-rights function, procedure, or package, then the invoker's authorization ID is used to connect as a remote user. For example, if the privileged database link appears inside procedure `scott.p` (an invoker-rights procedure created by `scott`), and user Jane calls procedure `scott.p`, then `CURRENT_USER` is `jane` and the procedure executes with Jane's privileges.

> **See Also:**
>
> - CREATE FUNCTION for more information on invoker-rights functions
> - "Defining a CURRENT_USER Database Link: Example"

user IDENTIFIED BY _password_

Specify the user name and password used to connect to the remote database using a **fixed user database link**. If you omit this clause, then the database link uses the user name and password of each user who is connected to the database. This is called a **connected user database link**.

> **See Also:**
>
> "Defining a Fixed-User Database Link: Example"

dblink_authentication

You can specify this clause only if you are creating a shared database link—that is, you have specified the `SHARED` clause. Specify the username and password on the target instance. This clause authenticates the user to the remote server and is required for security. The specified username and password must be a valid username and password on the remote instance. The username and password are used only for authentication. No other operations are performed on behalf of this user.

USING '_connect string_'

Specify the service name of a remote database. If you specify only the database name, then Oracle Database implicitly appends the database domain to the connect string to create a complete service name. Therefore, if the database domain of the remote database is different from that of the current database, then you must specify the complete service name.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on specifying remote databases

Examples

The examples that follow assume two databases, one with the database name `local` and the other with the database name `remote`. The examples use the Oracle Database domain. Your database domain will be different.

Defining a Public Database Link: Example

The following statement defines a shared public database link named `remote` that refers to the database specified by the service name `remote`:

```
CREATE PUBLIC DATABASE LINK remote
   USING 'remote';
```

This database link allows user `hr` on the `local` database to update a table on the `remote` database (assuming `hr` has appropriate privileges):

```
UPDATE employees@remote
   SET salary=salary*1.1
   WHERE last_name = 'Baer';
```

Defining a Fixed-User Database Link: Example

In the following statement, user `hr` on the `remote` database defines a fixed-user database link named `local` to the `hr` schema on the `local` database:

```
CREATE DATABASE LINK local
   CONNECT TO hr IDENTIFIED BY password
   USING 'local';
```

After this database link is created, `hr` can query tables in the schema `hr` on the `local` database in this manner:

```
SELECT * FROM employees@local;
```

User `hr` can also use DML statements to modify data on the `local` database:

```
INSERT INTO employees@local
   (employee_id, last_name, email, hire_date, job_id)
   VALUES (999, 'Claus', 'sclaus@example.com', SYSDATE, 'SH_CLERK');

UPDATE jobs@local SET min_salary = 3000
   WHERE job_id = 'SH_CLERK';

DELETE FROM employees@local
   WHERE employee_id = 999;
```

Using this fixed database link, user hr on the `remote` database can also access tables owned by other users on the same database. This statement assumes that user `hr` has the `READ` or `SELECT` privilege on the `oe.customers` table. The statement connects to the user `hr` on the `local` database and then queries the `oe.customers` table:

```
SELECT * FROM oe.customers@local;
```

Defining a CURRENT_USER Database Link: Example

The following statement defines a current-user database link to the `remote` database, using the entire service name as the link name:

```
CREATE DATABASE LINK remote.us.example.com
   CONNECT TO CURRENT_USER
   USING 'remote';
```

The user who issues this statement must be a global user registered with the LDAP directory service.

You can create a synonym to hide the fact that a particular table is on the `remote` database. The following statement causes all future references to `emp_table` to access the `employees` table owned by `hr` on the `remote` database:

```
CREATE SYNONYM emp_table
   FOR oe.employees@remote.us.example.com;
```

CREATE DIMENSION

Purpose

Use the `CREATE DIMENSION` statement to create a **dimension**. A dimension defines a parent-child relationship between pairs of column sets, where all the columns of a column set must come from the same table. However, columns in one column set (called a **level**) can come from a different table than columns in another set. The optimizer uses these relationships with materialized views to perform **query rewrite**. The SQL Access Advisor uses these relationships to recommend creation of specific materialized views.

> **Note:**
>
> Oracle Database does not automatically validate the relationships you declare when creating a dimension. To validate the relationships specified in the *`hierarchy_clause`* and the *`dimension_join_clause`* of `CREATE DIMENSION`, you must run the `DBMS_OLAP.VALIDATE_DIMENSION` procedure.

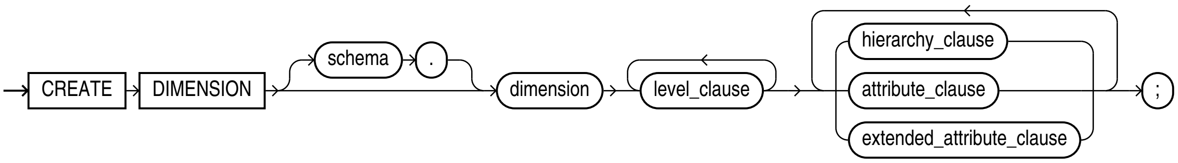
> **See Also:**
>
> - CREATE MATERIALIZED VIEW for more information on materialized views
> - *Oracle Database SQL Tuning Guide* for more information on query rewrite, the optimizer and the SQL Access Advisor

Prerequisites

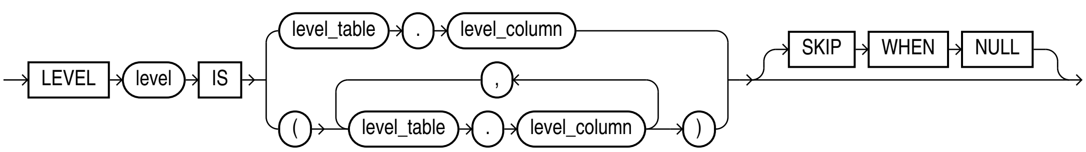
To create a dimension in your own schema, you must have the `CREATE DIMENSION` system privilege. To create a dimension in another user's schema, you must have the `CREATE ANY DIMENSION` system privilege. In either case, you must have the `READ` or `SELECT` object privilege on any objects referenced in the dimension.

Syntax

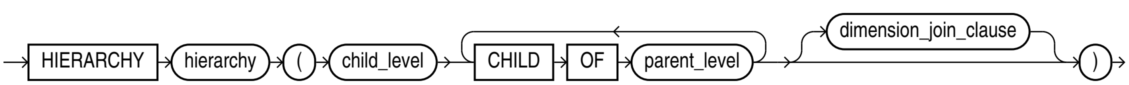
***create_dimension*::=**



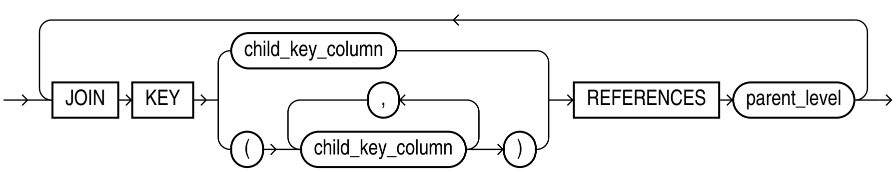
***level_clause*::=**



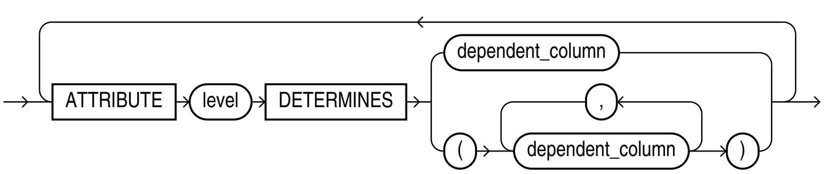
***hierarchy_clause*::=**



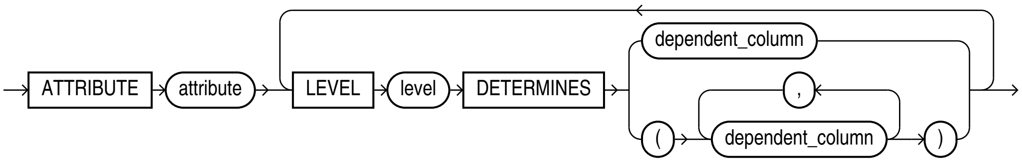
***dimension_join_clause*::=**



***attribute_clause*::=**



***extended_attribute_clause*::=**



Semantics

schema

Specify the schema in which the dimension will be created. If you do not specify *`schema`*, then Oracle Database creates the dimension in your own schema.

dimension

Specify the name of the dimension. The name must satisfy the requirements listed in "Database Object Naming Rules ".

level_clause

The *`level_clause`* defines a level in the dimension. A level defines dimension hierarchies and attributes.

level

Specify the name of the level.

level_table . level_column

Specify the columns in the level. You can specify up to 32 columns. The tables you specify in this clause must already exist.

SKIP WHEN NULL

Specify this clause to indicate that if the specified level is `NULL`, then the level is to be skipped. This clause lets you preserve the hierarchical chain of parent-child relationship by an alternative path that skips over the specified level. See *hierarchy_clause*.

Restrictions on Dimension Level Columns

Dimension level columns are subject to the following restrictions:

- All of the columns in a level must come from the same table.
- If columns in different levels come from different tables, then you must specify the *`dimension_join_clause`*.
- The set of columns you specify must be unique to this level.
- The columns you specify cannot be specified in any other dimension.
- Each `level_column` must be non-null unless the level is specified with `SKIP WHEN NULL`. The non-null columns need not have `NOT NULL` constraints. The column for which you specify `SKIP WHEN NULL` cannot have a `NOT NULL` constraint).

hierarchy_clause

The *hierarchy_clause* defines a linear hierarchy of levels in the dimension. Each hierarchy forms a chain of parent-child relationships among the levels in the dimension. Hierarchies in a dimension are independent of each other. They may, but need not, have columns in common.

Each level in the dimension should be specified at most once in this clause, and each level must already have been named in the *level_clause.*

hierarchy

Specify the name of the hierarchy. This name must be unique in the dimension.

child_level

Specify the name of a level that has an n:1 relationship with a parent level. The *level_columns* of *child_level* cannot be null, and each *child_level* value uniquely determines the value of the next named *parent_level*.

If the child *level_table* is different from the parent *level_table*, then you must specify a join relationship between them in the *dimension_join_clause*.

parent_level

Specify the name of a level.

dimension_join_clause

The *dimension_join_clause* lets you specify an inner equijoin relationship for a dimension whose columns are contained in multiple tables. This clause is required and permitted only when the columns specified in the hierarchy are not all in the same table.

child_key_column

Specify one or more columns that are join-compatible with columns in the parent level.

If you do not specify the schema and table of each *child_column*, then the schema and table are inferred from the CHILD OF relationship in the *hierarchy_clause*. If you do specify the schema and column of a *child_key_column*, then the schema and table must match the schema and table of columns in the child of *parent_level* in the *hierarchy_clause*.

parent_level

Specify the name of a level.

Restrictions on Join Dimensions

Join dimensions are subject to the following restrictions:

- You can specify only one *dimension_join_clause* for a given pair of levels in the same hierarchy.
- The *child_key_columns* must be non-null, and the parent key must be unique and non-null. You need not define constraints to enforce these conditions, but queries may return incorrect results if these conditions are not true.

- Each child key must join with a key in the *parent_level* table.
- Self-joins are not permitted. The *child_key_columns* cannot be in the same table as *parent_level*.
- All of the child key columns must come from the same table.
- The number of child key columns must match the number of columns in *parent_level*, and the columns must be joinable.
- You cannot specify multiple child key columns unless the parent level consists of multiple columns.

attribute_clause

The *attribute_clause* lets you specify the columns that are uniquely determined by a hierarchy level. The columns in *level* must all come from the same table as the *dependent_columns*. The *dependent_columns* need not have been specified in the *level_clause*.

For example, if the hierarchy levels are `city`, `state`, and `country`, then `city` might determine `mayor`, `state` might determine `governor`, and `country` might determine `president`.

extended_attribute_clause

This clause lets you specify an attribute name for one or more level-to-column relations. The type of attribute you create with this clause is not different from the type of attribute created using the *attribute_clause*. The only difference is that this clause lets you assign a name to the attribute that is different from the level name.

Examples

Creating a Dimension: Examples

This statement was used to create the `customers_dim` dimension in the sample schema `sh`:

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
;
```

Creating a Dimension with Extended Attributes: Example

Alternatively, the *extended_attribute_clause* could have been used instead of the *attribute_clause*, as shown in the following example:

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer_info LEVEL customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name);
```

Creating a Dimension with NULL Column Values: Example

The following example shows how to create the dimension if one of the level columns is null and you want to preserve the hierarchical chain. The example uses the `cust_marital_status` column for simplicity because it is not a `NOT NULL` column. If it had such a constraint, then you would have to disable the constraint before using the `SKIP WHEN NULL` clause.

```
CREATE DIMENSION customers_dim
   LEVEL customer IS (customers.cust_id)
   LEVEL status IS (customers.cust_marital_status) SKIP WHEN NULL
   LEVEL city IS (customers.cust_city)
   LEVEL state IS (customers.cust_state_province)
   LEVEL country IS (countries.country_id)
   LEVEL subregion IS (countries.country_subregion) SKIP WHEN NULL
   LEVEL region IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer CHILD OF
      city CHILD OF
      state CHILD OF
      country CHILD OF
      subregion CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
;
```

CREATE DIRECTORY

Purpose

Use the `CREATE DIRECTORY` statement to create a directory object. A directory object specifies an alias for a directory on the server file system where external binary file LOBs (`BFILE`s) and external table data are located. You can use directory names when referring to `BFILE`s in your PL/SQL code and OCI calls, rather than hard coding the operating system path name, for management flexibility.

All directories are created in a single namespace and are not owned by an individual schema. You can secure access to the `BFILE`s stored within the directory structure by granting object privileges on the directories to specific users.

> **See Also:**
>
> - "Large Object (LOB) Data Types " for more information on `BFILE` objects
> - GRANT for more information on granting object privileges
> - *external_table_clause*::= of `CREATE TABLE`

Prerequisites

You must have the `CREATE ANY DIRECTORY` system privilege to create directories.

When you create a directory, you are automatically granted the `READ`, `WRITE`, and `EXECUTE` object privileges on the directory, and you can grant these privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

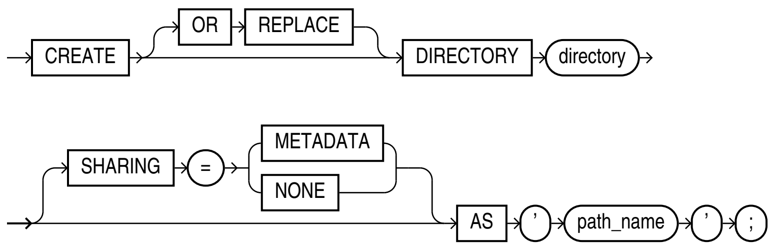
`WRITE` privileges on a directory are useful in connection with external tables. They let the grantee determine whether the external table agent can write a log file or a bad file to the directory.

For file storage, you must also create a corresponding operating system directory, an Oracle Automatic Storage Management (Oracle ASM) disk group, or a directory within an Oracle ASM disk group. Your system or database administrator must ensure that the operating system directory has the correct read and write permissions for Oracle Database processes.

Privileges granted for the directory are created independently of the permissions defined for the operating system directory, and the two may or may not correspond exactly. For example, an error occurs if sample user `hr` is granted `READ` privilege on the directory object but the corresponding operating system directory does not have `READ` permission defined for Oracle Database processes.

Syntax

***create_directory*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory.

Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges.

> **See Also:**
>
> DROP DIRECTORY for information on removing a directory from the database

SHARING

This clause applies only when creating a directory in an application root. This type of directory is called an application common object and it can be shared with the application PDBs that belong to the application root. To determine how the directory is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the directory's metadata, but its data is unique to each container. This type of directory is referred to as a **metadata-linked application common object**.
- `NONE` - The directory is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the directory. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a directory after it is created.

See Also:

- *Oracle Database Reference* for more information on the `DEFAULT_SHARING` initialization parameter
- *Oracle Database Administrator's Guide* for complete information on creating application common objects

directory

Specify the name of the directory object to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

Oracle Database does not verify that the directory you specify actually exists. Therefore, take care that you specify a valid directory in your operating system. In addition, if your operating system uses case-sensitive path names, then be sure you specify the directory in the correct format. You need not include a trailing slash at the end of the path name.

Do not refer to a parent directory in the directory name. For example, the following syntax is valid:

```
CREATE DIRECTORY mydir AS '/scratch/data/file_data';
```

However, the following syntax is not valid:

```
CREATE DIRECTORY mydir AS '/scratch/../file_data';
```

path_name

Specify the full path name of the operating system directory of the server where the files are located. The single quotation marks are required, with the result that the path name is case sensitive.

Examples

Creating a Directory: Examples

The following statement creates a directory database object that points to a directory on the server:

```
CREATE DIRECTORY admin AS '/disk1/oracle/admin';
```

The following statement redefines directory database object `bfile_dir` to enable access to `BFILE`s stored in the operating system directory /`usr/bin/bfile_dir`:

```
CREATE OR REPLACE DIRECTORY bfile_dir AS '/usr/bin/bfile_dir';
```

CREATE DISKGROUP

Note:

This SQL statement is valid only if you are using Oracle ASM and you have started an Oracle ASM instance. You must issue this statement from within the Oracle ASM instance, not from a normal database instance. For information on starting an Oracle ASM instance, refer to *Oracle Automatic Storage Management Administrator's Guide*.

Purpose

Use the `CREATE DISKGROUP` clause to name a group of disks and specify that Oracle Database should manage the group for you. Oracle Database manages a disk group as a logical unit and evenly spreads each file across the disks to balance I/O. Oracle Database also automatically distributes database files across all available disks in disk groups and rebalances storage automatically whenever the storage configuration changes.

This statement creates a disk group, assigns one or more disks to the disk group, and mounts the disk group for the first time. Note that `CREATE DISKGROUP` only mounts a disk group on the local node. If you want Oracle ASM to mount the disk group automatically in subsequent instances, then you must add the disk group name to the value of the `ASM_DISKGROUPS` initialization parameter in the initialization parameter file. If you use an `SPFILE`, then the disk group is added to the initialization parameter automatically.

See Also:

- ALTER DISKGROUP for information on modifying disk groups
- *Oracle Automatic Storage Management Administrator's Guide* for information on Oracle ASM and using disk groups to simplify database administration
- `ASM_DISKGROUPS` for more information about adding disk group names to the initialization parameter file
- `V$ASM_OPERATION` for information on monitoring Oracle ASM operations
- DROP DISKGROUP for information on dropping a disk group

Prerequisites

You must have the `SYSASM` system privilege to issue this statement.

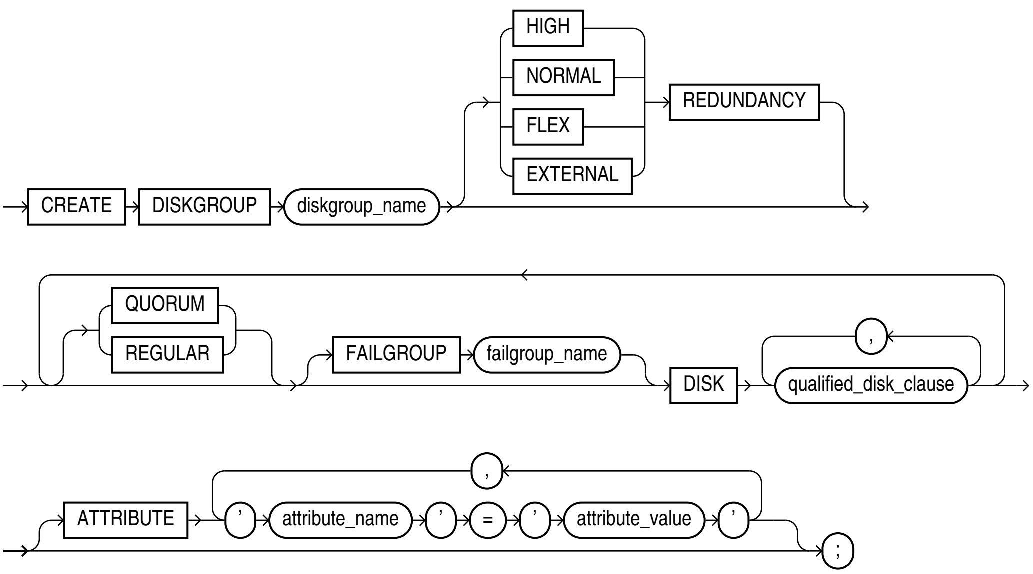
Before issuing this statement, you must format the disks using an operating system format utility. Also ensure that the Oracle Database user has read/write permission and the disks can be discovered using the `ASM_DISKSTRING`.

When you store your database files in Oracle ASM disk groups, rather than in a file system, before the database instance can access your files in the disk groups, you must configure and start up an Oracle ASM instance to manage the disk groups.

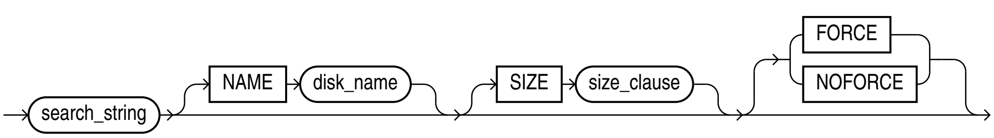
Each database instance communicates with a single Oracle ASM instance on the same node as the database. Multiple database instances on the same node can communicate with a single Oracle ASM instance.

Syntax

***create_diskgroup*::=**



***qualified_disk_clause*::=**



(*size_clause*::=)

diskgroup_name

Specify the name of the disk group. The name must satisfy the requirements listed in "Database Object Naming Rules ". However, disk groups are not schema objects.

> **Note:**
>
> Oracle does not recommend using quoted identifiers for disk group names. These quoted identifiers are accepted when issuing the `CREATE DISKGROUP` statement in SQL*Plus, but they may not be valid when using other tools that manage disk groups.

REDUNDANCY Clause

The REDUNDANCY clause lets you specify the redundancy level of the disk group.

- NORMAL REDUNDANCY requires the existence of at least two failure groups (see the FAILGROUP clause that follows). Oracle ASM provides redundancy for all files in the disk group according to the attributes specified in the disk group templates. NORMAL REDUNDANCY disk groups can tolerate the loss of one group. Refer to ALTER DISKGROUP ... *diskgroup_template_clauses* for more information on disk group templates.

  NORMAL REDUNDANCY is the default if you omit the REDUNDANCY clause. Therefore, if you omit this clause, you must create at least two failure groups, or the create operation will fail.

- HIGH REDUNDANCY requires the existence of at least three failure groups. Oracle ASM fixes mirroring at 3-way mirroring, with each extent getting two mirrored copies. HIGH REDUNDANCY disk groups can tolerate the loss of two failure groups.
- FLEX REDUNDANCY is a type of disk group that allows a database to specify its own redundancy after the disk group is created. A file's redundancy can also be changed after its creation. This type of disk group supports Oracle ASM file groups and quota groups. A flex disk group requires the existence of at least three failure groups. If a flex disk group has fewer than five failure groups, then it can tolerate the loss of one; otherwise, it can tolerate the loss of two failure groups. To create a flex disk group, the COMPATIBLE.ASM and COMPATIBLE.RDBMS disk group attributes must be set to 12.2 or greater.
- EXTERNAL REDUNDANCY indicates that Oracle ASM does not provide any redundancy for the disk group. The disks within the disk group must provide redundancy (for example, using a storage array), or you must be willing to tolerate loss of the disk group if a disk fails (for example, in a test environment). You cannot specify the FAILGROUP clause if you specify EXTERNAL REDUNDANCY.

You cannot change the redundancy level after the disk group has been created, with the following exception: You can convert a normal or high redundancy disk group to a flex disk group. For more information, see the *convert_redundancy_clause* of ALTER DISKGROUP.

QUORUM | REGULAR

Use these keywords to qualify either failure group or disk specifications.

- REGULAR disks, or disks in non-quorum failure groups, can contain any files.
- QUORUM disks, or disks in quorum failure groups, cannot contain any database files, the Oracle Cluster Registry (OCR), or dynamic volumes. However, QUORUM disks can contain the voting file for Cluster Synchronization Services (CSS). Oracle ASM uses quorum disks or disks in quorum failure groups for voting files whenever possible.

  A quorum failure group is not considered when determining redundancy requirements with respect to storing user data.

If you specify neither keyword, then REGULAR is the default.

Specify either QUORUM or REGULAR before the keyword FAILGROUP if you are explicitly specifying the failure group. If you are creating a disk group with implicitly created failure groups, then specify these keywords before the keyword DISK.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about quorum and regular disks and failure groups

FAILGROUP Clause

Use this clause to specify a name for one or more failure groups. If you omit this clause, and you have specified NORMAL or HIGH REDUNDANCY, then Oracle Database automatically adds each disk in the disk group to its own failure group. The implicit name of the failure group is the same as the operating system independent disk name (see "NAME Clause").

You cannot specify this clause if you are creating an EXTERNAL REDUNDANCY disk group.

qualified_disk_clause

Specify DISK *qualified_disk_clause* to add a disk to a disk group.

search_string

For each disk you are adding to the disk group, specify the operating system dependent search string that Oracle ASM will use to find the disk. The *search_string* must point to a subset of the disks returned by discovery using the strings in the ASM_DISKSTRING initialization parameter. If *search_string* does not point to any disks the Oracle Database user has read/write access to, then Oracle ASM returns an error. If it points to one or more disks that have already been assigned to a different disk group, then Oracle Database returns an error unless you also specify FORCE.

For each valid candidate disk, Oracle ASM formats the disk header to indicate that it is a member of the new disk group.

> **See Also:**
>
> The ASM_DISKSTRING initialization parameter for more information on specifying the search string

NAME Clause

The NAME clause is valid only if the *search_string* points to a single disk. This clause lets you specify an operating system independent name for the disk. The name can be up to 30 characters long and can contain only alphanumeric characters. The first character must be alphabetic. If you omit this clause and you assigned a label to a disk through ASMLIB, then that label is used as the disk name. If you omit this clause and you did not assign a label through ASMLIB, then Oracle ASM creates a default name of the form *diskgroupname_####*, where *####* is the disk number. You use this name to refer to the disk in subsequent Oracle ASM operations.

SIZE Clause

Use this clause to specify in bytes the size of the disk. If you specify a size greater than the capacity of the disk, then Oracle ASM returns an error. If you specify a size less than the capacity of the disk, then you limit the disk space Oracle ASM will use. The size value must be identical for all disks in a disk group. If you omit this clause, then Oracle ASM attempts programmatically to determine the size of the disk.

FORCE

Specify `FORCE` if you want Oracle ASM to add the disk to the disk group even if the disk is already a member of a different disk group.

> **Note:**
>
> Using `FORCE` in this way may destroy existing disk groups.

For this clause to be valid, the disk must already be a member of a disk group and the disk cannot be part of a mounted disk group.

NOFORCE

Specify `NOFORCE` if you want Oracle ASM to return an error if the disk is already a member of a different disk group. `NOFORCE` is the default.

ATTRIBUTE Clause

Use this clause to set attribute values for the disk group. You can view the current attribute values by querying the `V$ASM_ATTRIBUTE` view. Table 13-2 lists the attributes you can set with this clause. All attribute values are strings.

Table 13-2 Disk Group Attributes

Attribute	Valid Values	Description
`ACCESS_CONTROL.ENABLED`	`true` or `false`	Specifies whether Oracle ASM File Access Control is enabled for a disk group. If set to `true`, accessing Oracle ASM files is subject to access control. If `false`, any user can access every file in the disk group. All other operations behave independently of this attribute. The default value is `false`. If both the `compatible.rdbms` and `compatible.asm` attributes are set to at least 11.2, you can set this attribute in an `ALTER DISKGROUP ... SET ATTRIBUTE` statement. You cannot set this attribute when creating a disk group. When you set up file access control on an existing disk group, the files previously created remain accessible by everyone, unless you run the `ALTER DISKGROUP SET PERMISSION` statement to restrict the permissions. **Note**: This attribute is used in conjunction with `ACCESS_CONTROL.UMASK` to manage Oracle ASM File Access Control. After setting the `ACCESS_CONTROL.ENABLED` disk attribute, you must set permissions with the `ACCESS_CONTROL.UMASK` attribute.

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
ACCESS_CONTROL.UMASK	A three-digit number where each digit is 0, 2, or 6.	Determines which permissions are masked out on the creation of an Oracle ASM file for the user that owns the file (first digit), users in the same user group (second digit), and others not in the user group (third digit). This attribute applies to all files on a disk group. Setting to 0 masks out nothing. Setting to 2 masks out write permission. Setting to 6 masks out both read and write permissions. The default value is 066. If both the compatible.rdbms and compatible.asm attributes are set to at least 11.2, you can set this attribute in an ALTER DISKGROUP ... SET ATTRIBUTE statement. You cannot set this attribute when creating a disk group. When you set up file access control on an existing disk group, the files previously created remain accessible by everyone, unless you run the ALTER DISKGROUP SET PERMISSION statement to restrict the permissions. **Note**: This attribute is used in conjunction with ACCESS_CONTROL.ENABLED to manage Oracle ASM File Access Control. Before setting ACCESS_CONTROL.UMASK, you must set ACCESS_CONTROL.ENABLED to true.
AU_SIZE	Size in bytes. Valid values are powers of 2 from 1M to 64M. Examples '4M', '4194304'.	Specifies the allocation unit size. This attribute can be set only during disk group creation; it cannot be modified with an ALTER DISKGROUP statement.
COMPATIBLE.ADVM	Valid Oracle Database version number[1]	Determines whether the disk group can contain Oracle ADVM volumes. The value must be set to 11.2 or higher. Before setting this attribute, the COMPATIBLE.ASM value must be 11.2 or higher. Also, the Oracle ADVM volume drivers must be loaded. By default, the value of the COMPATIBLE.ADVM attribute is empty until set.
COMPATIBLE.ASM	Valid Oracle Database version number[1]	Determines the minimum software version for an Oracle ASM instance that can use the disk group. This setting also affects the format of the data structures for the Oracle ASM metadata on the disk. For Oracle ASM in Oracle Database 11*g*, 10.1 is the default setting for the COMPATIBLE.ASM attribute when using the SQL CREATE DISKGROUP statement, the ASMCMD mkdg command, and Oracle Enterprise Manager **Create Disk Group** page. When creating a disk group with ASMCA, the default setting is 11.2.

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
COMPATIBLE.RDBMS	Valid Oracle Database version number[1]	Determines the minimum COMPATIBLE database initialization parameter setting for any database instance that is allowed to use the disk group. Before advancing the COMPATIBLE.RDBMS attribute, ensure that the values for the COMPATIBLE initialization parameter for all of the databases that access the disk group are set to at least the value of the new setting for COMPATIBLE.RDBMS. For example, if the COMPATIBLE initialization parameters of the databases are set to either 11.1 or 11.2, then COMPATIBLE.RDBMS can be set to any value between 10.1 and 11.1 inclusively. For Oracle ASM in Oracle Database 11*g*, 10.1 is the default setting for the COMPATIBLE.RDBMS attribute when using the SQL CREATE DISKGROUP statement, the ASMCMD mkdg command, ASMCA **Create Disk Group** page, and Oracle Enterprise Manager **Create Disk Group** page.
CONTENT.CHECK	true or false	Enables (true) or disables (false) content checking when performing data copy operations for rebalancing a disk group. The default value is false. You cannot set this attribute when creating a disk group.
DISK_REPAIR_TIME	0 to 136 years	When disks are taken offline, Oracle ASM drops them after a default period of time. If both the compatible.rdbms and compatible.asm attributes are set to at least 11.1, you can set the disk_repair_time attribute in an ALTER DISKGROUP ... SET ATTRIBUTE statement to change that default period of time so that the disk can be repaired and brought back online. You cannot set this attribute when creating a disk group. The time can be specified in units of minute (M) or hour (H). The specified time elapses only when the disk group is mounted. If you omit the unit, then the default is H. If you omit this attribute, and both compatible.rdbms and compatible.asm are set to at least 11.1, then the default is 3.6 H. Otherwise the disk is dropped immediately. You can override this attribute with an ALTER DISKGROUP ... OFFLINE DISK statement and the DROP AFTER clause. **Note**: If a disk is taken offline using the current value of disk_repair_time, and the value of this attribute is subsequently changed, then the changed value is used by Oracle ASM in the disk offline logic. **See Also**: The ALTER DISKGROUP ... *disk_offline_clause* and *Oracle Automatic Storage Management Administrator's Guide* for more information

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
FAILGROUP_REPAIR_TIME	<number>m (number of minutes) or <number>h (number of hours)	Specifies a default repair time for the failure groups in the disk group. The failure group repair time is used if Oracle ASM determines that an entire failure group has failed. The default value is 24 hours (24h). If there is a repair time specified for a disk, such as with the DROP AFTER clause of the ALTER DISKGROUP OFFLINE DISK statement, then that disk repair time overrides the failure group repair time. This attribute can only be set when altering a disk group and is only applicable to normal and high redundancy disk groups.
LOGICAL_SECTOR_SIZE	512, 4096, or 4K	Sets the logical sector size of a disk group. This value specifies the smallest possible I/O that the disk group can accept. The default value is estimated from the disks that join the disk group. To set this disk group attribute during the creation of a disk group or to alter it after a disk group has been created, the COMPATIBLE.ASM disk group attribute must be set to 12.2 or higher.
PHYS_META_REPLICATED	true or false	Tracks the replication status of a disk group. When the Oracle ASM compatibility of a disk group is advanced to 12.0 or higher, the physical metadata of each disk, including its disk header, free space table blocks and allocation table blocks, is replicated. The replication is performed online asynchronously. PHYS_META_REPLICATED is set to true by Oracle ASM if the physical metadata of every disk in the disk group has been replicated. This disk group attribute is only defined in a disk group with the Oracle ASM disk group compatibility (COMPATIBLE.ASM) set to 12.0 and higher. This attribute is read-only and is intended for information only. You cannot set or change its value.
PREFERRED_READ.ENABLED	true or false	In an Oracle extended cluster, which contains nodes that span multiple physically separated sites, the PREFERRED_READ.ENABLED disk group attribute controls whether preferred read functionality is enabled for a disk group. If preferred read functionality is enabled, then this functionality enables an instance to determine and read from disks at the same site as itself, which can improve performance. For extended clusters, the default value is true. For clusters that are not extended (only one physical site), preferred read is disabled (false). Preferred read status applies to extended, normal, high, and flex redundancy disk groups. This disk group attribute is only defined in a disk group with the Oracle ASM disk group compatibility (COMPATIBLE.ASM) set to 12.2 and higher.

Table 13-2 (Cont.) Disk Group Attributes

Attribute	Valid Values	Description
`SECTOR_SIZE`	512, 4096, or 4K	Sets the physical sector size of a disk group. All disks in the disk group must have this physical sector size. The default value is obtained from the disks that join the disk group. To set this disk group attribute during the creation of a disk group, the `COMPATIBLE.ASM` and `COMPATIBLE.RDBMS` disk group attributes must be set to `11.2` or higher. To alter this disk group attribute after a disk group has been created, the `COMPATIBLE.ASM` disk group attribute must be set to `12.2` or higher.
`THIN_PROVISIONED`	`true` or `false`	Enables (`true`) or disables (`false`) the functionality to discard unused storage space after a disk group rebalance is completed. The default value is `false`.
`CONTENT_HARDCHECK`	`true` or `false`	`CONTENT_HARDCHECK` enables or disables Hardware Assisted Resilient Data (HARD) checking when performing data copy operations for rebalancing a disk group. This attribute can only be set when altering a disk group.

1 Specify at least the first two digits of a valid Oracle Database release number. Refer to *Oracle Database Administrator's Guide* for information on specifying valid version numbers. For example, you can specify `compatibility` as '11.2' or '12.1'.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information on managing these attribute settings

Examples

The following example assumes that the `ASM_DISKSTRING` parameter is a superset of `/devices/disks/c*`, `/devices/disks/c*` points to at least one device to be used as an Oracle ASM disk, and the Oracle Database user has read/write permission to the disks.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for information on Oracle ASM and using disk groups to simplify database administration

Creating a Diskgroup: Example

The following statement creates an Oracle ASM disk group `dgroup_01` where no redundancy for the disk group is provided by Oracle ASM and includes all disks that match the *`search_string`*:

```
CREATE DISKGROUP dgroup_01
  EXTERNAL REDUNDANCY
  DISK '/devices/disks/c*';
```

CREATE EDITION

Purpose

This statement creates a new edition as a child of an existing edition. An edition makes it possible to have two or more versions of the same editionable objects in the database. When you create an edition, it immediately inherits all of the editionable objects of its parent edition. The following object types are editionable:

- Synonym
- View
- Function
- Procedure
- Package (specification and body)
- Type (specification and body)
- Library
- Trigger

An editionable object is an object of one of the above editionable object types in an editions-enabled schema. The ability to have multiple versions of these objects in the database greatly facilitates online application upgrades.

> **Note:**
>
> All database object types not listed above are not editionable. Changes to object types that are not editionable are immediately visible across all editions in the database.

Every newly created or upgraded Oracle Database has one default edition named `ORA$BASE`, which serves as the parent of the first edition created with a `CREATE EDITION` statement. You can subsequently designate a user-defined edition as the database default edition using an `ALTER DATABASE DEFAULT EDITION` statement.

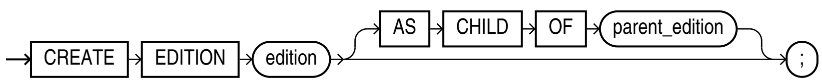
> **See Also:**
>
> - *Oracle Database Development Guide* for a more complete discussion of editionable object types and editions
> - The `ALTER DATABASE` "DEFAULT EDITION Clause" for information on designating an edition as the default edition for the database

Prerequisites

To create an edition, you must have the `CREATE ANY EDITION` system privilege, granted either directly or through a role. To create an edition as a child of another edition, you must have the `USE` object privilege on the parent edition.

Syntax

***create_edition*::=**



Semantics

edition

Specify the name of the edition to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

To view the editions that have been created for the database, query the `EDITION_NAME` column of the `DBA_OBJECTS` or `ALL_OBJECTS` data dictionary view.

When you create an edition, the system automatically grants you the `USE` object privilege `WITH GRANT OPTION` on the edition you create.

> **Note:**
>
> Oracle strongly recommends that you do not name editions with the prefixes `ORA`, `ORACLE`, `SYS`, `DBA`, and `DBMS`, as these prefixes are reserved for internal use.

AS CHILD OF Clause

If you use this clause, then the new edition is created as a child of *`parent_edition`*. If you omit this clause, then the new edition is created as a child of the leaf edition. At the time of its creation, the new edition inherits all editioned objects from its parent edition.

Restriction on Editions

An edition can have only one child edition. If you specify for *`parent_edition`* an edition that already has a child edition, then an error is returned.

Examples

The following very simple examples are intended to show the syntax for creating and working with an edition. For realistic examples of using editions refer to *Oracle Database Development Guide*.

In the following statements, the user `HR` is given the privileges needed to create and use an edition:

```
GRANT CREATE ANY EDITION, DROP ANY EDITION to HR;
Grant succeeded.

ALTER USER hr ENABLE EDITIONS;
User altered.
```

`HR` creates a new edition `TEST_ED` for testing purposes:

```
CREATE EDITION test_ed;
```

`HR` then creates an editioning view `ed_view` in the default edition `ORA$BASE` for testing purposes, first verifying that the current edition is the default edition:

```
SELECT SYS_CONTEXT('userenv', 'current_edition_name') FROM DUAL;
SYS_CONTEXT('USERENV','CURRENT_EDITION_NAME')
--------------------------------------------------------------------------------
ORA$BASE
1 row selected.

CREATE EDITIONING VIEW e_view AS
  SELECT last_name, first_name, email FROM employees;
View created.

DESCRIBE e_view
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
```

The view is then actualized in the `TEST_ED` edition when `HR` uses the `TEST_ED` edition and re-creates the view in a different form:

```
ALTER SESSION SET EDITION = TEST_ED;
Session altered.

CREATE OR REPLACE EDITIONING VIEW e_view AS
  SELECT last_name, first_name, email, salary FROM employees;

View created.
```

The view in the `TEST_ED` edition has an additional column:

```
DESCRIBE e_view
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
 SALARY                                             NUMBER(8,2)
```

The view in the `ORA$BASE` edition remains isolated from the test environment:

```
ALTER SESSION SET EDITION = ora$base;
Session altered.

DESCRIBE e_view;

 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
```

```
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
```

Even if the view is dropped in the test environment, it remains in the `ORA$BASE` edition:

```
ALTER SESSION SET EDITION = TEST_ED;
Session altered.

DROP VIEW e_view;
View dropped.

ALTER SESSION SET EDITION = ORA$BASE;
Session altered.

DESCRIBE e_view;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 FIRST_NAME                                         VARCHAR2(20)
 EMAIL                                     NOT NULL VARCHAR2(25)
```

When the testing of upgrade that necessitated the `TEST_ED` edition is complete, the edition can be dropped:

```
DROP EDITION TEST_ED;
```

CREATE FLASHBACK ARCHIVE

Purpose

Use the `CREATE FLASHBACK ARCHIVE` statement to create a flashback data archive, which provides the ability to automatically track and archive transactional data changes to specified database objects. A flashback data archive consists of multiple tablespaces and stores historic data from all transactions against tracked tables. The data is stored in internal history tables.

Flashback data archives retain historical data for the time duration specified using the `RETENTION` parameter. Historical data can be queried using the Flashback Query `AS OF` clause. Archived historic data that has aged beyond the specified retention period is automatically purged.

Flashback data archives retain historical data across data definition language (DDL) changes to tables enabled for flashback data archive. Flashback data archives supports many common DDL statements, including some DDL statements that alter table definitions or incur data movement. DDL statements that are not supported result in error ORA-55610.

See Also:

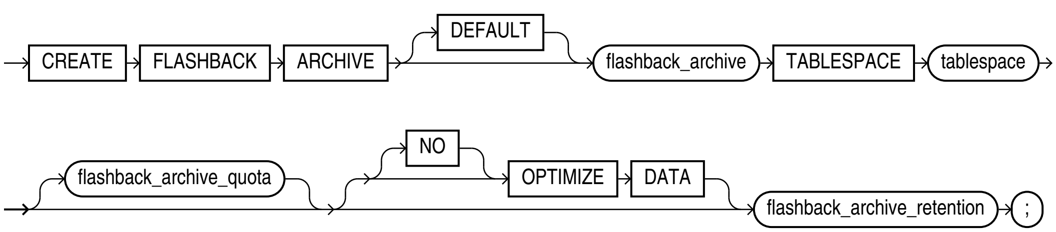
- *Oracle Database Development Guide* for general information on using flashback data archives
- The `CREATE TABLE` *flashback_archive_clause* for information on designating a table as a tracked table
- ALTER FLASHBACK ARCHIVE for information on changing the quota and retention attributes of the flashback data archive, as well as adding or changing tablespace storage for the flashback data archive

Prerequisites

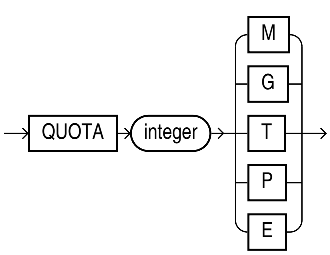
You must have the `FLASHBACK ARCHIVE ADMINISTER` system privilege to create a flashback data archive. In addition, you must have the `CREATE TABLESPACE` system privilege to create a flashback data archive, as well as sufficient quota on the tablespace in which the historical information will reside. To designate a flashback data archive as the system default flashback data archive, you must be logged in as `SYSDBA`.

Syntax

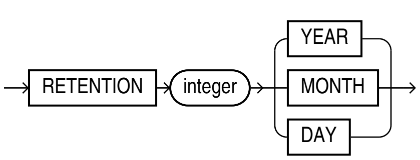
***create_flashback_archive*::=**



***flashback_archive_quota*::=**



***flashback_archive_retention*::=**



Semantics

DEFAULT

You must be logged in as `SYSDBA` to specify `DEFAULT`. Use this clause to designate this flashback data archive as the default flashback data archive for the database. When a `CREATE TABLE` or `ALTER TABLE` statement specifies the *`flashback_archive_clause`* without specifying a flashback data archive name, the database uses the default flashback data archive to store data from that table.

You cannot specify this clause if a default flashback data archive already exists. However, you can replace an existing default flashback data archive using the `ALTER FLASHBACK ARCHIVE` ... `SET DEFAULT` clause.

> **See Also:**
>
> The `CREATE TABLE` *flashback_archive_clause* for more information

flashback_archive

Specify the name of the flashback data archive. The name must satisfy the requirements specified in "Database Object Naming Rules ".

TABLESPACE Clause

Specify the tablespace where the archived data for this flashback data archive is to be stored. You can specify only one tablespace with this clause. However, you can subsequently add tablespaces to the flashback data archive with an `ALTER FLASHBACK ARCHIVE` statement.

flashback_archive_quota

Specify the amount of space in the initial tablespace to be reserved for the archived data. If the space for archiving in a flashback data archive becomes full, then DML operations on tracked tables that use this flashback data archive will fail. The database issues an out-of-space alert when the content of the flashback data archive is 90% of the specified quota, to allow time to purge old data or add additional quota. If you omit this clause, then the flashback data archive has unlimited quota on the specified tablespace.

[NO] OPTIMIZE DATA

Specify `OPTIMIZE DATA` to enable optimization for flashback data archive history tables. This instructs the database to optimize the storage of data in history tables using any of the following features: Advanced Row Compression, Advanced LOB Compression, Advanced LOB Deduplication, segment-level compression tiering, and row-level compression tiering. To specify this clause, you must have a license for the Advanced Compression option.

Specify `NO OPTIMIZE DATA` to instruct the database not to optimize the storage of data in history tables. This is the default.

flashback_archive_retention

Specify the length of time in months, days, or years that the archived data should be retained in the flashback data archive. If the length of time causes the flashback data archive to become full, then the database responds as described in *flashback_archive_quota*.

Examples

The following statement creates two flashback data archives for testing purposes. The first is designated as the default for the database. For both of them, the space quota is 1 megabyte, and the archive retention is one day.

```
CREATE FLASHBACK ARCHIVE DEFAULT test_archive1
   TABLESPACE example
   QUOTA 1 M
   RETENTION 1 DAY;

CREATE FLASHBACK ARCHIVE test_archive2
   TABLESPACE example
   QUOTA 1 M
   RETENTION 1 DAY;
```

The next statement alters the default flashback data archive to extend the retention period to 1 month:

```
ALTER FLASHBACK ARCHIVE test_archive1
   MODIFY RETENTION 1 MONTH;
```

The next statement specifies tracking for the `oe.customers` table. The flashback data archive is not specified, so data will be archived in the default flashback data archive, `test_archive1`:

```
ALTER TABLE oe.customers
   FLASHBACK ARCHIVE;
```

The next statement specifies tracking for the `oe.orders` table. In this case, data will be archived in the specified flashback data archive, `test_archive2`:

```
ALTER TABLE oe.orders
   FLASHBACK ARCHIVE test_archive2;
```

The next statement drops `test_archive2` flashback data archive:

```
DROP FLASHBACK ARCHIVE test_archive2;
```

CREATE FUNCTION

Purpose

Functions are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE FUNCTION` statement to create a standalone stored function or a call specification.

- A **stored function** (also called a **user function** or **user-defined function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

- A **call specification** declares a Java method or a third-generation language (3GL) routine so that it can be called from PL/SQL. You can also use the `CALL` SQL statement to call such a method or routine. The call specification tells Oracle Database which Java method, or which named function in which shared library, to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

> **Note:**
>
> You can also create a function as part of a package using the `CREATE PACKAGE` statement.

> **See Also:**
>
> - CREATE PROCEDURE for a general discussion of procedures and functions, CREATE PACKAGE for information on creating packages, ALTER FUNCTION and DROP FUNCTION for information on modifying and dropping a function
> - CREATE LIBRARY for information on shared libraries
> - *Oracle Database Development Guide* for more information about registering external functions

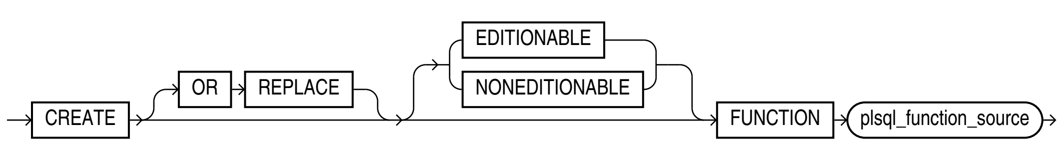
Prerequisites

To create or replace a function in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a function in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

Syntax

Functions are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_function*::=**



(*`plsql_function_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the function if it already exists. Use this clause to change the definition of an existing function without dropping, re-creating, and regranting object privileges previously granted on the function. If you redefine a function, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, then Oracle Database marks the indexes `DISABLED`.

> **See Also:**
>
> `ALTER FUNCTION` for information on recompiling functions using SQL

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the function is an editioned or noneditioned object if editioning is enabled for the schema object type `FUNCTION` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_function_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_function_source`*, including examples.

CREATE HIERARCHY

Purpose

Use the `CREATE HIERARCHY` statement to create a hierarchy. A hierarchy specifies the hierarchical relationships among the levels of an attribute dimension.

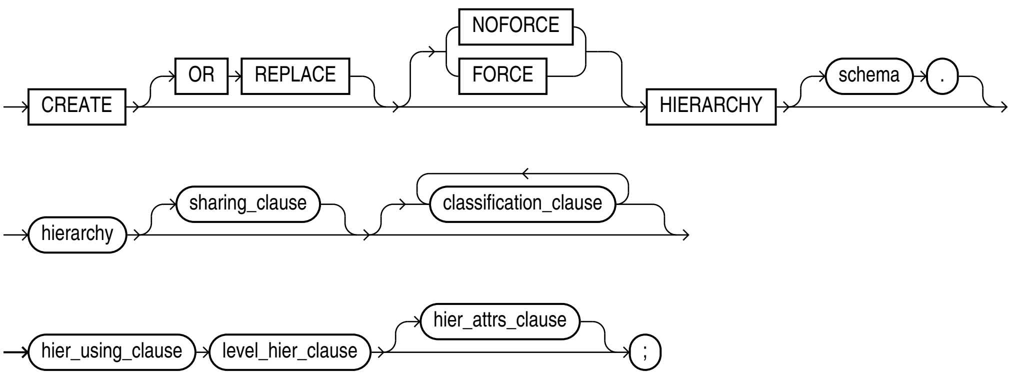
> **Tip:**
>
> You can view and run SQL scripts that create hierarchies at the Oracle Live SQL website at https://livesql.oracle.com/apex/livesql/file/index.html. The website has scripts and tutorials that demonstrate the creation and use of analytic views.

Prerequisites

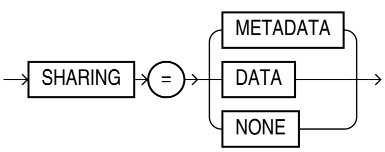
To create a hierarchy in your own schema, you must have the `CREATE HIERARCHY` system privilege. To create a hierarchy in another user's schema, you must have the `CREATE ANY HIERARCHY` system privilege.

Syntax

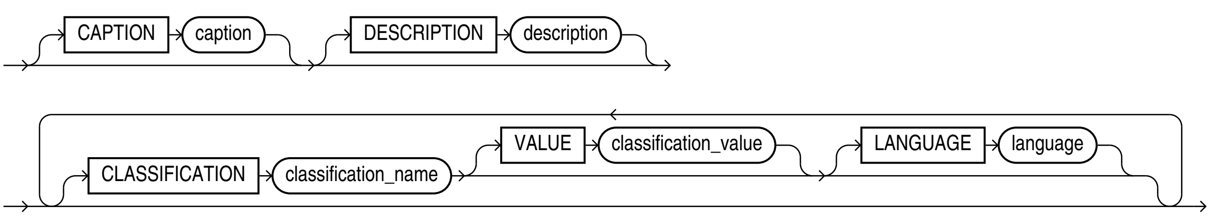
***create_hierarchy*::=**



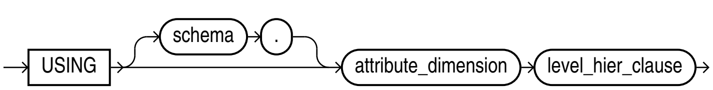
***sharing_clause*::=**



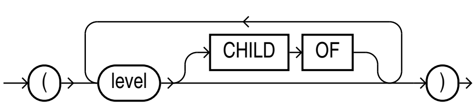
***classification_clause*::=**



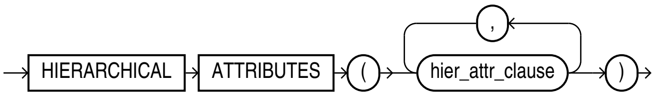
***hier_using_clause*::=**



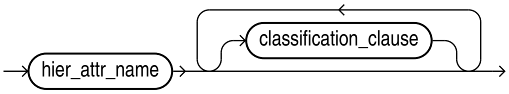
***level_hier_clause*::=**



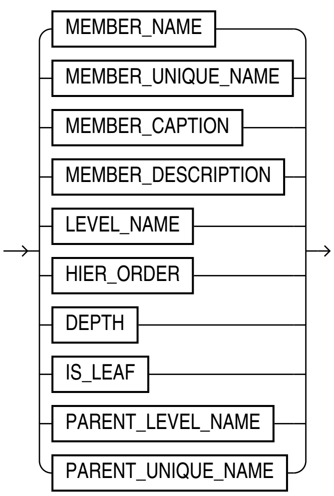
***hier_attrs_clause*::=**



***hier_attr_clause*::=**



***hier_attr_name*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing definition of a hierarchy with a different definition.

FORCE and NOFORCE

Specify `FORCE` to force the creation of the hierarchy even if it does not successfully compile. If you specify `NOFORCE`, then the hierarchy must compile successfully, otherwise an error occurs. The default is `NOFORCE`.

schema

Specify the schema in which to create the hierarchy. If you do not specify a schema, then Oracle Database creates the hierarchy in your own schema.

hierarchy

Specify a name for the hierarchy.

sharing_clause

Specify whether to create the hierarchy as an application common object. Specifying `METADATA` shares the hierarchy's metadata, but its data is unique to each container. Specifying `DATA` shares the hierarchy object; its data is the same for all containers in the application container and the data is stored only in the application root. Specifying `NONE` excludes the hierarchy from being shared.

classification_clause

Use the classification clause to specify values for the `CAPTION` or `DESCRIPTION` classifications and to specify user-defined classifications. Classifications provide descriptive metadata that applications may use to provide information about analytic views and their components.

You may specify any number of classifications for the same object. A classification can have a maximum length of 4000 bytes.

For the `CAPTION` and `DESCRIPTION` classifications, you may use the DDL shortcuts `CAPTION 'caption'` and `DESCRIPTION 'description'` or the full classification syntax.

You may vary the classification values by language. To specify a language for the `CAPTION` or `DESCRIPTION` classification, you must use the full syntax. If you do not specify a language, then the language value for the classification is `NULL`. The language value must either be `NULL` or a valid `NLS_LANGUAGE` value.

hier_using_clause

Specify the attribute dimension that has the members of the hierarchy.

level_hier_clause

Specify the organization of the hierarchy levels.

hier_attrs_clause

Specify classifications that contain descriptive metadata for the hierarchical attributes. A *`hier_attr_clause`* for a given *`hier_attr_name`* may appear only once in the list.

All hierarchies always contain all of the hierarchical attributes, but a hierarchical attribute does not have descriptive metadata associated with it unless you specify it with this clause.

hier_attr_clause

Specify a hierarchical attribute and provide one or more classifications for it.

hier_attr_name

Specify a hierarchical attribute.

Examples

The following example creates the `TIME_HIER` hierarchy:

```
CREATE OR REPLACE HIERARCHY time_hier  -- Hierarchy name
USING time_attr_dim                   -- Refers to TIME_ATTR_DIM attribute
dimension
 (month CHILD OF                      -- Months in the attribute dimension
 quarter CHILD OF
 year);
```

The following example creates the `PRODUCT_HIER` hierarchy:

```
CREATE OR REPLACE HIERARCHY product_hier
USING product_attr_dim
 (category
  CHILD OF department);
```

The following example creates the `GEOGRAPHY_HIER` hierarchy:

```
CREATE OR REPLACE HIERARCHY geography_hier
USING geography_attr_dim
 (state_province
  CHILD OF country
  CHILD OF region);
```

CREATE INDEX

Purpose

Use the `CREATE INDEX` statement to create an index on:

- One or more columns of a table, a partitioned table, an index-organized table, or a cluster
- One or more scalar typed object attributes of a table or a cluster
- A nested table storage table for indexing a nested table column

An **index** is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. The maximum size of a single index entry is dependent on the block size of the database.

Oracle Database supports several types of index:

- **Normal indexes.** (By default, Oracle Database creates B-tree indexes.)
- **Bitmap indexes**, which store rowids associated with a key value as a bitmap.
- **Partitioned indexes**, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table.
- **Function-based indexes**, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.
- **Domain indexes**, which are instances of an application-specific index of type *`indextype`*.

> **Note:**
>
> - *Oracle Database Concepts* for a discussion of indexes
> - *Oracle Database Reference* for more information about the limits related to index size
> - ALTER INDEX and DROP INDEX

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

- The table or cluster to be indexed must be in your own schema.
- You must have the `INDEX` object privilege on the table to be indexed.
- You must have the `CREATE ANY INDEX` system privilege.

To create an index in another schema, you must have the `CREATE ANY INDEX` system privilege. Also, the owner of the schema to contain the index must have either the `UNLIMITED TABLESPACE` system privilege or space quota on the tablespaces to contain the index or index partitions.

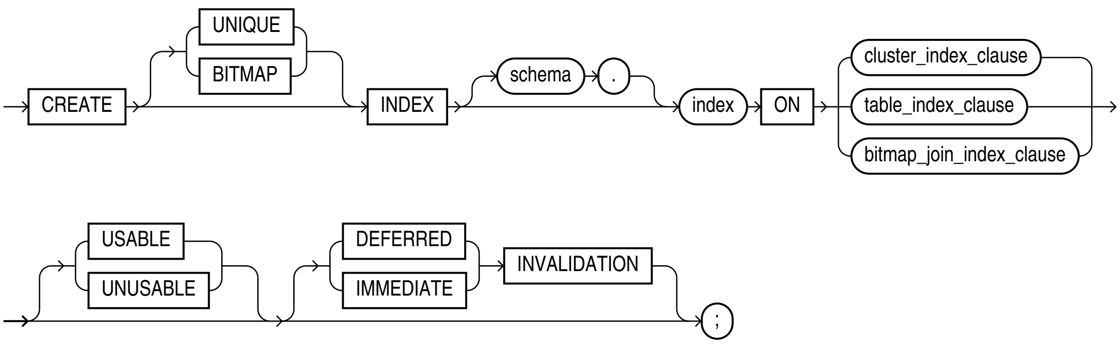
To create a function-based index, in addition to the prerequisites for creating a conventional index, if the index is based on user-defined functions, then those functions must be marked `DETERMINISTIC`. A function-based index is executed with the credentials of the index owner, so the index owner must have the `EXECUTE` object privilege on the function.

To create a domain index in your own schema, in addition to the prerequisites for creating a conventional index, you must also have the `EXECUTE` object privilege on the indextype. If you are creating a domain index in another user's schema, then the index owner also must have the `EXECUTE` object privilege on the indextype and its underlying implementation type. Before creating a domain index, you should first define the indextype.

> **See Also:**
>
> CREATE INDEXTYPE

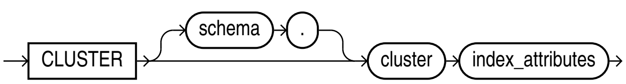
Syntax

***create_index*::=**



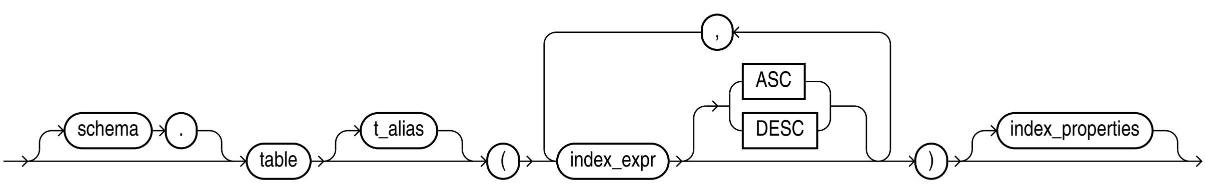
(*cluster_index_clause*::=, *table_index_clause*::=, *bitmap_join_index_clause*::=)

***cluster_index_clause*::=**



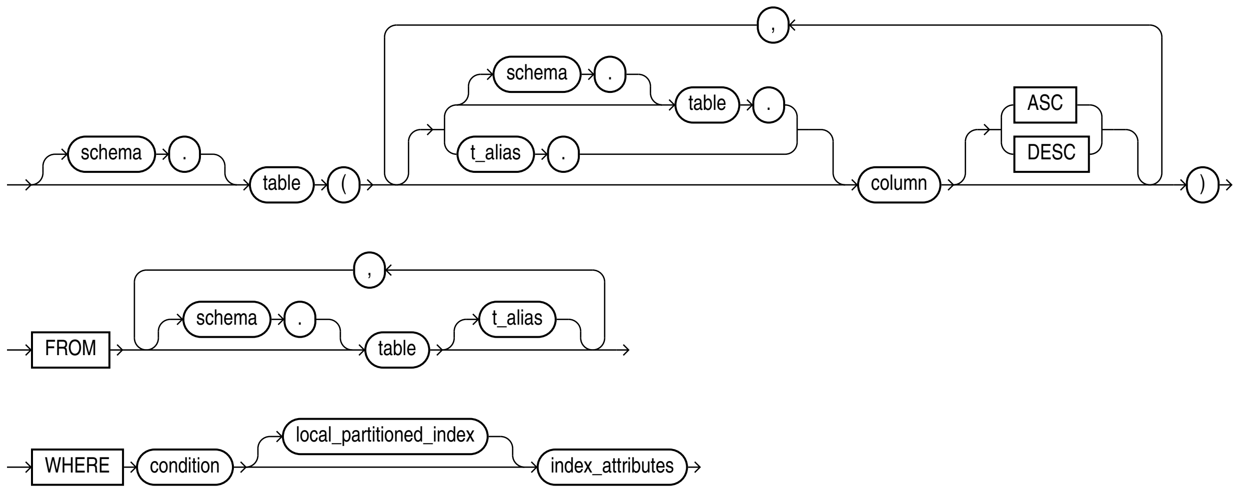
(*index_attributes*::=)

***table_index_clause*::=**



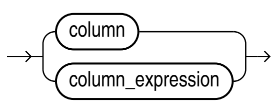
(*index_properties*::=)

***bitmap_join_index_clause*::=**

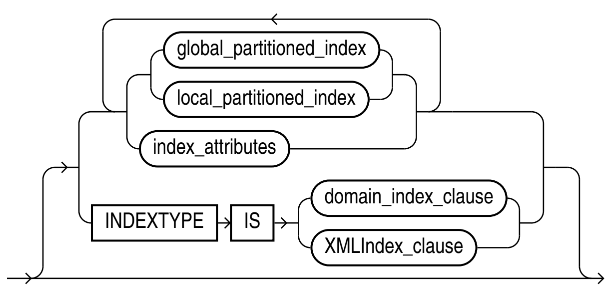


(*local_partitioned_index*::=, *index_attributes*::=)

***index_expr*::=**

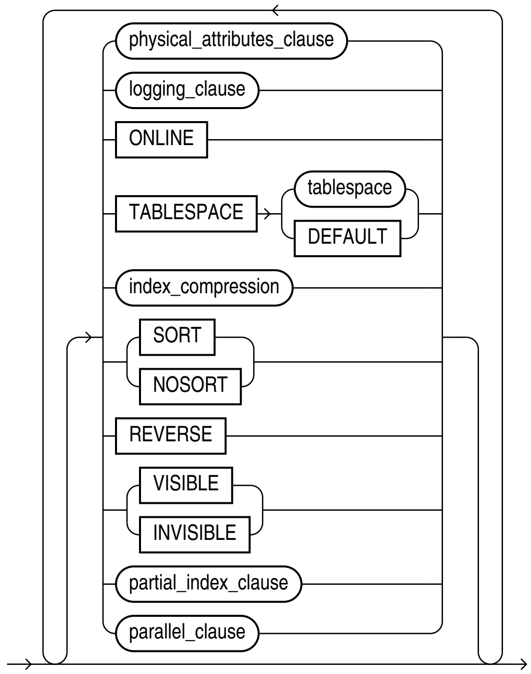


***index_properties*::=**



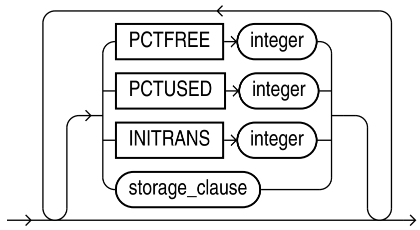
(*global_partitioned_index*::=, *local_partitioned_index*::=, *index_attributes*::=, *domain_index_clause*::=, *XMLIndex_clause*::=)

***index_attributes*::=**



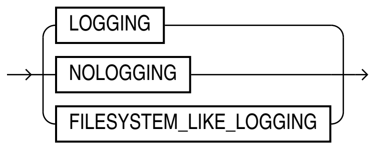
(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, *partial_index_clause*::=, *parallel_clause*::=)

***physical_attributes_clause*::=**



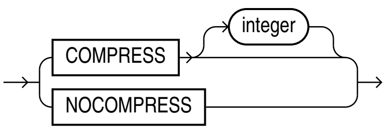
(*storage_clause*::=)

***logging_clause*::=**

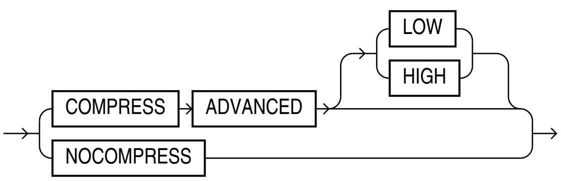


***index_compression*::=**

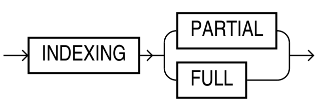
***prefix_compression*::=**



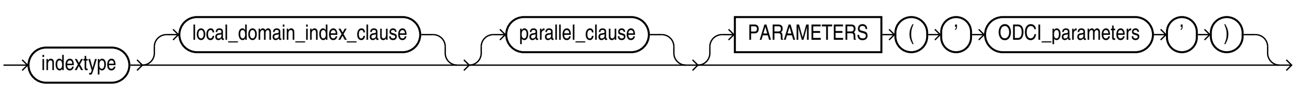
***advanced_index_compression*::=**



***partial_index_clause*::=**

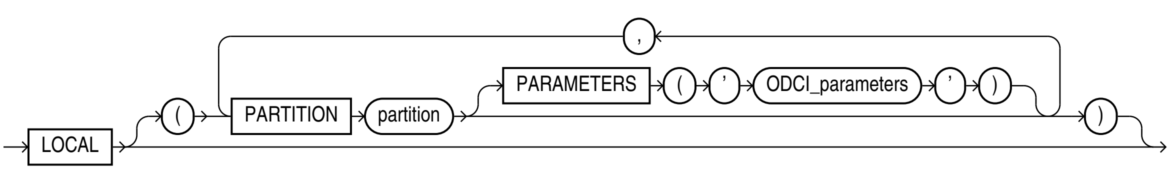


***domain_index_clause*::=**

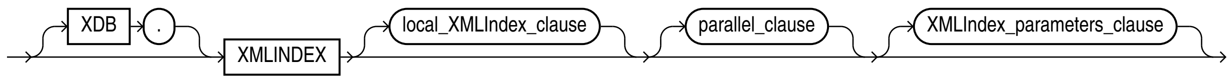


(*parallel_clause*::=)

***local_domain_index_clause*::=**

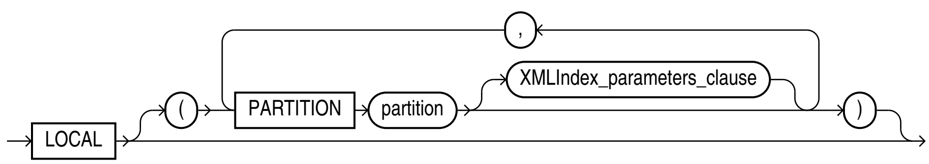


***XMLIndex_clause*::=**



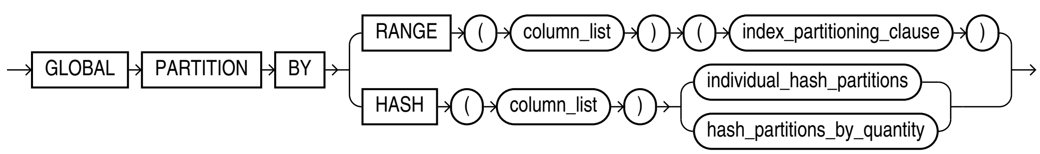
(The *XMLIndex_parameters_clause* is documented in *Oracle XML DB Developer's Guide*.)

***local_XMLIndex_clause*::=**



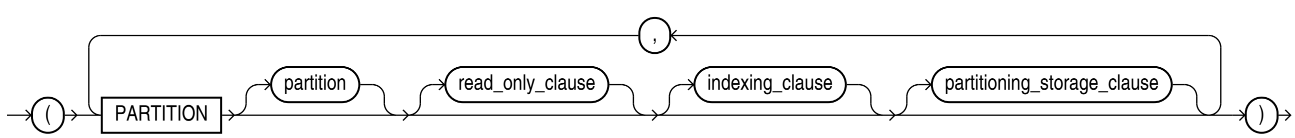
(The *XMLIndex_parameters_clause* is documented in *Oracle XML DB Developer's Guide*.)

***global_partitioned_index*::=**



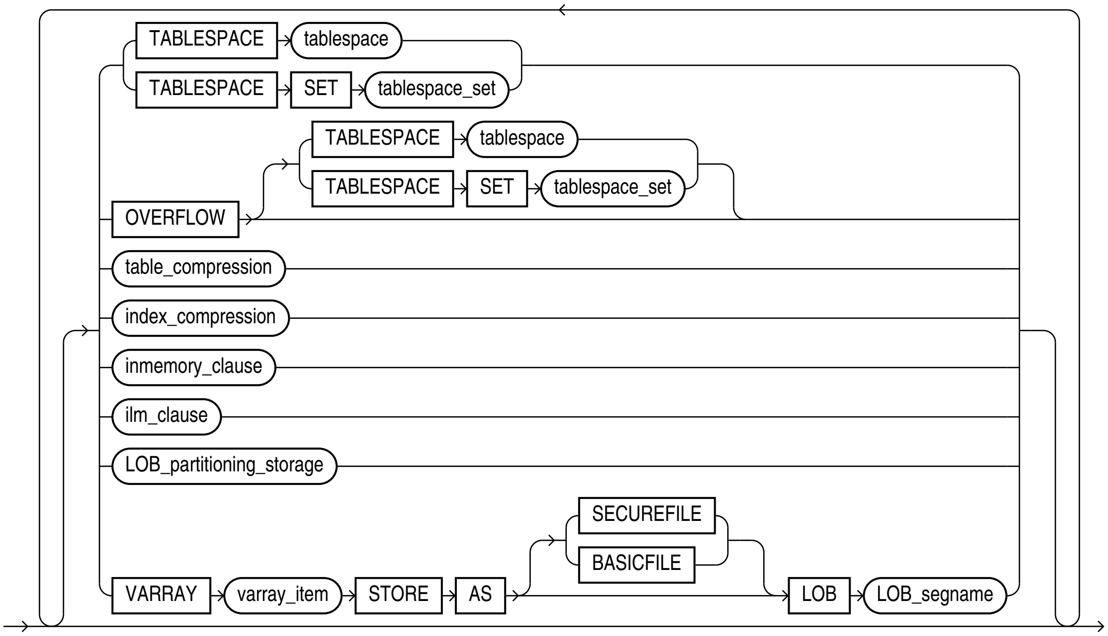
(*index_partitioning_clause*::=, *individual_hash_partitions*::=, *hash_partitions_by_quantity*::=)

***individual_hash_partitions*::=**



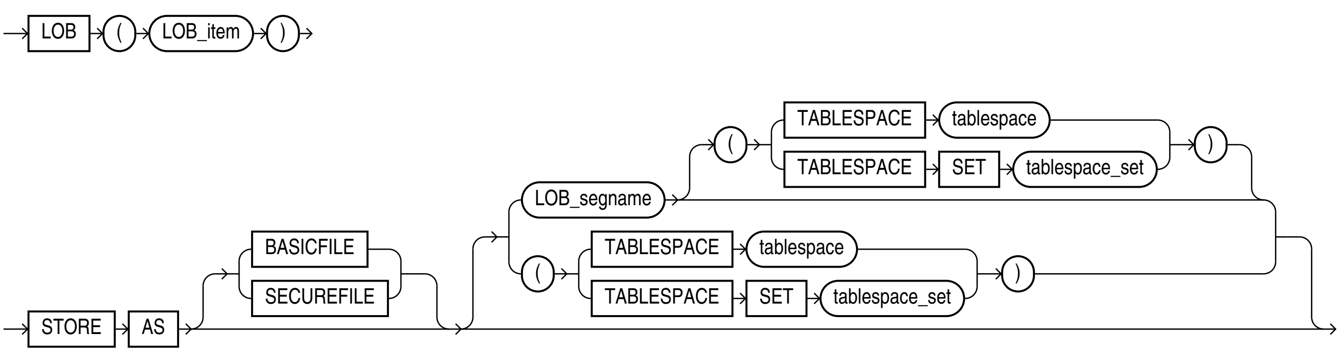
(*read_only_clause* and *indexing_clause*: not supported in *table_index_clause*, *partitioning_storage_clause*::=)

***partitioning_storage_clause*::=**



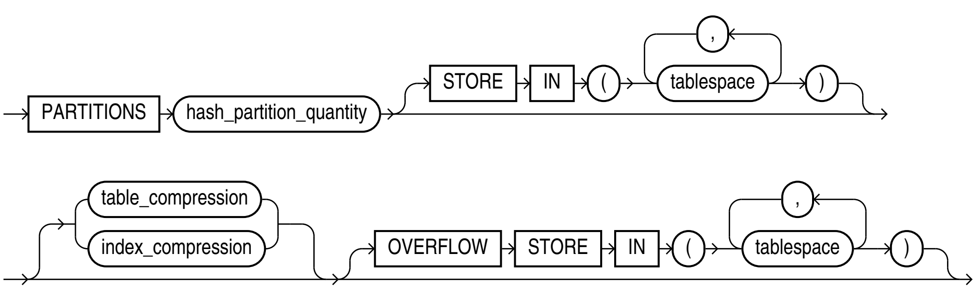
(`TABLESPACE SET`, *`table_compression`*, *`inmemory_clause`*, and *`ilm_clause`* not supported with `CREATE INDEX`, *index_compression*::=, *LOB_partitioning_storage*::=)

***LOB_partitioning_storage*::=**

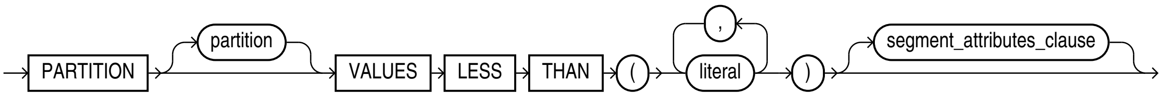


(`TABLESPACE SET`: not supported with `CREATE INDEX`)

***hash_partitions_by_quantity*::=**

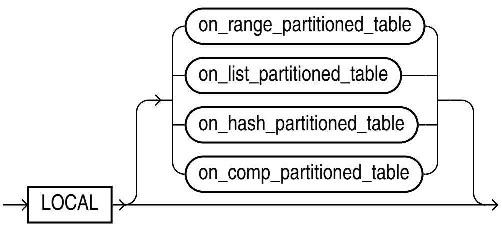


***index_partitioning_clause*::=**



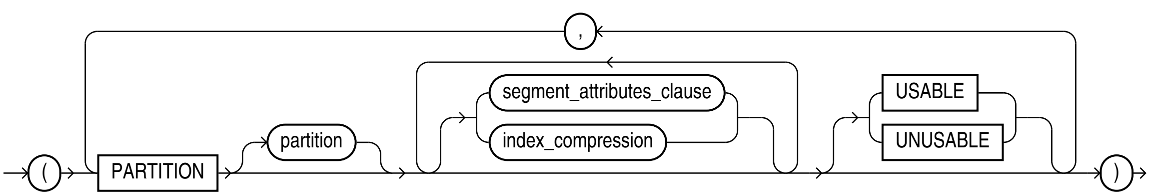
(*segment_attributes_clause::=*)

***local_partitioned_index*::=**



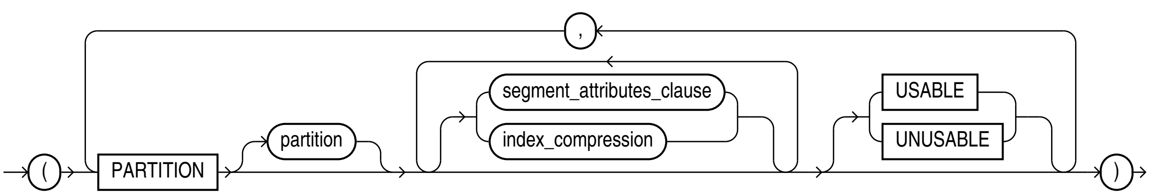
(*on_range_partitioned_table::=*, *on_list_partitioned_table::=*, *on_hash_partitioned_table::=*, *on_comp_partitioned_table::=*)

***on_range_partitioned_table*::=**



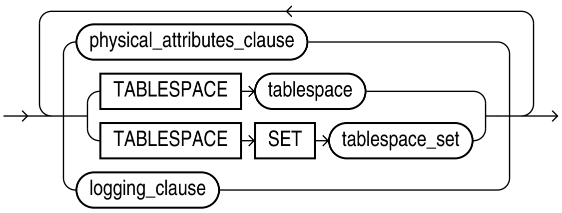
(*segment_attributes_clause::=*)

***on_list_partitioned_table*::=**



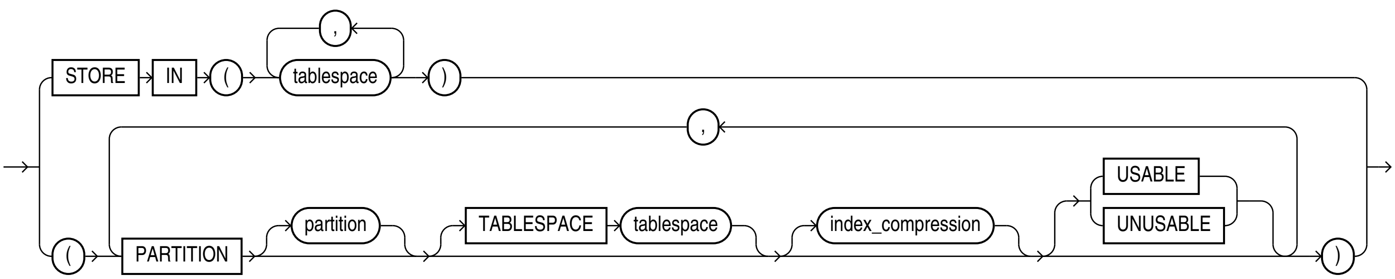
(*segment_attributes_clause::=*)

***segment_attributes_clause*::=**

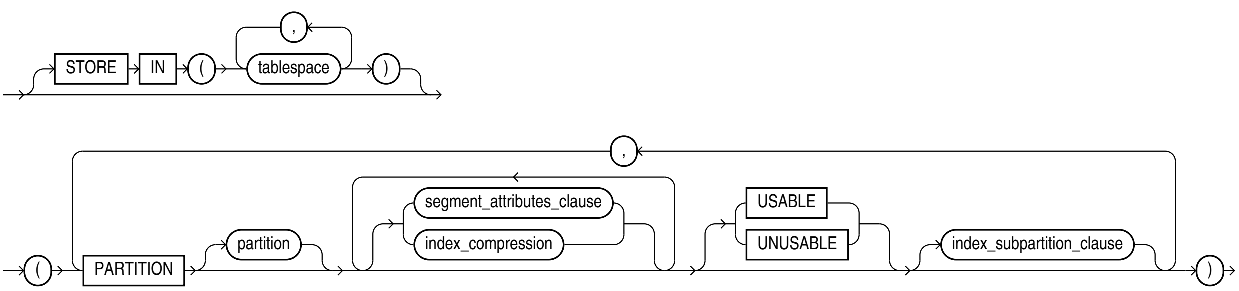


(*physical_attributes_clause::=*, `TABLESPACE SET`: not supported with `CREATE INDEX`, *logging_clause::=*

***on_hash_partitioned_table*::=**

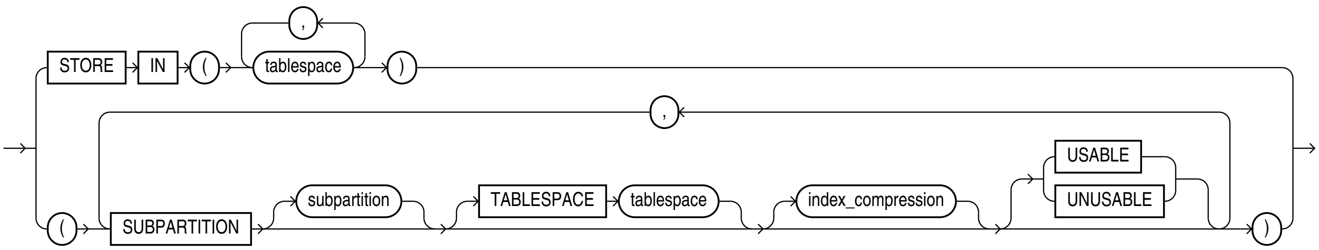


***on_comp_partitioned_table*::=**

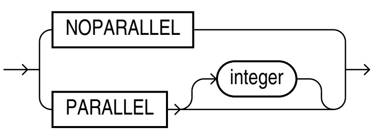


(*segment_attributes_clause::=*, *index_compression::=*, *index_subpartition_clause::=*)

***index_subpartition_clause*::=**



***parallel_clause*::=**



Semantics

UNIQUE

Specify `UNIQUE` to indicate that the value of the column (or columns) upon which the index is based must be unique.

Restrictions on Unique Indexes

Unique indexes are subject to the following restrictions:

- You cannot specify both `UNIQUE` and `BITMAP`.
- You cannot specify `UNIQUE` for a domain index.

> **See Also:**
>
> "Unique Constraints " for information on the conditions that satisfy a unique constraint

BITMAP

Specify `BITMAP` to indicate that *`index`* is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible rowid. If the bit is set, then it means that the row with the corresponding rowid contains the key value. The internal representation of bitmaps is best suited for applications with low levels of concurrent transactions, such as data warehousing.

> **Note:**
>
> Oracle does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, then you must either specify `NOT NULL` constraints for the index key columns or create a bitmap index.

Restrictions on Bitmap Indexes

Bitmap indexes are subject to the following restrictions:

- You cannot specify `BITMAP` when creating a global partitioned index.
- You cannot create a bitmap secondary index on an index-organized table unless the index-organized table has a mapping table associated with it.
- You cannot specify both `UNIQUE` and `BITMAP`.
- You cannot specify `BITMAP` for a domain index.
- A bitmap index can have a maximum of 30 columns.

> **See Also:**
>
> - *Oracle Database Concepts* and *Oracle Database SQL Tuning Guide* for more information about using bitmap indexes
> - CREATE TABLE for information on mapping tables
> - "Bitmap Index Examples"

schema

Specify the schema to contain the index. If you omit *`schema`*, then Oracle Database creates the index in your own schema.

index

Specify the name of the index to be created. The name must satisfy the requirements listed in Database Object Naming Rules .

> **See Also:**
>
> Creating an Index: Example and Creating an Index on an XMLType Table: Example

cluster_index_clause

Use the *`cluster_index_clause`* to identify the cluster for which a cluster index is to be created. If you do not qualify cluster with *`schema`*, then Oracle Database assumes the cluster is in your current schema. You cannot create a cluster index for a hash cluster.

> **See Also:**
>
> CREATE CLUSTER and "Creating a Cluster Index: Example"

table_index_clause

Specify the table on which you are defining the index. If you do not qualify *`table`* with *`schema`*, then Oracle Database assumes the table is contained in your own schema.

You create an index on a nested table column by creating the index on the nested table storage table. Include the `NESTED_TABLE_ID` pseudocolumn of the storage table to create a `UNIQUE` index, which effectively ensures that the rows of a nested table value are distinct.

> **See Also:**
>
> "Indexes on Nested Tables: Example"

You can perform DDL operations (such as `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`) on a temporary table only when no session is bound to it. A session becomes bound to a temporary table by performing an `INSERT` operation on the table. A session becomes unbound to the temporary table by issuing a `TRUNCATE` statement or at session termination, or, for a transaction-specific temporary table, by issuing a `COMMIT` or `ROLLBACK` statement.

Restrictions on the *table_index_clause*

This clause is subject to the following restrictions:

- If *`index`* is locally partitioned, then *`table`* must be partitioned.
- If *`table`* is index-organized, then this statement creates a secondary index. The index contains the index key and the logical rowid of the index-organized table. The logical rowid excludes columns that are also part of the index key. You cannot specify `REVERSE` for this secondary index, and the combined size of the index key and the logical rowid should be less than the block size.
- If *`table`* is a temporary table, then *`index`* will also be temporary with the same scope (session or transaction) as *`table`*. The following restrictions apply to indexes on temporary tables:
  - The only part of *`index_properties`* you can specify is *`index_attributes`*.
  - Within *`index_attributes`*, you cannot specify the *`physical_attributes_clause`*, the *`parallel_clause`*, the *`logging_clause`*, or `TABLESPACE`.
  - You cannot create a domain index or a partitioned index on a temporary table.
- You cannot create an index on an external table.

> **See Also:**
>
> CREATE TABLE and *Oracle Database Concepts* for more information on temporary tables

t_alias

Specify a correlation name (alias) for the table upon which you are building the index.

> **Note:**
>
> This alias is required if the *`index_expr`* references any object type attributes or object type methods. See "Creating a Function-based Index on a Type Method: Example" and "Indexing on Substitutable Columns: Examples".

index_expr

For *`index_expr`*, specify the column or column expression upon which the index is based.

You can create multiple indexes on the same set of columns, column expressions, or both if the following conditions are met:

- The indexes are of different types, use different partitioning, or have different uniqueness properties.
- Only one of the indexes is `VISIBLE` at any given time.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on creating multiple indexes

column

Specify the name of one or more columns in the table. A bitmap index can have a maximum of 30 columns. Other indexes can have as many as 32 columns. These columns define the **index key**.

If a unique index is local nonprefixed (see *local_partitioned_index*), then the index key must contain the partitioning key.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for information on prefixed and nonprefixed indexes

You can create an index on a scalar object attribute column or on the system-defined NESTED_TABLE_ID column of the nested table storage table. If you specify an object attribute column, then the column name must be qualified with the table name. If you specify a nested table column attribute, then it must be qualified with the outermost table name, the containing column name, and all intermediate attribute names leading to the nested table column attribute.

When you create an index on a column or expression with a declared or derived named collation other than BINARY, or a declared or derived pseudo-collation USING_NLS_SORT_CI or USING_NLS_SORT_AI, the database creates a functional index on the function NLSSORT. See *Oracle Database Globalization Support Guide* for more information.

Creating an Index on an Extended Data Type Column

If *column* is an extended data type column, then you may receive a "maximum key length exceeded" error when attempting to create the index. The maximum key length for an index varies depending on the database block size and some additional index metadata stored in a block. For example, for databases that use the Oracle standard 8K block size, the maximum key length is approximately 6400 bytes.

To work around this situation, you must shorten the length of the values you want to index, using one of the following methods:

- Create a function-based index to shorten the values stored in the extended data type column as part of the expression used for the index definition.
- Create a virtual column to shorten the values stored in the extended data type column as part of the expression used for the virtual column definition and build a normal index on the virtual column. Using a virtual column also enables you to leverage functionality for regular columns, such as collecting statistics and using constraint and triggers.

For both methods you can use either the SUBSTR or STANDARD_HASH function to shorten the values of the extended data type column to build an index. These methods have the following advantages and disadvantages:

- Use the SUBSTR function to return a substring, or prefix, of *column* that is an acceptable length for the index key. This type of index can be used for equality, IN-list, and range predicates on the original column without the need to specify the SUBSTR column as part of the predicate. Refer to SUBSTR for more information.
- Using the STANDARD_HASH function is likely to create an index that is more compact than the substring-based index and may result in fewer unnecessary index accesses. This type of index can be used for equality and IN-list predicates on the original column without the need to specify the SUBSTR column as part of the predicate. Refer to STANDARD_HASH for more information.

The following example shows how to create a function-based index on an extended data type column:

```
CREATE INDEX index ON table (SUBSTR(column, 0, n));
```

For n, specify a prefix length that is large enough to differentiate between values in *column*.

The following example shows how to create a virtual column for an extended data type column, and then create an index on the virtual column:

```
ALTER TABLE table ADD (new_hash_column AS (STANDARD_HASH(column)));
CREATE INDEX index ON table (new_hash_column);
```

> **See Also:**
>
> "Extended Data Types" for more information on extended data types

Restrictions on Index Columns

The following restrictions apply to index columns:

- You cannot create an index on columns or attributes whose type is user-defined, `LONG`, `LONG RAW`, LOB, or `REF`, except that Oracle Database supports an index on `REF` type columns or attributes that have been defined with a `SCOPE` clause.
- Only normal (B-tree) indexes can be created on encrypted columns, and they can only be used for equality searches.

column_expression

Specify an expression built from columns of *`table`*, constants, SQL functions, and user-defined functions. When you specify *`column_expression`*, you create a **function-based index**.

> **See Also:**
>
> "Column Expressions ", "Notes on Function-based Indexes", "Restrictions on Function-based Indexes", and "Function-Based Index Examples"

Name resolution of the function is based on the schema of the index creator. User-defined functions used in *`column_expression`* are fully name resolved during the `CREATE INDEX` operation.

After creating a function-based index, collect statistics on both the index and its base table using the `DBMS_STATS` package. Such statistics will enable Oracle Database to correctly decide when to use the index.

Function-based unique indexes can be useful in defining a conditional unique constraint on a column or combination of columns. Refer to "Using a Function-based Index to Define Conditional Uniqueness: Example" for an example.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_STATS` package

Notes on Function-based Indexes

The following notes apply to function-based indexes:

- When you subsequently query a table that uses a function-based index, Oracle Database will not use the index unless the query filters out nulls. However, Oracle Database will use a function-based index in a query even if the columns specified in the WHERE clause are in a different order than their order in the *column_expression* that defined the function-based index.

> **See Also:**
>
> "Function-Based Index Examples"

- If the function on which the index is based becomes invalid or is dropped, then Oracle Database marks the index DISABLED. Queries on a DISABLED index fail if the optimizer chooses to use the index. DML operations on a DISABLED index fail unless the index is also marked UNUSABLE **and** the parameter SKIP_UNUSABLE_INDEXES is set to true. Refer to ALTER SESSION for more information on this parameter.
- If a public synonym for a function, package, or type is used in *column_expression*, and later an actual object with the same name is created in the table owner's schema, then Oracle Database disables the function-based index. When you subsequently enable the function-based index using ALTER INDEX ... ENABLE or ALTER INDEX ... REBUILD, the function, package, or type used in the *column_expression* continues to resolve to the function, package, or type to which the public synonym originally pointed. It will not resolve to the new function, package, or type.
- If the definition of a function-based index generates internal conversion to character data, then use caution when changing NLS parameter settings. Function-based indexes use the current database settings for NLS parameters. If you reset these parameters at the session level, then queries using the function-based index may return incorrect results. Two exceptions are the collation parameters (NLS_SORT and NLS_COMP). Oracle Database handles the conversions correctly even if these have been reset at the session level.
- Oracle Database cannot convert data in all cases, even when conversion is explicitly requested. For example, an attempt to convert the string '105 lbs' from VARCHAR2 to NUMBER using the TO_NUMBER function fails with an error. Therefore, if *column_expression* contains a data conversion function such as TO_NUMBER or TO_DATE, and if a subsequent INSERT or UPDATE statement includes data that the conversion function cannot convert, then the index will cause the INSERT or UPDATE statement to fail.
- If *column_expression* contains a datetime format model, then the function-based index expression defining the column may contain format elements that are different from those specified. For example, define a function-based index using the yyyy datetime format element:

  ```
  CREATE INDEX cust_eff_ix ON customers
    (NVL(cust_eff_to, TO_DATE('9000-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss')));
  ```

  Query the ALL_IND_EXPRESSIONS view to see that the function-based index expression defining the column uses the syyyy datetime format element:

  ```
  SELECT column_expression
    FROM all_ind_expressions
    WHERE index_name='CUST_EFF_IX';

  COLUMN_EXPRESSION
  ------------------------------------------------------------------------------
  NVL("CUST_EFF_TO",TO_DATE(' 9000-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
  ```

Restrictions on Function-based Indexes

Function-based indexes are subject to the following restrictions:

- The value returned by the function referenced in *column_expression* is subject to the same restrictions as are the index columns of a B-tree index. Refer to "Restrictions on Index Columns".
- Any user-defined function referenced in *column_expression* must be declared as DETERMINISTIC.
- For a function-based globally partitioned index, the *column_expression* cannot be the partitioning key.
- The *column_expression* can be any of the forms of expression described in Column Expressions .
- All functions must be specified with parentheses, even if they have no parameters. Otherwise Oracle Database interprets them as column names.
- Any function you specify in *column_expression* must return a repeatable value. For example, you cannot specify the SYSDATE or USER function or the ROWNUM pseudocolumn.

> **See Also:**
>
> CREATE FUNCTION and *Oracle Database PL/SQL Language Reference*

ASC | DESC

Use ASC or DESC to indicate whether the index should be created in ascending or descending order. Indexes on character data are created in ascending or descending order of the character values in the database character set.

Oracle Database treats descending indexes as if they were function-based indexes. As with other function-based indexes, the database does not use descending indexes until you first analyze the index and the table on which the index is defined. See the *column_expression* clause of this statement.

Ascending unique indexes allow multiple NULL values. However, in descending unique indexes, multiple NULL values are treated as duplicate values and therefore are not permitted.

Restriction on Ascending and Descending Indexes

You cannot specify either of these clauses for a domain index. You cannot specify DESC for a reverse index. Oracle Database ignores DESC if *index* is bitmapped or if the COMPATIBLE initialization parameter is set to a value less than 8.1.0.

index_attributes

Specify the optional index attributes.

physical_attributes_clause

Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the index.

If you omit this clause, then Oracle Database sets PCTFREE to 10 and INITRANS to 2.

Restriction on Index Physical Attributes

You cannot specify the PCTUSED parameter for an index.

> **See Also:**
>
> *physical_attributes_clause* and storage_clause for a complete description of these clauses

TABLESPACE

For *tablespace*, specify the name of the tablespace to hold the index, index partition, or index subpartition. If you omit this clause, then Oracle Database creates the index in the default tablespace of the owner of the schema containing the index.

For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions or subpartitions added to the local index will be created in the same tablespace(s) as the corresponding partitions or subpartitions of the underlying table.

index_compression

The *index_compression* clauses let you enable or disable index compression for the index. Specify the COMPRESS clause of *prefix_compression* to enable prefix compression for the index, specify the COMPRESS ADVANCED clause of *advanced_index_compression* to enable advanced index compression for the index, or specify the NOCOMPRESS clause of either *prefix_compression* or *advanced_index_compression* to disable compression for the index. The default is NOCOMPRESS.

If you want to use compression for a partitioned index, then you must create the index with compression enabled at the index level. You can subsequently enable and disable the compression setting for individual partitions of such a partitioned index. You can also enable and disable compression when rebuilding individual partitions. You can modify an existing nonpartitioned index to enable or disable compression only when rebuilding the index.

prefix_compression

Specify COMPRESS to enable **prefix compression**, also known as key compression, which eliminates repeated occurrence of key column values. Use *integer* to specify the prefix length (number of prefix columns to compress). You can specify prefix compression for indexes that are nonunique or unique indexes of at least two columns.

- For unique indexes, the range of valid prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the range of valid prefix length values is from 1 to the number of key columns. The default prefix length is the number of key columns.

advanced_index_compression

Specify this clause to enable **advanced index compression**. Advanced index compression improves compression ratios significantly while still providing efficient access to indexes. Therefore, advanced index compression works well on all supported indexes, including those indexes that are not good candidates for prefix compression.

- COMPRESS ADVANCED LOW - This level compresses the index less than the HIGH level, but provides faster access to the index. You can specify this clause for indexes that are nonunique or unique indexes of at least two columns. Before enabling COMPRESS ADVANCED LOW, the database must be at 12.1.0 or higher compatibility level.

- `COMPRESS ADVANCED HIGH` - This level compresses the index more than the `LOW` level, but provides slower access to the index. You can specify this clause for indexes that are nonunique or unique indexes of one or more columns. Before enabling `COMPRESS ADVANCED HIGH`, the database must be at `12.2.0` or higher compatibility level.

If you omit the `LOW` and `HIGH` keywords, then the default is `HIGH`.

Restrictions on Index Compression

The following restrictions apply to index compression:

- You cannot specify prefix compression or advanced index compression for a bitmap index.
- You cannot specify advanced index compression for index-organized tables.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information on prefix compression and advanced index compression
> - "Compressing an Index: Example"

partial_index_clause

You can specify this clause only when creating an index on a partitioned table. Specify `INDEXING FULL` to create a **full index**. Specify `INDEXING PARTIAL` to create a **partial index**. The default is `INDEXING FULL`.

A full index includes all partitions in the underlying table, regardless of their indexing properties. A partial index includes only partitions in the underlying table with an indexing property of `ON`.

If a partial index is a local partitioned index, then index partitions that correspond with table partitions with an indexing property of `ON` are marked `USABLE`. Index partitions that correspond with table partitions with an indexing property of `OFF` are marked `UNUSABLE`.

If the underlying table is a composite-partitioned table, then the preceding conditions for index partitions and table partitions apply instead to index subpartitions and table subpartitions.

Restrictions on Partial Indexes

Partial indexes are subject to the following restrictions:

- The underlying table of a partial index cannot be a nonpartitioned table.
- Unique indexes cannot be partial indexes. This applies to indexes created with the `CREATE UNIQUE INDEX` statement and indexes that are implicitly created when you specify a unique constraint on one or more columns.

> **See Also:**
>
> CREATE TABLE *indexing_clause* for information on the indexing property

SORT | NOSORT

By default, Oracle Database sorts indexes in ascending order when it creates the index. You can specify NOSORT to indicate to the database that the rows are already stored in the database in ascending order, so that Oracle Database does not have to sort the rows when creating the index. If the rows of the indexed column or columns are not stored in ascending order, then the database returns an error. For greatest savings of sort time and space, use this clause immediately after the initial load of rows into a table. If you specify neither of these keywords, then SORT is the default.

Restrictions on NOSORT

This parameter is subject to the following restrictions:

- You cannot specify REVERSE with this clause.
- You cannot use this clause to create a cluster index partitioned or bitmap index.
- You cannot specify this clause for a secondary index on an index-organized table.

REVERSE

Specify REVERSE to store the bytes of the index block in reverse order, excluding the rowid.

Restrictions on Reverse Indexes

Reverse indexes are subject to the following restrictions:

- You cannot specify NOSORT with this clause.
- You cannot reverse a bitmap index or an index on an index-organized table.

VISIBLE | INVISIBLE

Use this clause to specify whether the index is visible or invisible to the optimizer. An invisible index is maintained by DML operations, but it is not be used by the optimizer during queries unless you explicitly set the parameter OPTIMIZER_USE_INVISIBLE_INDEXES to TRUE at the session or system level.

To determine whether an existing index is visible or invisible to the optimizer, you can query the VISIBILITY column of the USER_, DBA_, ALL_INDEXES data dictionary views.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on this feature

logging_clause

Specify whether the creation of the index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. This setting also determines whether subsequent Direct Loader

(SQL*Loader) and direct-path `INSERT` operations against the index are logged or not logged. `LOGGING` is the default.

If *`index`* is nonpartitioned, then this clause specifies the logging attribute of the index.

If *`index`* is partitioned, then this clause determines:

- The default value of all partitions specified in the `CREATE` statement, unless you specify the *`logging_clause`* in the `PARTITION` description clause
- The default value for the segments associated with the index partitions
- The default value for local index partitions or subpartitions added implicitly during subsequent `ALTER TABLE` ... `ADD PARTITION` operations

The logging attribute of the index is independent of that of its base table.

If you omit this clause, then the logging attribute is that of the tablespace in which it resides.

> **See Also:**
>
> - *logging_clause* for a full description of this clause
> - *Oracle Database VLDB and Partitioning Guide* for more information about logging and parallel DML
> - "Creating an Index in NOLOGGING Mode: Example"

ONLINE

Specify `ONLINE` to indicate that DML operations on the table will be allowed during creation of the index.

Restrictions on Online Index Building

Online index building is subject to the following restrictions:

- Parallel DML is not supported during online index building. If you specify `ONLINE` and then issue parallel DML statements, then Oracle Database returns an error.
- You can specify `ONLINE` for a bitmap index or a cluster index as long as `COMPATIBLE` is set to 10 or higher.
- You cannot specify `ONLINE` for a conventional index on a `UROWID` column.
- For a nonunique secondary index on an index-organized table, the number of index key columns plus the number of primary key columns that are included in the logical rowid in the index-organized table cannot exceed 32. The logical rowid excludes columns that are part of the index key.

> **See Also:**
>
> *Oracle Database Concepts* for a description of online index building and rebuilding

parallel_clause

Specify the `parallel_clause` if you want creation of the index to be parallelized.

For complete information on this clause, refer to *parallel_clause* in the documentation on `CREATE TABLE`.

Index Partitioning Clauses

Use the `global_partitioned_index` clause and the `local_partitioned_index` clauses to partition `index`.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

> **See Also:**
>
> "Partitioned Index Examples"

global_partitioned_index

The `global_partitioned_index` clause lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes.

You can partition a global index by range or by hash. In both cases, you can specify up to 32 columns as partitioning key columns. The partitioning column list must specify a left prefix of the index column list. If the index is defined on columns `a`, `b`, and `c`, then for the columns you can specify `(a, b, c)`, or (`a, b)`, or (`a, c)`, but you cannot specify (`b, c)` or `(c)` or (`b, a`). If you specify a partition name, then it must conform to the rules for naming schema objects and their parts as described in "Database Object Naming Rules ". If you omit the partition names, then Oracle Database assigns names of the form `SYS_Pn`.

GLOBAL PARTITION BY RANGE

Use this clause to create a range-partitioned global index. Oracle Database will partition the global index on the ranges of values from the table columns you specify in the column list.

> **See Also:**
>
> "Creating a Range-Partitioned Global Index: Example"

GLOBAL PARTITION BY HASH

Use this clause to create a hash-partitioned global index. Oracle Database assigns rows to the partitions using a hash function on values in the partitioning key columns.

See Also:

The CREATE TABLE clause *hash_partitions* for information on the two methods of hash partitioning and "Creating a Hash-Partitioned Global Index: Example"

Restrictions on Global Partitioned Indexes

Global partitioned indexes are subject to the following restrictions:

- The partitioning key column list cannot contain the ROWID pseudocolumn or a column of type ROWID.
- The only property you can specify for hash partitions is tablespace storage. Therefore, you cannot specify LOB or varray storage clauses in the *partitioning_storage_clause* of *individual_hash_partitions*.
- You cannot specify the OVERFLOW clause of *hash_partitions_by_quantity*, as that clause is valid only for index-organized table partitions.
- In the *partitioning_storage_clause*, you cannot specify *table_compression* or the *inmemory_clause*, but you can specify *index_compression*.

Note:

If your enterprise has or will have databases using different character sets, then use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

See Also:

Oracle Database Globalization Support Guide for more information on character set support

index_partitioning_clause

Use this clause to describe the individual index partitions. The number of repetitions of this clause determines the number of partitions. If you omit *partition*, then Oracle Database generates a name with the form SYS_P*n*.

For VALUES LESS THAN (*value_list*), specify the noninclusive upper bound for the current partition in a global index. The value list is a comma-delimited, ordered list of literal values corresponding to the column list in the *global_partitioned_index* clause. Always specify MAXVALUE as the value of the last partition.

> **Note:**
>
> If the index is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, then you must use the TO_DATE function with a 4-character format mask for the year. The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT. Refer to *Oracle Database Globalization Support Guide* for more information on these initialization parameters.

> **See Also:**
>
> "Range Partitioning Example"

local_partitioned_index

The *local_partitioned_index* clauses let you specify that the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as *table*. For composite-partitioned tables, this clause lets you specify that the index is subpartitioned on the same columns, with the same number of subpartitions and the same subpartition bounds as *table*. Oracle Database automatically maintains local index partitioning as the underlying table is repartitioned.

If you specify only the keyword LOCAL and do not specify a subclause, then Oracle Database creates each index partition in the same tablespace as its corresponding table partition and assigns it the same name as its corresponding table partition. If *table* is a composite-partitioned table, then Oracle Database creates each index subpartition in the same tablespace as its corresponding table subpartition and assigns it the same name as its corresponding table subpartition.

If you specify a partition name, then it must conform to the rules for naming schema objects and their parts as described in "Database Object Naming Rules ". If you omit a partition name, then Oracle Database generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, then the database uses the form SYS_P*n*.

on_range_partitioned_table

This clause lets you specify the names and attributes of index partitions on a range-partitioned table. If you specify this clause, then the number of PARTITION clauses must be equal to the number of table partitions, and in the same order.

You cannot specify prefix compression for an index partition unless you have specified prefix compression for the index.

For more information on the USABLE and UNUSABLE clauses, refer to USABLE | UNUSABLE.

on_list_partitioned_table

The *on_list_partitioned_table* clause is identical to on_range_partitioned_table.

on_hash_partitioned_table

This clause lets you specify names and tablespace storage for index partitions on a hash-partitioned table.

If you specify any PARTITION clauses, then the number of these clauses must be equal to the number of table partitions. You can optionally specify tablespace storage for one or more individual partitions. If you do not specify tablespace storage either here or in the STORE IN clause, then the database stores each index partition in the same tablespace as the corresponding table partition.

The STORE IN clause lets you specify one or more tablespaces across which Oracle Database will distribute all the index hash partitions. The number of tablespaces need not equal the number of index partitions. If the number of index partitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

For more information on the USABLE and UNUSABLE clauses, refer to USABLE | UNUSABLE.

on_comp_partitioned_table

This clause lets you specify the name and attributes of index partitions on a composite-partitioned table.

The STORE IN clause is valid only for range-hash or list-hash composite-partitioned tables. It lets you specify one or more default tablespaces across which Oracle Database will distribute all index hash subpartitions for all partitions. You can override this storage by specifying different default tablespace storage for the subpartitions of an individual partition in the second STORE IN clause in the *index_subpartition_clause*.

For range-range, range-list, and list-list composite-partitioned tables, you can specify default attributes for the range or list subpartitions in the PARTITION clause. You can override this storage by specifying different attributes for the range or list subpartitions of an individual partition in the SUBPARTITION clause of the *index_subpartition_clause*.

You cannot specify prefix compression for an index partition unless you have specified prefix compression for the index.

For more information on the USABLE and UNUSABLE clauses, refer to USABLE | UNUSABLE.

index_subpartition_clause

This clause lets you specify names and tablespace storage for index subpartitions in a composite-partitioned table.

The STORE IN clause is valid only for hash subpartitions of a range-hash and list-hash composite-partitioned table. It lets you specify one or more tablespaces across which Oracle Database will distribute all the index hash subpartitions. The SUBPARTITION clause is valid for all subpartition types.

If you specify any SUBPARTITION clauses, then the number of those clauses must be equal to the number of table subpartitions. If you specify a subpartition name, then it must conform to the rules for naming schema objects and their parts as described in "Database Object Naming Rules ". If you omit *subpartition*, then the database generates a name that is consistent with the corresponding table subpartition. If the name conflicts with an existing index subpartition name, then the database uses the form SYS_SUBP*n*.

The number of tablespaces need not equal the number of index subpartitions. If the number of index subpartitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

If you do not specify tablespace storage for subpartitions either in the `on_comp_partitioned_table` clause or in the `index_subpartition_clause`, then Oracle Database uses the tablespace specified for `index`. If you also do not specify tablespace storage for `index`, then the database stores the subpartition in the same tablespace as the corresponding table subpartition.

For more information on the `USABLE` and `UNUSABLE` clauses, refer to `CREATE INDEX` ... USABLE | UNUSABLE.

domain_index_clause

Use the `domain_index_clause` to indicate that `index` is a domain index, which is an instance of an application-specific index of type `indextype`.

Creating a domain index requires a number of preceding operations. You must first create an implementation type for an indextype. You must also create a functional implementation and then create an operator that uses the function. Next you create an indextype, which associates the implementation type with the operator. Finally, you create the domain index using this clause. Refer to Extended Examples, which contains an example of creating a simple domain index, including all of these operations.

index_expr

In the `index_expr` (in `table_index_clause`), specify the table columns or object attributes on which the index is defined. You can define multiple domain indexes on a single column only if the underlying indextypes are different and the indextypes support a disjoint set of user-defined operators.

Restrictions on Domain Indexes

Domain indexes are subject to the following restrictions:

- The `index_expr` (in `table_index_clause`) can specify only a single column, and the column cannot be of data type `REF`, varray, nested table, `LONG`, or `LONG RAW`.
- You cannot create a bitmap or unique domain index.
- You cannot create a domain index on a temporary table.
- You can create local domain indexes on only range-, list-, hash-, and interval-partitioned tables, with one exception: You cannot create a local domain index on an automatic list-partitioned table.
- Domain indexes can be created only on table columns declared with collation `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`. See *Oracle Database Globalization Support Guide* for more information.

indextype

For `indextype`, specify the name of the indextype. This name should be a valid schema object that has already been created.

If you have installed Oracle Text, then you can use various built-in indextypes to create Oracle Text domain indexes. For more information on Oracle Text and the indexes it uses, refer to *Oracle Text Reference*.

> **See Also:**
>
> CREATE INDEXTYPE

local_domain_index_clause

Use this clause to specify that the index is a local index on a partitioned table.

- The `PARTITIONS` clause lets you specify names for the index partitions. The number of partitions you specify must match the number of partitions in the base table. If you omit this clause, then the database creates the partitions with system-generated names of the form `SYS_P`*`n`*.
- The `PARAMETERS` clause lets you specify the parameter string specific to an individual partition. If you omit this clause, then the parameter string associated with the index is also associated with the partition.

parallel_clause

Use the *`parallel_clause`* to parallelize creation of the domain index. For a nonpartitioned domain index, Oracle Database passes the explicit or default degree of parallelism to the `ODCIIndexCreate` cartridge routine, which in turn establishes parallelism for the index. For local domain indexes, this clause causes the index partitions to be created in parallel.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for complete information on the Oracle Data Cartridge Interface (ODCI) routines

PARAMETERS

In the `PARAMETERS` clause, specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the default parameters for the index partitions. If you specify this clause as part of the *`local_domain_index_clause`*, then you override any default parameters with parameters for the individual partition.

After the domain index is created, Oracle Database invokes the appropriate ODCI routine. If the routine does not return successfully, then the domain index is marked `FAILED`. The only operations supported on an failed domain index are `DROP INDEX` and (for non-local indexes) `REBUILD INDEX`.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information on the Oracle Data Cartridge Interface (ODCI) routines

XMLIndex_clause

The `XMLIndex_clause` lets you define an `XMLIndex` index, typically on a column contain XML data. An `XMLIndex` index is a type of domain index designed specifically for the domain of XML data.

XMLIndex_parameters_clause

This clause lets you specify information about the path table and about the secondary indexes corresponding to the components of `XMLIndex`. This clause also lets you specify information about the structured component of the index. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the parameters of the index and the default parameters for the index partitions. If you specify this clause as part of the `local_xmlindex_clause` clause, then you override any default parameters with parameters for the individual partition.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for the syntax and semantics of the `XMLIndex_parameters_clause`, as well as detailed information about the use of `XMLIndex`

bitmap_join_index_clause

Use the `bitmap_join_index_clause` to define a **bitmap join index**. A bitmap join index is defined on a single table. For an index key made up of dimension table columns, it stores the fact table rowids corresponding to that key. In a data warehousing environment, the table on which the index is defined is commonly referred to as a **fact table**, and the tables with which this table is joined are commonly referred to as **dimension tables**. However, a star schema is not a requirement for creating a join index.

ON

In the `ON` clause, first specify the fact table, and then inside the parentheses specify the columns of the dimension tables on which the index is defined.

FROM

In the `FROM` clause, specify the joined tables.

WHERE

In the `WHERE` clause, specify the join condition.

If the underlying fact table is partitioned, then you must also specify one of the `local_partitioned_index` clauses (see *local_partitioned_index*).

Restrictions on Bitmap Join Indexes

In addition to the restrictions on bitmap indexes in general (see BITMAP), the following restrictions apply to bitmap join indexes:

- You cannot create a bitmap join index on a temporary table.

- No table may appear twice in the FROM clause.
- You cannot create a function-based join index.
- The dimension table columns must be either primary key columns or have unique constraints.
- If a dimension table has a composite primary key, then each column in the primary key must be part of the join.
- You cannot specify the *local_partitioned_index* clause unless the fact table is partitioned.
- A bitmap join index definition can only reference columns with collation BINARY, USING_NLS_COMP, USING_NLS_SORT, or USING_NLS_SORT_CS. For any of these collations, index keys are collated and the join condition is evaluated using the BINARY collation. See *Oracle Database Globalization Support Guide* for more information.

> **Note:**
>
> *Oracle Database Data Warehousing Guide* for information on fact and dimension tables and on using bitmap indexes in a data warehousing environment

USABLE | UNUSABLE

You can specify the USABLE and UNUSABLE keywords:

- For an index, in the CREATE INDEX statement
- For an index partition, in the *on_range_partitioned_table*, *on_list_partitioned_table*, *on_hash_partitioned_table*, and *on_comp_partitioned_table* clauses
- For an index subpartition, in the *index_subpartition_clause*

For nonpartitioned indexes, specify UNUSABLE to create an index in an unusable state. An unusable index must be rebuilt, or dropped and re-created, before it can be used. Specify USABLE to create an index in a usable state. USABLE is the default.

For partitioned indexes, specify USABLE or UNUSABLE as follows:

- If you specify UNUSABLE for the index, then all index partitions are marked UNUSABLE.
- If you specify USABLE for the index, then all index partitions are marked USABLE.
- If you do not specify USABLE or UNUSABLE for the index, then all index partitions are marked USABLE. The exception is a local partial index. If you specify the LOCAL and INDEXING PARTIAL clauses, and do not specify USABLE or UNUSABLE, then each index partition is marked USABLE if the indexing property of its corresponding table partition is ON, or UNUSABLE if the indexing property of its corresponding table partition is OFF.

You can override the preceding conditions by specifying USABLE or UNUSABLE for a specific index partition.

If the underlying table is a composite-partitioned table, then the preceding conditions for index partitions and table partitions apply instead to index subpartitions and table subpartitions.

After you create a partitioned index, you can choose to rebuild specific index partitions or subpartitions to make them `USABLE`. Doing so can be useful if you want to maintain indexes only on some index partitions or subpartitions—for example, if you want to enable index access for new partitions but not for old partitions.

When an index, or some partitions or subpartitions of an index, are created `UNUSABLE`, no segment is allocated for the unusable object. The unusable index or index partition consumes no space in the database.

If an index, or some partitions or subpartitions of the index, are marked `UNUSABLE`, then the index will be considered as an access path by the optimizer only under the following circumstances: the optimizer must know at compile time which partitions are to be accessed, and all of those partitions to be accessed must be marked `USABLE`. Therefore, the query cannot contain any bind variables.

Restrictions on USABLE | UNUSABLE

The following restrictions apply when marking an index `USABLE` or `UNUSABLE`:

- You cannot specify this clause for an index on a temporary table.
- Unusable indexes or index partitions will still have a segment under the following conditions:
  - The index (or index partition) is owned by `SYS`, `SYSTEM`, `PUBLIC`, `OUTLN`, or `XDB`
  - The index (or index partition) is stored in dictionary-managed tablespaces
  - The global partitioned or nonpartitioned index on a partitioned table becomes unusable due to a partition maintenance operation

{ DEFERRED | IMMEDIATE } INVALIDATION

This clause lets you control when the database invalidates dependent cursors while creating the index. It has the same semantics here as for the `ALTER INDEX` statement. Refer to { DEFERRED | IMMEDIATE } INVALIDATION in the documentation on `ALTER INDEX` for the full semantics of this clause.

Examples

General Index Examples

Creating an Index: Example

The following statement shows how the sample index `ord_customer_ix` on the `customer_id` column of the sample table `oe.orders` was created:

```
CREATE INDEX ord_customer_ix
   ON orders (customer_id);
```

Compressing an Index: Example

To create the `ord_customer_ix_demo` index with the `COMPRESS` clause, you might issue the following statement:

```
CREATE INDEX ord_customer_ix_demo
   ON orders (customer_id, sales_rep_id)
   COMPRESS 1;
```

The index will compress repeated occurrences of `customer_id` column values.

Creating an Index in NOLOGGING Mode: Example

If the sample table `orders` had been created using a fast parallel load (so all rows were already sorted), then you could issue the following statement to quickly create an index.

```
/* Unless you first sort the table oe.orders, this example fails
   because you cannot specify NOSORT unless the base table is
   already sorted.
*/
CREATE INDEX ord_customer_ix_demo
   ON orders (order_mode)
   NOSORT
   NOLOGGING;
```

Creating a Cluster Index: Example

To create an index for the `personnel` cluster, which was created in "Creating a Cluster: Example", issue the following statement:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

No index columns are specified, because cluster indexes are automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

Creating an Index on an XMLType Table: Example

The following example creates an index on the area element of the `xwarehouses` table (created in "XMLType Table Examples"):

```
CREATE INDEX area_index ON xwarehouses e
   (EXTRACTVALUE(VALUE(e),'/Warehouse/Area'));
```

Such an index would greatly improve the performance of queries that select from the table based on, for example, the square footage of a warehouse, as shown in this statement:

```
SELECT e.getClobVal() AS warehouse
   FROM xwarehouses e
   WHERE EXISTSNODE(VALUE(e),'/Warehouse[Area>50000]') = 1;
```

> **See Also:**
>
> EXISTSNODE and VALUE

Function-Based Index Examples

The following examples show how to create and use function-based indexes.

Creating a Function-Based Index: Example

The following statement creates a function-based index on the `employees` table based on an uppercase evaluation of the `last_name` column:

```
CREATE INDEX upper_ix ON employees (UPPER(last_name));
```

See the "Prerequisites " for the privileges and parameter settings required when creating function-based indexes.

To increase the likelihood that Oracle Database will use the index rather than performing a full table scan, be sure that the value returned by the function is not null in subsequent queries. For example, this statement will use the index, unless some other condition exists that prevents the optimizer from doing so:

```
SELECT first_name, last_name
   FROM employees WHERE UPPER(last_name) IS NOT NULL
   ORDER BY UPPER(last_name);
```

Without the `WHERE` clause, Oracle Database may perform a full table scan.

In the next statements showing index creation and subsequent query, Oracle Database will use index `income_ix` even though the columns are in reverse order in the query:

```
CREATE INDEX income_ix
   ON employees(salary + (salary*commission_pct));

SELECT first_name||' '||last_name "Name"
   FROM employees
   WHERE (salary*commission_pct) + salary > 15000
   ORDER BY employee_id;
```

Creating a Function-Based Index on a LOB Column: Example

The following statement uses the `text_length` function to create a function-based index on a LOB column in the sample `pm` schema. See *Oracle Database PL/SQL Language Reference* for the example that creates this function. The example selects rows from the sample table `print_media` where that `CLOB` column has fewer than 1000 characters.

```
CREATE INDEX src_idx ON print_media(text_length(ad_sourcetext));

SELECT product_id FROM print_media
   WHERE text_length(ad_sourcetext) < 1000
   ORDER BY product_id;

PRODUCT_ID
----------
      2056
      2268
      3060
      3106
```

Creating a Function-based Index on a Type Method: Example

This example entails an object type `rectangle` containing two number attributes: `length` and `width`. The `area()` method computes the area of the rectangle.

```
CREATE TYPE rectangle AS OBJECT
( length   NUMBER,
  width    NUMBER,
  MEMBER FUNCTION area RETURN NUMBER DETERMINISTIC
);
```

```
CREATE OR REPLACE TYPE BODY rectangle AS
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
   RETURN (length*width);
  END;
END;
```

Now, if you create a table `rect_tab` of type `rectangle`, you can create a function-based index on the `area()` method as follows:

```
CREATE TABLE rect_tab OF rectangle;
CREATE INDEX area_idx ON rect_tab x (x.area());
```

You can use this index efficiently to evaluate a query of the form:

```
SELECT * FROM rect_tab x WHERE x.area() > 100;
```

Using a Function-based Index to Define Conditional Uniqueness: Example

The following statement creates a unique function-based index on the `oe.orders` table that prevents a customer from taking advantage of promotion ID 2 ("blowout sale") more than once:

```
CREATE UNIQUE INDEX promo_ix ON orders
   (CASE WHEN promotion_id =2 THEN customer_id ELSE NULL END,
    CASE WHEN promotion_id = 2 THEN promotion_id ELSE NULL END);

INSERT INTO orders (order_id, order_date, customer_id, order_total, promotion_id)
   VALUES (2459, systimestamp, 106, 251, 2);
1 row created.

INSERT INTO orders (order_id, order_date, customer_id, order_total, promotion_id)
   VALUES (2460, systimestamp+1, 106, 110, 2);
insert into orders (order_id, order_date, customer_id, order_total, promotion_id)
*
ERROR at line 1:
ORA-00001: unique constraint (OE.PROMO_IX) violated
```

The objective is to remove from the index any rows where the `promotion_id` is not equal to 2. Oracle Database does not store in the index any rows where all the keys are `NULL`. Therefore, in this example, both `customer_id` and `promotion_id` are mapped to `NULL` unless promotion_id is equal to 2. The result is that the index constraint is violated only if promotion_id is equal to 2 for two rows with the same `customer_id` value.

Partitioned Index Examples

Creating a Range-Partitioned Global Index: Example

The following statement creates a global prefixed index `cost_ix` on the sample table `sh.sales` with three partitions that divide the range of costs into three groups:

```
CREATE INDEX cost_ix ON sales (amount_sold)
   GLOBAL PARTITION BY RANGE (amount_sold)
      (PARTITION p1 VALUES LESS THAN (1000),
       PARTITION p2 VALUES LESS THAN (2500),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Creating a Hash-Partitioned Global Index: Example

The following statement creates a hash-partitioned global index `cust_last_name_ix` on the sample table `sh.customers` with four partitions:

```
CREATE INDEX cust_last_name_ix ON customers (cust_last_name)
  GLOBAL PARTITION BY HASH (cust_last_name)
  PARTITIONS 4;
```

Creating an Index on a Hash-Partitioned Table: Example

The following statement creates a local index on the `category_id` column of the `hash_products` partitioned table (which was created in "Hash Partitioning Example"). The `STORE IN` clause immediately following `LOCAL` indicates that `hash_products` is hash partitioned. Oracle Database will distribute the hash partitions between the `tbs1` and `tbs2` tablespaces:

```
CREATE INDEX prod_idx ON hash_products(category_id) LOCAL
   STORE IN (tbs_01, tbs_02);
```

The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACE for examples that create tablespaces `tbs_01` and `tbs_02`.

Creating an Index on a Composite-Partitioned Table: Example

The following statement creates a local index on the `composite_sales` table, which was created in "Composite-Partitioned Table Examples". The `STORAGE` clause specifies default storage attributes for the index. However, this default is overridden for the five subpartitions of partitions `q3_2000` and `q4_2000`, because separate `TABLESPACE` storage is specified.

The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACE for examples that create tablespaces `tbs_02` and `tbs_03`.

```
CREATE INDEX sales_ix ON composite_sales(time_id, prod_id)
   STORAGE (INITIAL 1M)
   LOCAL
   (PARTITION q1_1998,
    PARTITION q2_1998,
    PARTITION q3_1998,
    PARTITION q4_1998,
    PARTITION q1_1999,
    PARTITION q2_1999,
    PARTITION q3_1999,
    PARTITION q4_1999,
    PARTITION q1_2000,
    PARTITION q2_2000
      (SUBPARTITION pq2001, SUBPARTITION pq2002,
       SUBPARTITION pq2003, SUBPARTITION pq2004,
       SUBPARTITION pq2005, SUBPARTITION pq2006,
       SUBPARTITION pq2007, SUBPARTITION pq2008),
    PARTITION q3_2000
      (SUBPARTITION c1 TABLESPACE tbs_02,
       SUBPARTITION c2 TABLESPACE tbs_02,
       SUBPARTITION c3 TABLESPACE tbs_02,
       SUBPARTITION c4 TABLESPACE tbs_02,
       SUBPARTITION c5 TABLESPACE tbs_02),
    PARTITION q4_2000
      (SUBPARTITION pq4001 TABLESPACE tbs_03,
       SUBPARTITION pq4002 TABLESPACE tbs_03,
       SUBPARTITION pq4003 TABLESPACE tbs_03,
       SUBPARTITION pq4004 TABLESPACE tbs_03)
);
```

Bitmap Index Examples

The following creates a bitmap index on the table `oe.hash_products`, which was created in "Hash Partitioning Example":

```
CREATE BITMAP INDEX product_bm_ix
   ON hash_products(list_price)
   LOCAL(PARTITION ix_p1 TABLESPACE tbs_01,
         PARTITION ix_p2,
         PARTITION ix_p3 TABLESPACE tbs_02,
         PARTITION ix_p4 TABLESPACE tbs_03)
   TABLESPACE tbs_04;
```

Because `hash_products` is a partitioned table, the bitmap join index must be locally partitioned. In this example, the user must have quota on tablespaces specified. See CREATE TABLESPACE for examples that create tablespaces `tbs_01`, `tbs_02`, `tbs_03`, and `tbs_04`.

The next series of statements shows how one might create a bitmap join index on a fact table using a join with a dimension table.

```
CREATE TABLE hash_products
    ( product_id          NUMBER(6)
    , product_name        VARCHAR2(50)
    , product_description VARCHAR2(2000)
    , category_id         NUMBER(2)
    , weight_class        NUMBER(1)
    , warranty_period     INTERVAL YEAR TO MONTH
    , supplier_id         NUMBER(6)
    , product_status      VARCHAR2(20)
    , list_price          NUMBER(8,2)
    , min_price           NUMBER(8,2)
    , catalog_url         VARCHAR2(50)
    , CONSTRAINT          pk_product_id PRIMARY KEY (product_id)
    , CONSTRAINT          product_status_lov_demo
                          CHECK (product_status in ('orderable'
                                                  ,'planned'
                                                  ,'under development'
                                                  ,'obsolete')
 ) )
 PARTITION BY HASH (product_id)
 PARTITIONS 5
 STORE IN (example);

CREATE TABLE sales_quota
    ( product_id          NUMBER(6)
    , customer_name       VARCHAR2(50)
    , order_qty           NUMBER(6)
  ,CONSTRAINT u_product_id UNIQUE(product_id)
 );

CREATE BITMAP INDEX product_bm_ix
   ON hash_products(list_price)
   FROM hash_products h, sales_quota s
   WHERE h.product_id = s.product_id
   LOCAL(PARTITION ix_p1 TABLESPACE example,
         PARTITION ix_p2,
         PARTITION ix_p3 TABLESPACE example,
         PARTITION ix_p4,
         PARTITION ix_p5 TABLESPACE example)
   TABLESPACE example;
```

Indexes on Nested Tables: Example

The sample table `pm.print_media` contains a nested table column `ad_textdocs_ntab`, which is stored in storage table `textdocs_nestedtab`. The following example creates a unique index on storage table `textdocs_nestedtab`:

```
CREATE UNIQUE INDEX nested_tab_ix
      ON textdocs_nestedtab(NESTED_TABLE_ID, document_typ);
```

Including pseudocolumn `NESTED_TABLE_ID` ensures distinct rows in nested table column `ad_textdocs_ntab`.

Indexing on Substitutable Columns: Examples

You can build an index on attributes of the declared type of a substitutable column. In addition, you can reference the subtype attributes by using the appropriate `TREAT` function. The following example uses the table `books`, which is created in "Substitutable Table and Column Examples". The statement creates an index on the `salary` attribute of all employee authors in the `books` table:

```
CREATE INDEX salary_i
   ON books (TREAT(author AS employee_t).salary);
```

The target type in the argument of the `TREAT` function must be the type that added the attribute being referenced. In the example, the target of `TREAT` is `employee_t`, which is the type that added the `salary` attribute.

If this condition is not satisfied, then Oracle Database interprets the `TREAT` function as any functional expression and creates the index as a function-based index. For example, the following statement creates a function-based index on the `salary` attribute of part-time employees, assigning nulls to instances of all other types in the type hierarchy.

```
CREATE INDEX salary_func_i ON persons p
   (TREAT(VALUE(p) AS part_time_emp_t).salary);
```

You can also build an index on the type-discriminant column underlying a substitutable column by using the `SYS_TYPEID` function.

> **Note:**
>
> Oracle Database uses the type-discriminant column to evaluate queries that involve the `IS OF` *`type`* condition. The cardinality of the typeid column is normally low, so Oracle recommends that you build a bitmap index in this situation.

The following statement creates a bitmap index on the typeid of the author column of the books table:

```
CREATE BITMAP INDEX typeid_i ON books (SYS_TYPEID(author));
```

See Also:

- *Oracle Database PL/SQL Language Reference* to see the creation of the type hierarchy underlying the `books` table
- the functions TREAT and SYS_TYPEID and the condition "IS OF *type* Condition "

CREATE INDEXTYPE

Purpose

Use the `CREATE INDEXTYPE` statement to create an **indextype**, which is an object that specifies the routines that manage a domain (application-specific) index. Indextypes reside in the same namespace as tables, views, and other schema objects. This statement binds the indextype name to an implementation type, which in turn specifies and refers to user-defined index functions and procedures that implement the indextype.

See Also:

Oracle Database Data Cartridge Developer's Guide for more information on implementing indextypes

Prerequisites

To create an indextype in your own schema, you must have the `CREATE INDEXTYPE` system privilege. To create an indextype in another schema, you must have the `CREATE ANY INDEXTYPE` system privilege. In either case, you must have the `EXECUTE` object privilege on the implementation type and the supported operators.

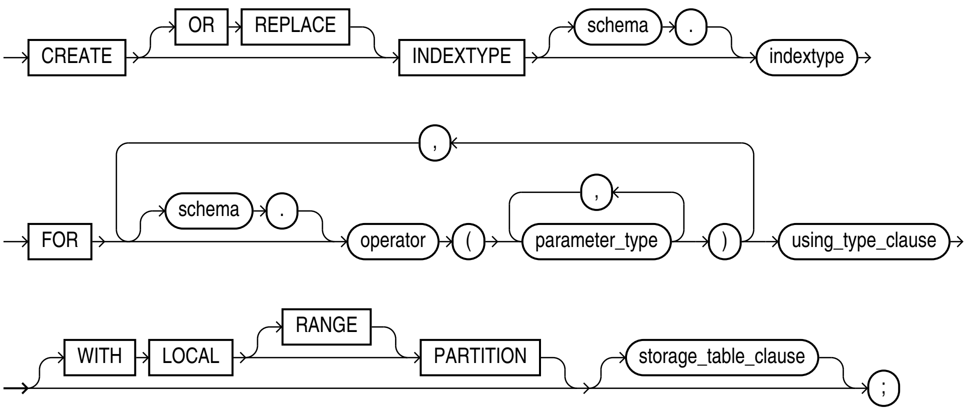
An indextype supports one or more operators, so before creating an indextype, you must first design the operator or operators to be supported and provide functional implementation for those operators.

See Also:

CREATE OPERATOR

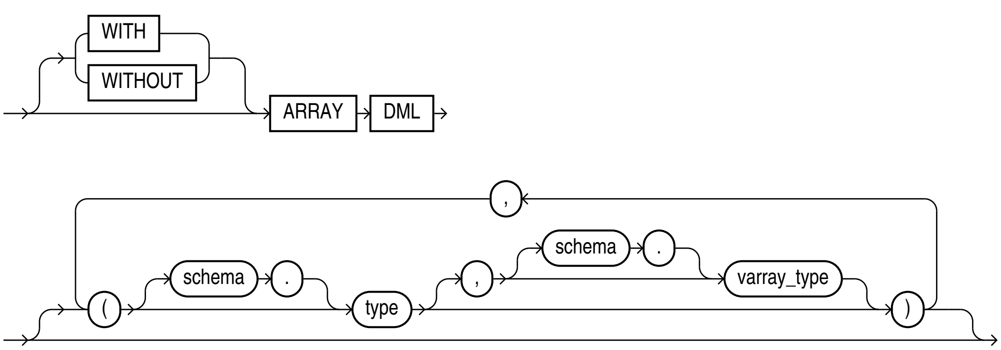
Syntax

***create_indextype*::=**

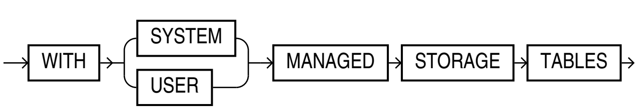


***using_type_clause*::=**

***array_DML_clause*::=**



***storage_table_clause*::=**



Semantics

schema

Specify the name of the schema in which the indextype resides. If you omit `schema`, then Oracle Database creates the indextype in your own schema.

indextype

Specify the name of the indextype to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

FOR Clause

Use the `FOR` clause to specify the list of operators supported by the indextype.

- For `schema`, specify the schema containing the operator. If you omit `schema`, then Oracle assumes the operator is in your own schema.
- For `operator`, specify the name of the operator supported by the indextype.

  All the operators listed in this clause must be valid operators.
- For `parameter_type`, list the types of parameters to the operator.

using_type_clause

The `USING` clause lets you specify the type that provides the implementation for the new indextype.

For `implementation_type`, specify the name of the type that implements the appropriate Oracle Data Cartridge Interface (ODCI).

- You must specify a valid type that implements the routines in the ODCI.
- The implementation type must reside in the same schema as the indextype.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for additional information on this interface

WITH LOCAL PARTITION

Use this clause to indicate that the indextype can be used to create local domain indexes on range-, list-, hash-, and interval-partitioned tables. You use this clause in combination with the `storage_table_clause` in several ways (see *storage_table_clause*).

- The recommended method is to specify `WITH LOCAL PARTITION WITH SYSTEM MANAGED STORAGE TABLES`. This combination uses *system-managed* storage tables, which are the preferred storage management, and lets you create local domain indexes on range-, list-, hash-, and interval-partitioned tables. In this case the `RANGE` keyword is optional and ignored, because it is no longer needed if you specify `WITH LOCAL PARTITION WITH SYSTEM MANAGED STORAGE TABLES`.

- You can specify `WITH LOCAL RANGE PARTITION` (including the `RANGE` keyword) and omit the *`storage_table`* clause. Local domain indexes on range-partitioned tables are supported with *user-managed* storage tables for backward compatibility. Oracle does not recommend this combination because it uses the less efficient user-managed storage tables.

If you omit this clause entirely, then you cannot subsequently use this indexype to create a local domain index on a range, list-, hash-, or interval-partitioned table.

storage_table_clause

Use this clause to specify how storage tables and partition maintenance operations for indexes built on this indextype are managed:

- Specify `WITH SYSTEM MANAGED STORAGE TABLES` to indicate that the storage of statistics data is to be managed by the system. The type you specify in *`statistics_type`* should be storing the statistics related information in tables that are maintained by the system. Also, the indextype you specify must already have been created or altered to support the `WITH SYSTEM MANAGED STORAGE TABLES` clause.
- Specify `WITH USER MANAGED STORAGE TABLES` to indicate that the tables that store the user-defined statistics will be managed by the user. This is the default behavior.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about storage tables for domain indexes

array_DML_clause

Use this clause to let the indextype support the array interface for the `ODCIIndexInsert` method.

type and *varray_type*

If the data type of the column to be indexed is a user-defined object type, then you must specify this clause to identify the varray *`varray_type`* that Oracle should use to hold column values of *`type`*. If the indextype supports a list of types, then you can specify a corresponding list of varray types. If you omit *`schema`* for either *`type`* or *`varray_type`*, then Oracle assumes the type is in your own schema.

If the data type of the column to be indexed is a built-in system type, then any varray type specified for the indextype takes precedence over the ODCI types defined by the system.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on the ODCI array interface

Examples

Creating an Indextype: Example

The following statement creates an indextype named `position_indextype` and specifies the `position_between` operator that is supported by the indextype and the `position_im` type that implements the index interface. Refer to "Using Extensible Indexing " for an extensible indexing scenario that uses this indextype:

```
CREATE INDEXTYPE position_indextype
   FOR position_between(NUMBER, NUMBER, NUMBER)
   USING position_im;
```

CREATE INMEMORY JOIN GROUP

Purpose

Use the `CREATE INMEMORY JOIN GROUP` statement to create a join group, which is an object that specifies frequently joined columns from the same table or different tables. Such columns typically contain values of compatible data types that fall in similar ranges. When you create a join group, Oracle Database stores special metadata for the columns in the global dictionary, which enables the database to optimize join queries for the columns. In order to achieve this optimization, the table columns must be populated in the In-Memory Column Store (IM column store).

Creating a join group for tables causes the current In-Memory contents of these tables to be invalidated. Subsequent repopulation causes the In-Memory Compression Units (IMCUs) of the tables to be re-encoded with the global dictionary. Thus, Oracle recommends that you first create the join group, and then populate the tables.

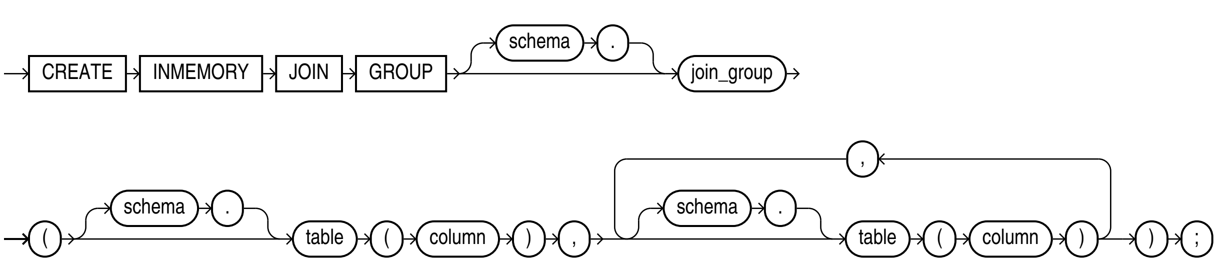
> **See Also:**
>
> - ALTER INMEMORY JOIN GROUP and DROP INMEMORY JOIN GROUP
> - *Oracle Database In-Memory Guide* for more information on join groups

Prerequisites

To create a join group in another user's schema, or to include in the join group a column in a table in another user's schema, you must have the `CREATE ANY TABLE` system privilege.

Syntax

***create_inmemory_join_group*::=**



Semantics

schema

Specify the schema to contain the join group. If you omit *schema*, then the database creates the join group in your own schema.

join_group

Specify the name of the join group to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

schema

Specify the schema of the table that contains a column to be included in the join group If you omit *schema*, then Oracle Database assumes the table is in your own schema.

table

Specify the name of the table that contains a column to be included in the join group.

column

Specify the name of a column to be included in the join group. A join group can contain columns in the same table or different tables.

Restrictions on Join Groups

The following restrictions apply to join groups:

- A join group must contain at least 2 columns.
- A join group can contain at most 255 columns.
- A table column can be a member of at most one join group.
- Oracle Active Data Guard does not support join groups.

Examples

The following statement creates a join group named `prod_id1` in the `oe` schema. Both tables involved in this join group reside in the `oe` schema.

```
CREATE INMEMORY JOIN GROUP prod_id1
  (inventories(product_id), order_items(product_id));
```

The following statement creates a join group named `prod_id2` in the `oe` schema. The table `inventories` resides in the `oe` schema and the table `online_media` resides in the `pm` schema.

```
CREATE INMEMORY JOIN GROUP prod_id2
  (inventories(product_id), pm.online_media(product_id));
```

CREATE JAVA

Purpose

Use the `CREATE JAVA` statement to create a schema object containing a Java source, class, or resource.

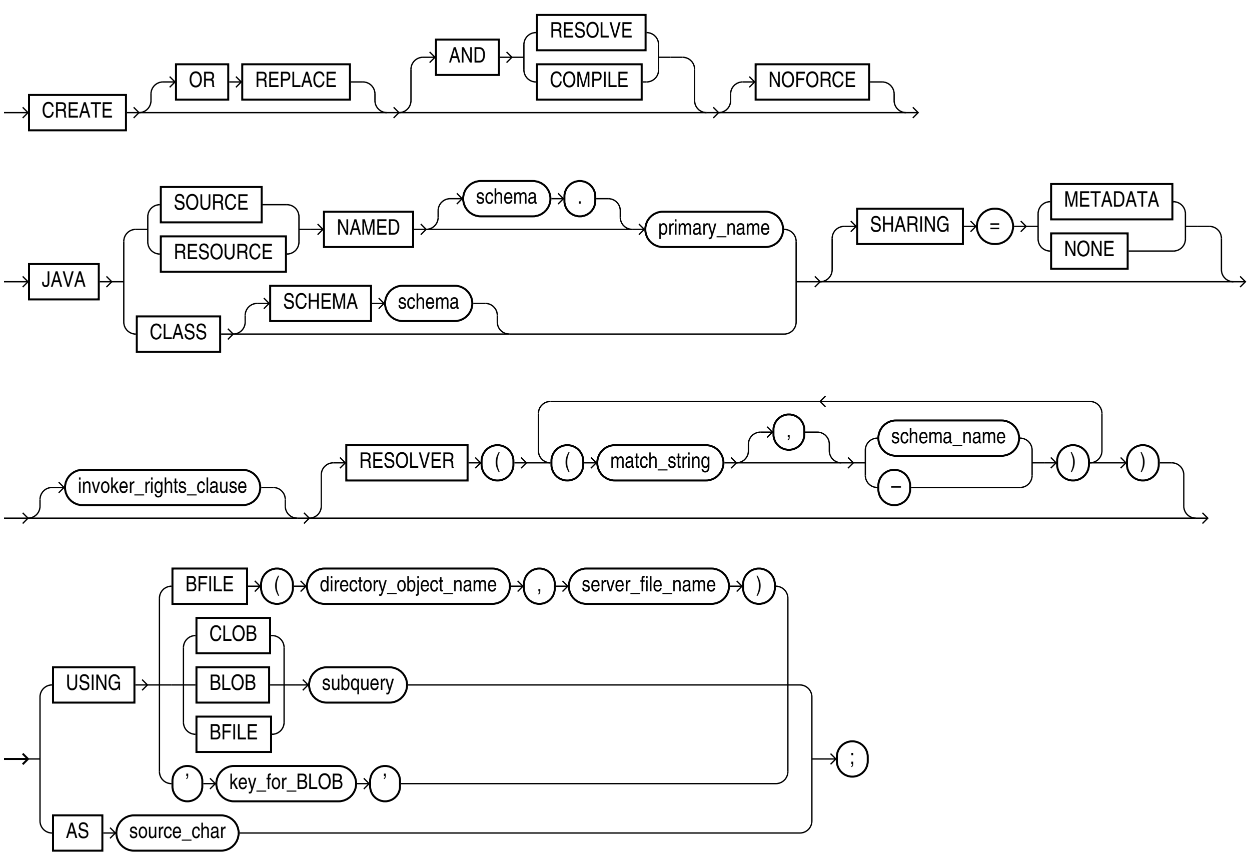
> **See Also:**
>
> - *Oracle Database Java Developer's Guide* for Java concepts and information about Java stored procedures
> - *Oracle Database JDBC Developer's Guide* for information on JDBC

Prerequisites

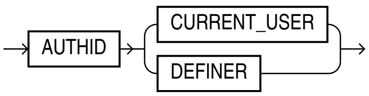
To create or replace a schema object containing a Java source, class, or resource in your own schema, you must have `CREATE PROCEDURE` system privilege. To create or replace such a schema object in another user's schema, you must have `CREATE ANY PROCEDURE` system privilege.

Syntax

***create_java*::=**



***invoker_rights_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the schema object containing the Java class, source, or resource if it already exists. Use this clause to change the definition of an existing object without dropping, re-creating, and regranting object privileges previously granted.

If you redefine a Java schema object and specify `RESOLVE` or `COMPILE`, then Oracle Database recompiles or resolves the object. Whether or not the resolution or compilation is successful, the database invalidates classes that reference the Java schema object.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

> **See Also:**
>
> ALTER JAVA for additional information

RESOLVE | COMPILE

`RESOLVE` and `COMPILE` are synonymous keywords. They specify that Oracle Database should attempt to resolve the Java schema object that is created if this statement succeeds.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

Restriction on RESOLVE and COMPILE

You cannot specify these keywords for a Java resource.

NOFORCE

Specify `NOFORCE` to roll back the results of this `CREATE` command if you have specified either `RESOLVE` or `COMPILE` and the resolution or compilation fails. If you do not specify this option, then Oracle Database takes no action if the resolution or compilation fails, and the created schema object remains.

JAVA SOURCE Clause

Specify `JAVA SOURCE` to load a Java source file.

JAVA CLASS Clause

Specify `JAVA CLASS` to load a Java class file.

JAVA RESOURCE Clause

Specify `JAVA RESOURCE` to load a Java resource file.

NAMED Clause

The `NAMED` clause is required for a Java source or resource. The *`primary_name`* must be enclosed in double quotation marks and its length must not exceed 4000 bytes in the database character set.

- For a Java source, this clause specifies the name of the schema object in which the source code is held. A successful `CREATE JAVA SOURCE` statement will also create additional schema objects to hold each of the Java classes defined by the source.
- For a Java resource, this clause specifies the name of the schema object to hold the Java resource.

Use double quotation marks to preserve a lower- or mixed-case *`primary_name`*.

If you do not specify *`schema`*, then Oracle Database creates the object in your own schema.

Restrictions on NAMED Java Classes

The `NAMED` clause is subject to the following restrictions:

- You cannot specify `NAMED` for a Java class.
- The *`primary_name`* cannot contain a database link.

SCHEMA Clause

The `SCHEMA` clause applies only to a Java class. This optional clause specifies the schema in which the object containing the Java file will reside. If you do not specify this clause, then Oracle Database creates the object in your own schema.

SHARING

This clause applies only when creating a Java schema object in an application root. This type of object is called an application common object and it can be shared with the application PDBs that belong to the application root. To determine how the Java schema object is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the Java schema object's metadata, but its data is unique to each container. This type of Java schema object is referred to as a **metadata-linked application common object**.
- `NONE` - The Java schema object is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the Java schema object. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a Java schema object after it is created.

> **See Also:**
>
> - *Oracle Database Reference* for more information on the `DEFAULT_SHARING` initialization parameter
> - *Oracle Database Administrator's Guide* for complete information on creating application common objects

invoker_rights_clause

Use the *`invoker_rights_clause`* to indicate whether the methods of the class execute with the privileges and in the schema of the user who owns the class or with the privileges and in the schema of `CURRENT_USER`.

This clause also determines how Oracle Database resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER

`CURRENT_USER` indicates that the methods of the class execute with the privileges of `CURRENT_USER`. This clause is the default and creates an **invoker-rights class**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of `CURRENT_USER`. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER

`DEFINER` indicates that the methods of the class execute with the privileges of the owner of the schema in which the class resides, and that external names resolve in the schema where the class resides. This clause creates a **definer-rights class**.

> **See Also:**
>
> - *Oracle Database Java Developer's Guide*
> - *Oracle Database PL/SQL Language Reference* for information on how `CURRENT_USER` is determined

RESOLVER Clause

The `RESOLVER` clause lets you specify a mapping of the fully qualified Java name to a Java schema object, where:

- *`match_string`* is either a fully qualified Java name, a wildcard that can match such a Java name, or a wildcard that can match any name.
- *`schema_name`* designates a schema to be searched for the corresponding Java schema object.

- A dash (-) as an alternative to *schema_name* indicates that if *match_string* matches a valid Java name, Oracle Database can leave the name unresolved. The resolution succeeds, but the name cannot be used at run time by the class.

This mapping is stored with the definition of the schema objects created in this command for use in later resolutions (either implicit or in explicit `ALTER JAVA ... RESOLVE` statements).

USING Clause

The `USING` clause determines a sequence of character data (`CLOB` or `BFILE`) or binary data (`BLOB` or `BFILE`) for the Java class or resource. Oracle Database uses the sequence of characters to define one file for a Java class or resource, or one source file and one or more derived classes for a Java source.

BFILE Clause

Specify the directory and filename of a previously created file on the operating system (*directory_object_name*) and server file (*server_file_name*) containing the sequence. `BFILE` is usually interpreted as a character sequence by `CREATE JAVA SOURCE` and as a binary sequence by `CREATE JAVA CLASS` or `CREATE JAVA RESOURCE`.

CLOB | BLOB | BFILE *subquery*

Specify a subquery that selects a single row and column of the type specified (`CLOB`, `BLOB`, or `BFILE`). The value of the column makes up the sequence of characters.

> **Note:**
>
> In earlier releases, the `USING` clause implicitly supplied the keyword `SELECT`. This is no longer the case. However, the subquery without the keyword `SELECT` is still supported for backward compatibility.

key_for_BLOB

The *key_for_BLOB* clause supplies the following implicit query:

```
SELECT LOB FROM CREATE$JAVA$LOB$TABLE
   WHERE NAME = 'key_for_BLOB';
```

Restriction on the *key_for_BLOB* Clause

For you to use this case, the table `CREATE$JAVA$LOB$TABLE` must exist in the current schema and must have a column LOB of type `BLOB` and a column `NAME` of type `VARCHAR2`.

AS *source_char*

Specify a sequence of characters for a Java source.

Examples

Creating a Java Class Object: Example

The following statement creates a schema object containing a Java class using the name found in a Java binary file:

```
CREATE JAVA CLASS USING BFILE (java_dir, 'Agent.class')
/
```

This example assumes the directory object `java_dir`, which points to the operating system directory containing the Java class `Agent.class`, already exists. In this example, the name of the class determines the name of the Java class schema object.

Creating a Java Source Object: Example

The following statement creates a Java source schema object:

```
CREATE JAVA SOURCE NAMED "Welcome" AS
   public class Welcome {
      public static String welcome() {
         return "Welcome World";   } }
/
```

Creating a Java Resource Object: Example

The following statement creates a Java resource schema object named `apptext` from a `bfile`:

```
CREATE JAVA RESOURCE NAMED "appText"
   USING BFILE (java_dir, 'textBundle.dat')
/
```

14

SQL Statements: CREATE LIBRARY to CREATE SCHEMA

This chapter contains the following SQL statements:

- CREATE LIBRARY
- CREATE LOCKDOWN PROFILE
- CREATE MATERIALIZED VIEW
- CREATE MATERIALIZED VIEW LOG
- CREATE MATERIALIZED ZONEMAP
- CREATE OPERATOR
- CREATE OUTLINE
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PFILE
- CREATE PLUGGABLE DATABASE
- CREATE PROCEDURE
- CREATE PROFILE
- CREATE RESTORE POINT
- CREATE ROLE
- CREATE ROLLBACK SEGMENT
- CREATE SCHEMA

CREATE LIBRARY

Purpose

Use the `CREATE LIBRARY` statement to create a schema object associated with an operating-system shared library. The name of this schema object can then be used in the *`call_spec`* of `CREATE FUNCTION` or `CREATE PROCEDURE` statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can call to third-generation-language (3GL) functions and procedures.

> **See Also:**
>
> CREATE FUNCTION and *Oracle Database PL/SQL Language Reference* for more information on functions and procedures

Prerequisites

The CREATE LIBRARY statement is valid only on platforms that support shared libraries and dynamic linking.

To create a library in your own schema, you must have the CREATE LIBRARY system privilege. To create a library in another user's schema, you must have the CREATE ANY LIBRARY system privilege.

To use the library in the *call_spec* of a CREATE FUNCTION statement, or when declaring a function in a package or type, you must have the EXECUTE object privilege on the library and the CREATE FUNCTION system privilege. Refer to *Oracle Database PL/SQL Language Reference* for information on the *call_spec* of a CREATE FUNCTION statement.

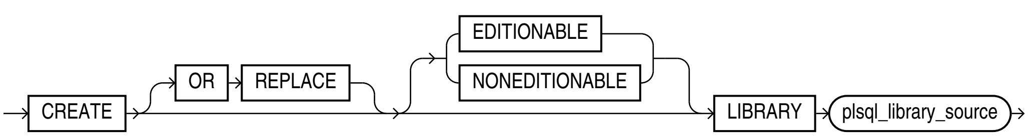
To use the library in the *call_spec* of a CREATE PROCEDURE statement, or when declaring a procedure in a package or type, you must have the EXECUTE object privilege on the library and the CREATE PROCEDURE system privilege. Refer to *Oracle Database PL/SQL Language Reference* for information on the *call_spec* of a CREATE PROCEDURE statement.

To execute a procedure or function defined with the *call_spec* (including a procedure or function defined within a package or type), you must have the EXECUTE object privilege on the procedure or function (but you do not need the EXECUTE object privilege on the library).

Syntax

Libraries are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_library*::=**



(*plsql_library_source*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify OR REPLACE to re-create the library if it already exists. Use this clause to change the definition of an existing library without dropping, re-creating, and regranting object privileges granted on it.

Users who had previously been granted privileges on a redefined library can still access the library without being regranted the privileges.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the library is an editioned or noneditioned object if editioning is enabled for the schema object type `LIBRARY` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_library_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_library_source`*.

CREATE LOCKDOWN PROFILE

Purpose

Use the `CREATE LOCKDOWN PROFILE` statement to create a PDB lockdown profile. You can use PDB lockdown profiles in a multitenant container database (CDB) to restrict user operations in PDBs.

After you create a PDB lockdown profile, you can add restrictions to the profile with the `ALTER LOCKDOWN PROFILE` statement. You can restrict user operations associated with certain database features, options, and SQL statements.

When a lockdown profile is assigned to a PDB, users in that PDB cannot perform the operations that are the disabled for the profile. To assign a lockdown profile, set its name for the value of the `PDB_LOCKDOWN` initialization parameter. You can assign a lockdown profile to individual PDBs, or to all PDBs in a CDB or application container, as follows:

- If you set `PDB_LOCKDOWN` while connected to a CDB root, then the lockdown profile applies to all PDBs in the CDB. It does not apply to the CDB root.
- If you set `PDB_LOCKDOWN` while connected to an application root, then the lockdown profile applies to the application root and all PDBs in the application container.
- If you set `PDB_LOCKDOWN` while connected to a particular PDB, then the lockdown profile applies to that PDB and overrides the lockdown profile for the CDB or application container, if one exists.

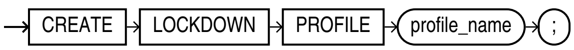
> **See Also:**
>
> - ALTER LOCKDOWN PROFILE and DROP LOCKDOWN PROFILE
> - *Oracle Database Security Guide* for more information on PDB lockdown profiles

Prerequisites

The `CREATE LOCKDOWN PROFILE` statement is valid only in a CDB. The current container must be the CDB root and you must have the `CREATE LOCKDOWN PROFILE` system privilege, either granted commonly or granted locally in the CDB root.

Syntax

***create_lockdown_profile*::=**



Semantics

profile_name

Specify the name of the PDB lockdown profile to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". The *`profile_name`* must be unique across the entire CDB.

Example

The following statement creates PDB lockdown profile `hr_prof`:

```
CREATE LOCKDOWN PROFILE hr_prof;
```

CREATE MATERIALIZED VIEW

Purpose

Use the `CREATE MATERIALIZED VIEW` statement to create a **materialized view**. A materialized view is a database object that contains the results of a query. The `FROM` clause of the query can name tables, views, and other materialized views. Collectively these objects are called **master tables** (a replication term) or **detail tables** (a data warehousing term). This reference uses "master tables" for consistency. The databases containing the master tables are called the **master databases**.

> **Note:**
>
> The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

For replication purposes, materialized views allow you to maintain read-only copies of remote data on your local node. You can select data from a materialized view as you would from a table or view. In replication environments, the materialized views commonly created are **primary key**, **rowid**, **object**, and **subquery** materialized views.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on the types of materialized views used to support replication

For data warehousing purposes, the materialized views commonly created are **materialized aggregate views**, **single-table materialized aggregate views**, and **materialized join views**. All three types of materialized views can be used by query rewrite, an optimization technique that transforms a user request written in terms of master tables into a semantically equivalent request that includes one or more materialized views.

> **See Also:**
>
> - ALTER MATERIALIZED VIEW
> - *Oracle Database Data Warehousing Guide* for information on the types of materialized views used to support data warehousing

Prerequisites

The privileges required to create a materialized view should be granted directly rather than through a role.

To create a materialized view **in your own schema:**

- You must have been granted the `CREATE MATERIALIZED VIEW` system privilege **and** either the `CREATE TABLE` or `CREATE ANY TABLE` system privilege.
- You must also have access to any master tables of the materialized view that you do not own, either through a `READ` or `SELECT` object privilege on each of the tables or through the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

To create a materialized view **in another user's schema:**

- You must have the `CREATE ANY MATERIALIZED VIEW` system privilege.
- The owner of the materialized view must have the `CREATE TABLE` system privilege. The owner must also have access to any master tables of the materialized view that the schema owner does not own (for example, if the master tables are on a remote database) and to any materialized view logs defined on those master tables, either through a `READ` or `SELECT` object privilege on each of the tables or through the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

To create a refresh-on-commit materialized view (`REFRESH ON COMMIT` clause), in addition to the preceding privileges, you must have the `ON COMMIT REFRESH` object privilege on any master tables that you do not own or you must have the `ON COMMIT REFRESH` system privilege.

To create the materialized view **with query rewrite enabled**, in addition to the preceding privileges:

- If the schema owner does not own the master tables, then the schema owner must have the `GLOBAL QUERY REWRITE` privilege or the `QUERY REWRITE` object privilege on each table outside the schema.
- If you are defining the materialized view on a prebuilt container (`ON PREBUILT TABLE` clause), then you must have the `READ` or `SELECT` privilege `WITH GRANT OPTION` on the container table.

The user whose schema contains the materialized view must have sufficient quota in the target tablespace to store the master table and index of the materialized view or must have the `UNLIMITED TABLESPACE` system privilege.

When you create a materialized view, Oracle Database creates one internal table and at least one index, and may create one view, all in the schema of the materialized view. Oracle Database uses these objects to maintain the materialized view data. You must have the privileges necessary to create these objects.

You can create the following types of local materialized views (including both `ON COMMIT` and `ON DEMAND`) on master tables with commit SCN-based materialized view logs:

- Materialized aggregate views, including materialized aggregate views on a single table
- Materialized join views
- Primary-key-based and rowid-based single table materialized views
- `UNION ALL` materialized views, where each `UNION ALL` branch is one of the above materialized view types

You cannot create remote materialized views on master tables with commit SCN-based materialized view logs.

Creating a materialized view on master tables with different types of materialized view logs (that is, a master table with timestamp-based materialized view logs and a master table with commit SCN-based materialized view logs) is not supported and causes `ORA-32414`.

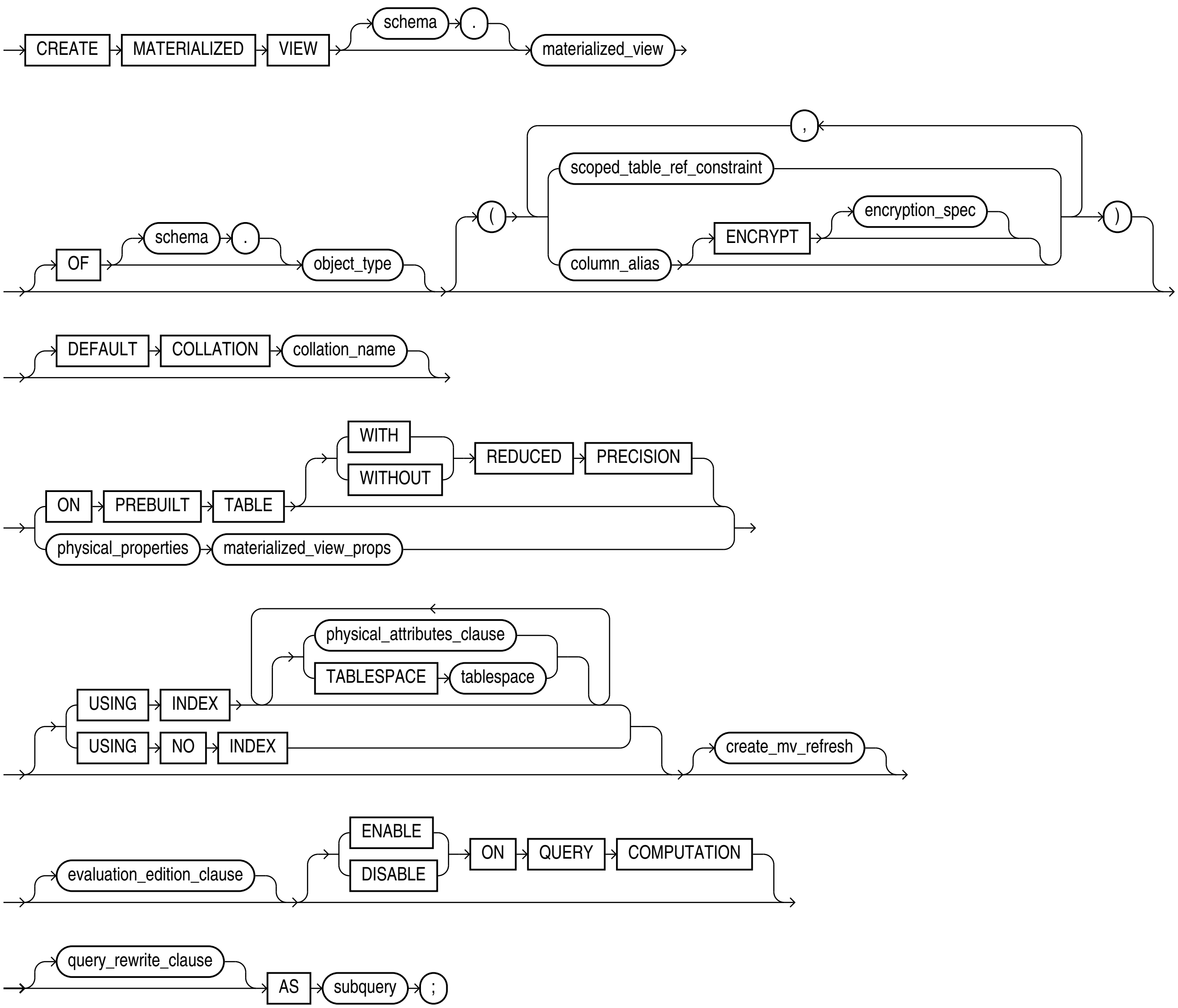
To specify an edition in the *`evaluation_edition_clause`* or the *`unusable_editions_clause`*, you must have the `USE` privilege on the edition.

See Also:

- CREATE TABLE, CREATE VIEW , and CREATE INDEX for information on these privileges
- *Oracle Database Administrator's Guide* for information about the prerequisites that apply to creating replication materialized views
- *Oracle Database Data Warehousing Guide* for information about the prerequisites that apply to creating data warehousing materialized views

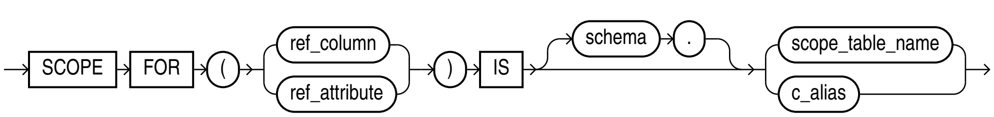
Syntax

***create_materialized_view*::=**

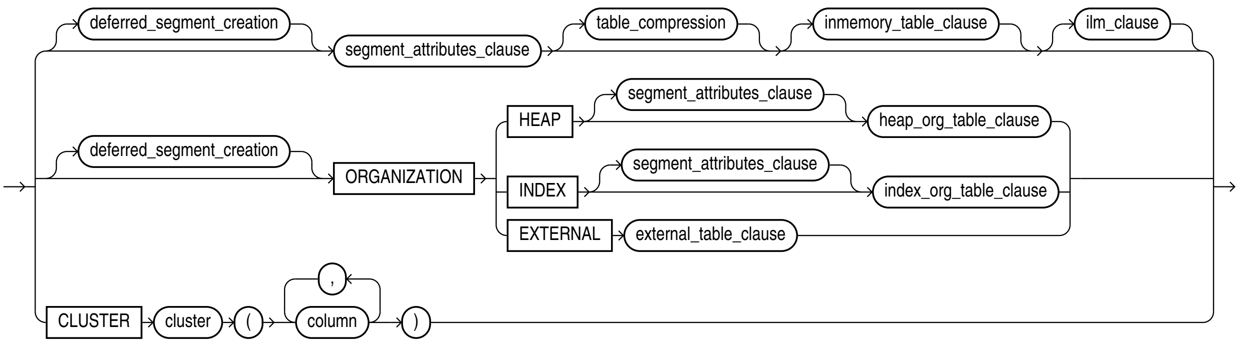


(*scoped_table_ref_constraint*::=, *physical_properties*::=, *materialized_view_props*::=, *physical_attributes_clause*::=, *create_mv_refresh*::=, *evaluation_edition_clause*::=, *query_rewrite_clause*::=, *subquery*::=)

***scoped_table_ref_constraint*::=**

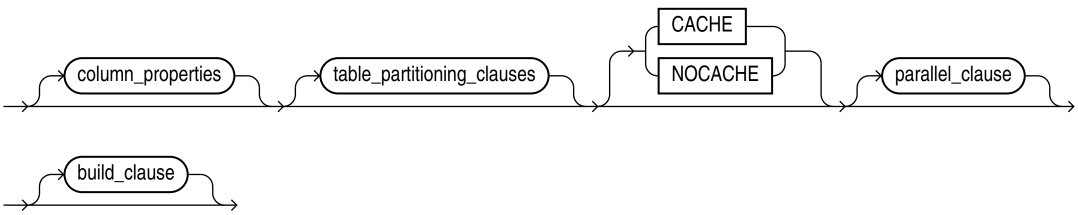


***physical_properties*::=**



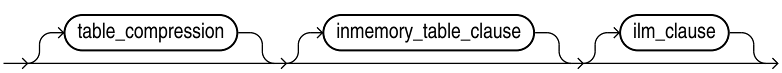
(*deferred_segment_creation*::=, *segment_attributes_clause*::=, *table_compression*::=, *inmemory_table_clause*::=, *heap_org_table_clause*::=, *index_org_table_clause*::=)

***materialized_view_props*::=**

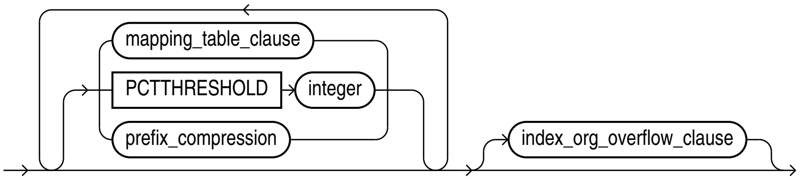


(*column_properties*::=, *table_partitioning_clauses*::=—part of `CREATE TABLE` syntax, *parallel_clause*::=, *build_clause*::=)

***heap_org_table_clause*::=**

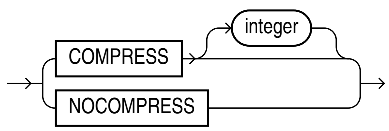


***index_org_table_clause*::=**

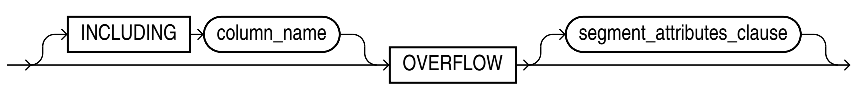


(`mapping_table_clause`: not supported with materialized views, *prefix_compression*::=, *index_org_overflow_clause*::=)

***prefix_compression*::=**

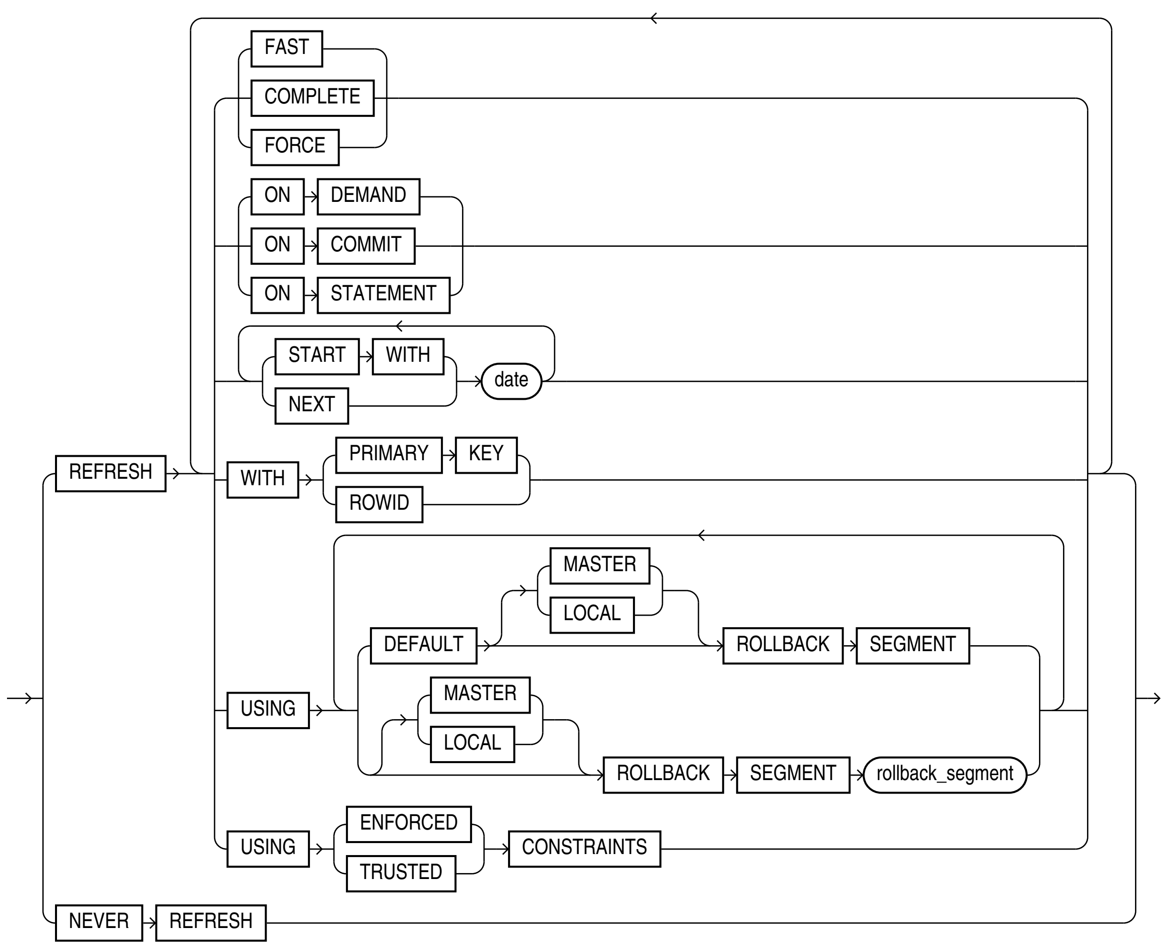


***index_org_overflow_clause*::=**

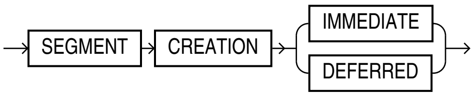


(*segment_attributes_clause*::=)

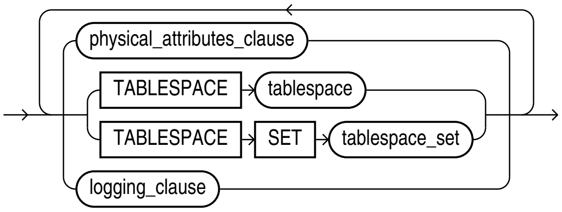
***create_mv_refresh*::=**



***deferred_segment_creation*::=**

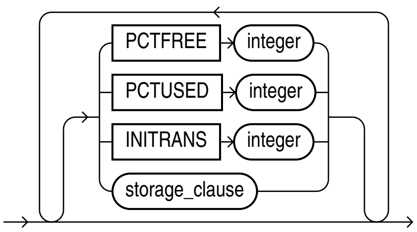


***segment_attributes_clause*::=**



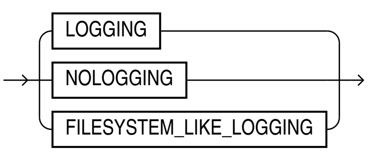
(*physical_attributes_clause::=*, `TABLESPACE SET`: not supported with `CREATE MATERIALIZED VIEW`, *logging_clause::=*)

***physical_attributes_clause*::=**

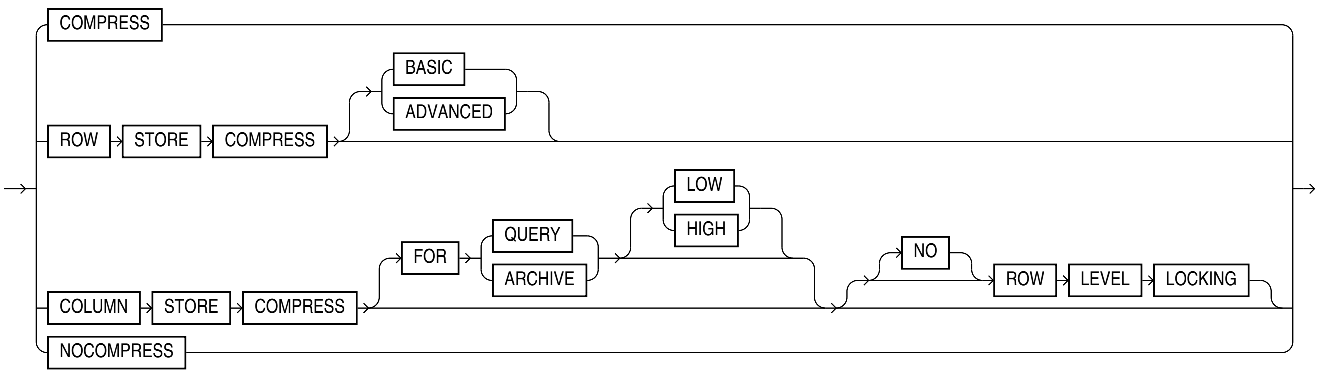


(*logging_clause::=*)

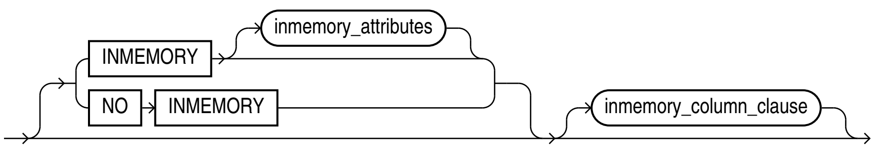
***logging_clause*::=**



***table_compression*::=**

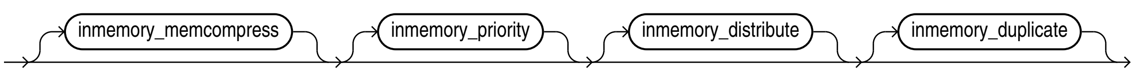


***inmemory_table_clause*::=**



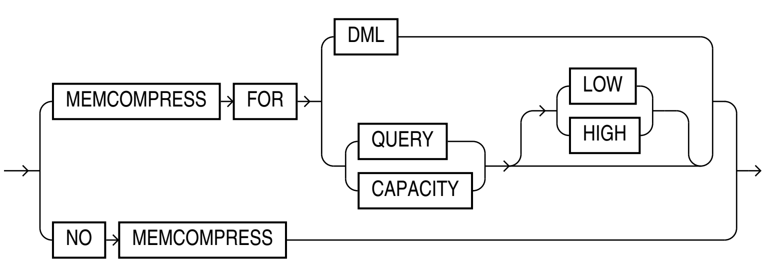
(*inmemory_attributes*::=, *inmemory_column_clause*::=)

***inmemory_attributes*::=**

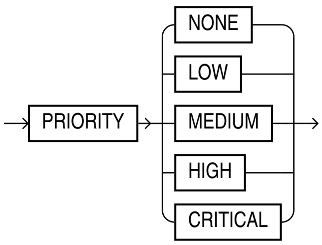


(*inmemory_memcompress*::=, *inmemory_priority*::=, *inmemory_distribute*::=, *inmemory_duplicate*::=)

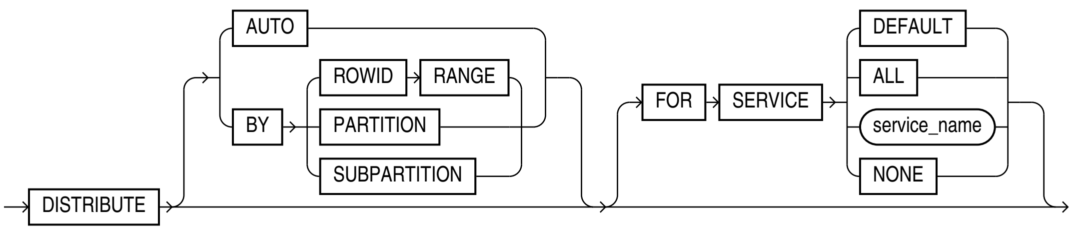
***inmemory_memcompress*::=**



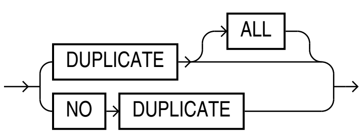
***inmemory_priority*::=**



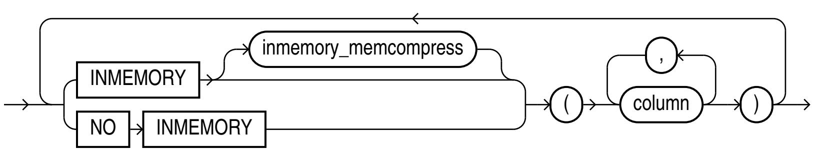
***inmemory_distribute*::=**



***inmemory_duplicate*::=**

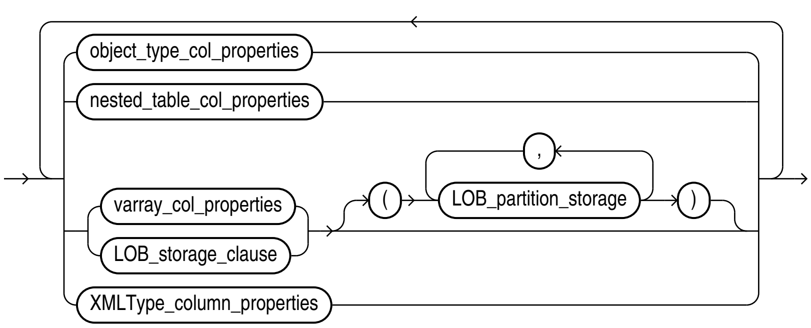


***inmemory_column_clause*::=**



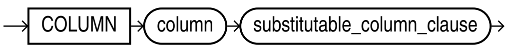
(*inmemory_memcompress::=*)

***column_properties*::=**



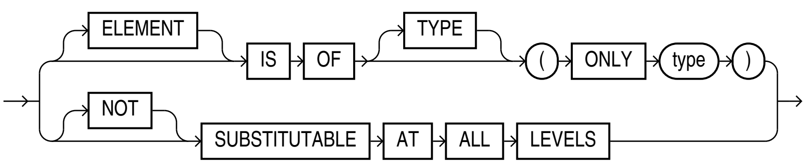
(*object_type_col_properties*::=, *nested_table_col_properties*::=, *varray_col_properties*::=, *LOB_partition_storage*::=, *LOB_storage_clause*::=, `XMLType_column_properties`: not supported for materialized views)

***object_type_col_properties*::=**

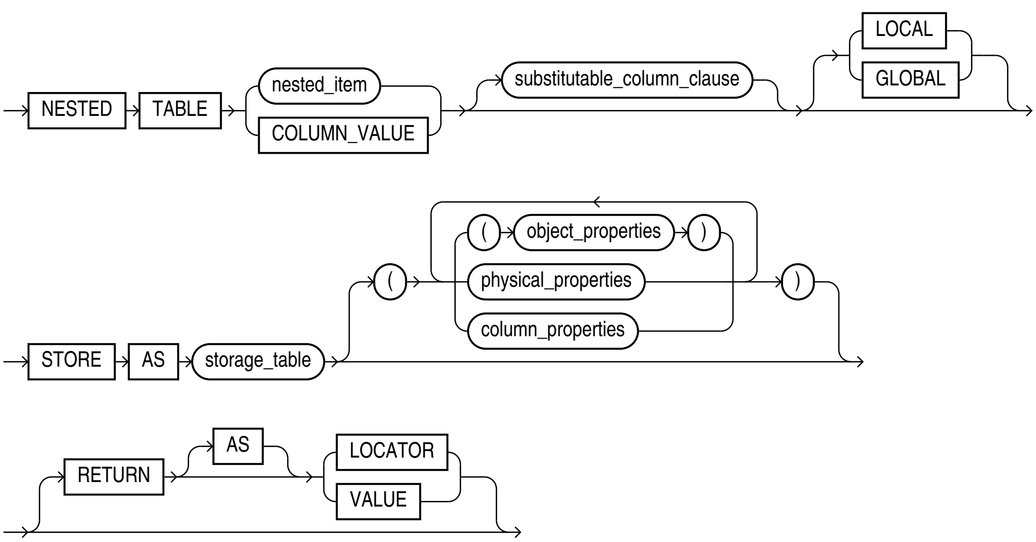


(*substitutable_column_clause*::=)

***substitutable_column_clause*::=**

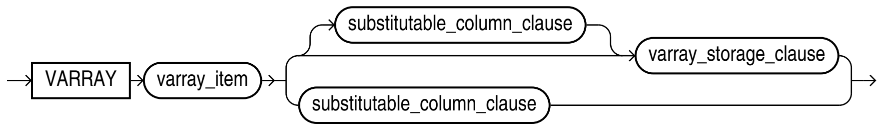


***nested_table_col_properties*::=**



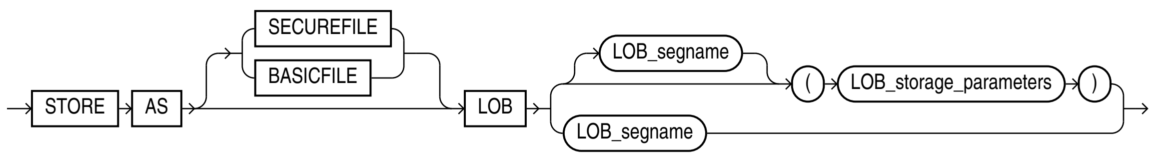
(*substitutable_column_clause*::=, *object_properties*::=, *physical_properties*::=—part of `CREATE TABLE` syntax, *column_properties*::=)

***varray_col_properties*::=**



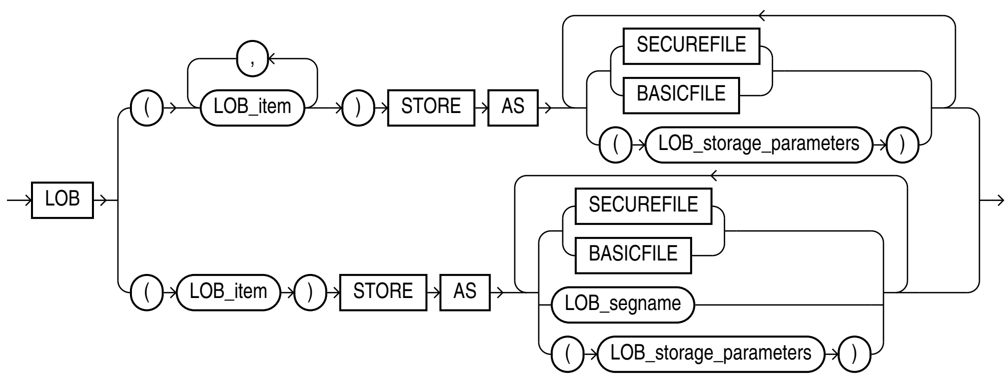
(*substitutable_column_clause::=*, *varray_storage_clause::=*)

***varray_storage_clause*::=**



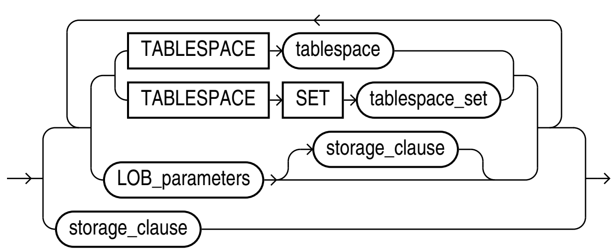
(*LOB_parameters::=*)

***LOB_storage_clause*::=**



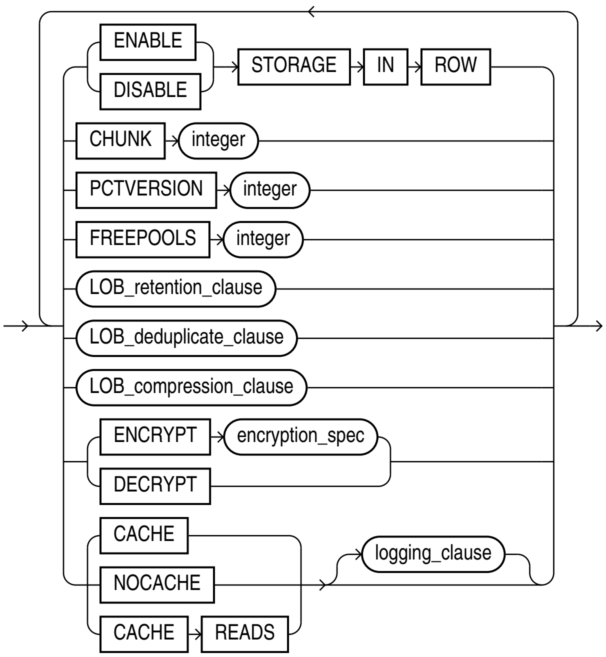
(*LOB_storage_parameters::=*)

***LOB_storage_parameters*::=**



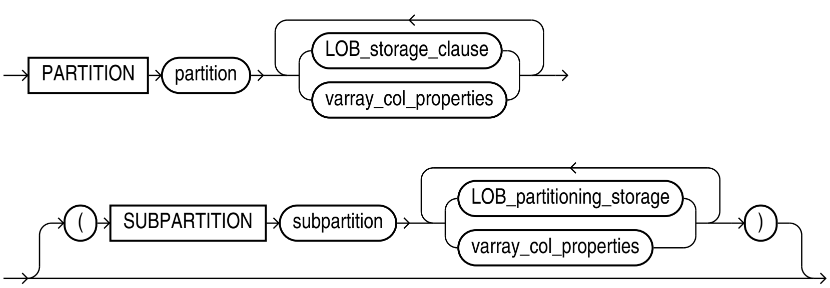
(TABLESPACE SET: not supported with CREATE MATERIALIZED VIEW, *LOB_parameters::=*, *storage_clause::=*)

***LOB_parameters*::=**



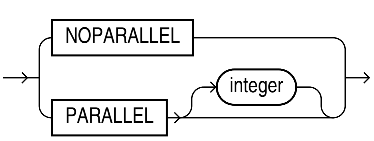
(*storage_clause::=*, *logging_clause::=*)

***LOB_partition_storage*::=**

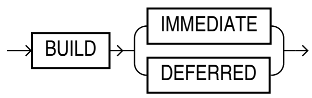


(*LOB_storage_clause::=*, *varray_col_properties::=*)

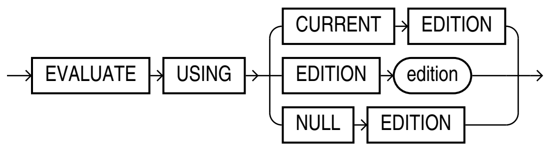
***parallel_clause*::=**



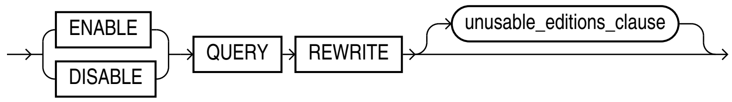
***build_clause*::=**



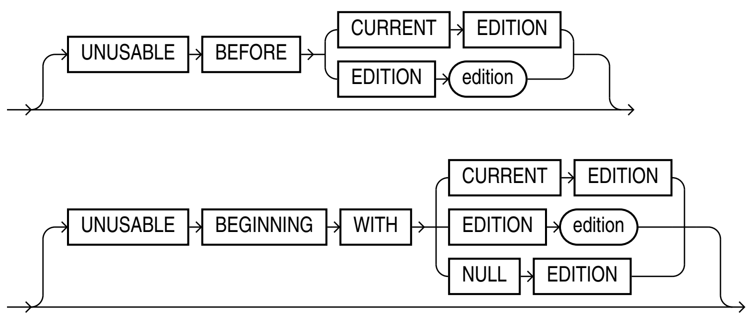
***evaluation_edition_clause*::=**



***query_rewrite_clause*::=**



***unusable_editions_clause*::=**



Semantics

schema

Specify the schema to contain the materialized view. If you omit `schema`, then Oracle Database creates the materialized view in your schema.

materialized_view

Specify the name of the materialized view to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". Oracle Database generates names for the table and indexes used to maintain the materialized view by adding a prefix or suffix to the materialized view name.

column_alias

You can specify a column alias for each column of the materialized view. The column alias list explicitly resolves any column name conflict, eliminating the need to specify aliases in the `SELECT` clause of the materialized view. If you specify any column alias in this clause, then you must specify an alias for each data source referenced in the `SELECT` clause.

ENCRYPT clause

Use this clause to encrypt this column of the materialized view. Refer to the `CREATE TABLE` clause *encryption_spec* for more information on column encryption.

OF *object_type*

The `OF` *`object_type`* clause lets you explicitly create an **object materialized view** of type *`object_type`*.

> **See Also:**
>
> See `CREATE TABLE` ... *object_table* for more information on the `OF` *`type_name`* clause

scoped_table_ref_constraint

Use the `SCOPE FOR` clause to restrict the scope of references to a single object table. You can refer either to the table name with *`scope_table_name`* or to a column alias. The values in the `REF` column or attribute point to objects in *`scope_table_name`* or *`c_alias`*, in which object instances of the same type as the `REF` column are stored. If you specify aliases, then they must have a one-to-one correspondence with the columns in the `SELECT` list of the defining query of the materialized view.

> **See Also:**
>
> "SCOPE REF Constraints" for more information

DEFAULT COLLATION

Use this clause to specify the default collation for the materialized view. The default collation is used as the derived collation for all the character literals included in the defining query of the materialized view. The default collation is not used by the materialized view columns; the collations for the materialized view columns are derived from the materialized view's defining subquery. The `CREATE MATERIALIZED VIEW` statement fails with an error, or the materialized view is created in an invalid state, if any of its character columns is based on an expression in the defining subquery that has no derived collation.

For *`collation_name`*, specify a valid named collation or pseudo-collation.

If you omit this clause, then the default collation for the materialized view is set to the effective schema default collation of the schema containing the materialized view. Refer to the DEFAULT_COLLATION clause of `ALTER SESSION` for more information on the effective schema default collation.

You can specify the DEFAULT COLLATION clause only if the COMPATIBLE initialization parameter is set to 12.2 or greater, and the MAX_STRING_SIZE initialization parameter is set to EXTENDED.

To change the default collation for a materialized view, you must recreate the materialized view.

Restrictions on the Default Collation for Materialized Views

The following restrictions apply when specifying the default collation for a materialized view:

- If the defining query of the materialized view contains the WITH *plsql_declarations* clause, then the default collation of the materialized view must be USING_NLS_COMP.
- If the materialized view is created on a prebuilt table, then the declared collations of the table columns must be the same as the corresponding collations of the materialized view columns, as derived from the defining query.

ON PREBUILT TABLE Clause

The ON PREBUILT TABLE clause lets you register an existing table as a preinitialized materialized view. This clause is particularly useful for registering large materialized views in a data warehousing environment. The table must have the same name and be in the same schema as the resulting materialized view.

If the materialized view is dropped, then the preexisting table reverts to its identity as a table.

> **Note:**
>
> This clause assumes that the table object reflects the materialization of a subquery. Oracle strongly recommends that you ensure that this assumption is true in order to ensure that the materialized view correctly reflects the data in its master tables.

The ON PREBUILT TABLE clause could be useful in the following scenarios:

- You have a table representing the result of a query. Creating the table was an expensive operation that possibly took a long time. You want to create a materialized view on the query. You can use the ON PREBUILT TABLE clause to avoid the expense of executing the query and populating the container for the materialized view.
- You temporarily discontinue having a materialized view, but keep its container table, using the DROP MATERIALIZED VIEW ... PRESERVE TABLE statement. You then decide to recreate the materialized view and you know that the master tables of the view have not changed. You can create the materialized view using the ON PREBUILT TABLE clause. This avoids the expense and time of creating and populating the container table for the materialized view.

If you specify ON PREBUILT TABLE, then Oracle database does not create the I_SNAP$ index. This index improves fast refresh performance. If you want the benefits of this index, then you can create it manually. Refer to *Oracle Database Data Warehousing Guide* for more information.

WITH REDUCED PRECISION

Specify `WITH REDUCED PRECISION` to authorize the loss of precision that will result if the precision of the table or materialized view columns do not exactly match the precision returned by *subquery*.

WITHOUT REDUCED PRECISION

Specify `WITHOUT REDUCED PRECISION` to require that the precision of the table or materialized view columns match exactly the precision returned by *subquery*, or the create operation will fail. This is the default.

Restrictions on Using Prebuilt Tables

Prebuilt tables are subject to the following restrictions:

- Each column alias in *subquery* must correspond to a column in the prebuilt table, and corresponding columns must have matching data types.
- If you specify this clause, then you cannot specify a `NOT NULL` constraint for any column that is not referenced in *subquery* unless you also specify a default value for that column.
- You cannot specify the `ON PREBUILT TABLE` clause when creating a rowid materialized view.

> **See Also:**
>
> "Creating Prebuilt Materialized Views: Example"

physical_properties_clause

The components of the *physical_properties_clause* have the same semantics for materialized views that they have for tables, with exceptions and additions described in the sections that follow.

Restriction on the *physical_properties_clause*

You cannot specify `ORGANIZATION EXTERNAL` for a materialized view.

deferred_segment_creation

Use this clause to determine when the segment for this materialized view should be created. See the `CREATE TABLE` clause *deferred_segment_creation* for more information.

segment_attributes_clause

Use the *segment_attributes_clause* to establish values for the `PCTFREE`, `PCTUSED`, and `INITRANS` parameters, the storage characteristics for the materialized view, to assign a tablespace, and to specify whether logging is to occur. In the `USING INDEX` clause, you cannot specify `PCTFREE` or `PCTUSED`.

TABLESPACE Clause

Specify the tablespace in which the materialized view is to be created. If you omit this clause, then Oracle Database creates the materialized view in the default tablespace of the schema containing the materialized view.

> **See Also:**
>
> *physical_attributes_clause* and storage_clause for a complete description of these clauses, including default values

logging_clause

Specify `LOGGING` or `NOLOGGING` to establish the logging characteristics for the materialized view. The logging characteristic affects the creation of the materialized view and any nonatomic refresh that is initiated by way of the `DBMS_REFRESH` package. The default is the logging characteristic of the tablespace in which the materialized view resides.

> **See Also:**
>
> *logging_clause* for a full description of this clause and *Oracle Database PL/SQL Packages and Types Reference* for more information on atomic and nonatomic refresh

table_compression

Use the *`table_compression`* clause to instruct the database whether to compress data segments to reduce disk and memory use. This clause has the same semantics in `CREATE MATERIALIZED VIEW` and `CREATE TABLE`. Refer to the *table_compression* clause in the documentation on `CREATE TABLE` for the full semantics of this clause.

inmemory_table_clause

Use the *`inmemory_table_clause`* to enable or disable the materialized view for the In-Memory Column Store (IM column store). This clause has the same semantics as the *inmemory_table_clause* in the `CREATE TABLE` documentation.

inmemory_column_clause

Use the *`inmemory_column_clause`* to disable specific materialized view columns for the IM column store, and to specify the data compression method for specific columns. This clause has the same semantics here as it has for the *inmemory_column_clause* in the `CREATE TABLE` documentation, with the following addition: If you specify the *`inmemory_column_clause`*, then you must also specify a *`column_alias`* for each column in the materialized view.

index_org_table_clause

The `ORGANIZATION INDEX` clause lets you create an index-organized materialized view. In such a materialized view, data rows are stored in an index defined on the primary key of the materialized view. You can specify index organization for the following types of materialized views:

- Read-only and updatable object materialized views. You must ensure that the master table has a primary key.
- Read-only and updatable primary key materialized views.
- Read-only rowid materialized views.

The keywords and parameters of the *index_org_table_clause* have the same semantics as described in CREATE TABLE, with the restrictions that follow.

> **See Also:**
>
> The *index_org_table_clause* of CREATE TABLE

Restrictions on Index-Organized Materialized Views

Index-organized materialized views are subject to the following restrictions:

- You cannot specify the following CREATE MATERIALIZED VIEW clauses: CACHE or NOCACHE, CLUSTER, or ON PREBUILT TABLE.
- In the *index_org_table_clause*:
  - You cannot specify the *mapping_table_clause*.
  - You can specify COMPRESS only for a materialized view based on a composite primary key. You can specify NOCOMPRESS for a materialized view based on either a simple or composite primary key.

CLUSTER Clause

The CLUSTER clause lets you create the materialized view as part of the specified cluster. A cluster materialized view uses the space allocation of the cluster. Therefore, you do not specify physical attributes or the TABLESPACE clause with the CLUSTER clause.

Restriction on Cluster Materialized Views

If you specify CLUSTER, then you cannot specify the *table_partitioning_clauses* in *materialized_view_props*.

materialized_view_props

Use these property clauses to describe a materialized view that is not based on an existing table. To create a materialized view that is based on an existing table, use the ON PREBUILT TABLE clause.

column_properties

The *column_properties* clause lets you specify the storage characteristics of a LOB, nested table, varray, or XMLType column. The *object_type_col_properties* are not relevant for a materialized view.

> **See Also:**
>
> CREATE TABLE for detailed information about specifying the parameters of this clause

table_partitioning_clauses

The *table_partitioning_clauses* let you specify that the materialized view is partitioned on specified ranges of values or on a hash function. Partitioning of materialized views is the same as partitioning of tables.

> **See Also:**
>
> *table_partitioning_clauses* in the CREATE TABLE documentation

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this table are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

> **Note:**
>
> NOCACHE has no effect on materialized views for which you specify KEEP in the *storage_clause*.

> **See Also:**
>
> CREATE TABLE for information about specifying CACHE or NOCACHE

parallel_clause

The *parallel_clause* lets you indicate whether parallel operations will be supported for the materialized view and sets the default degree of parallelism for queries and DML on the materialized view after creation.

For complete information on this clause, refer to *parallel_clause* in the documentation on CREATE TABLE.

build_clause

The *build_clause* lets you specify when to populate the materialized view.

IMMEDIATE

Specify IMMEDIATE to indicate that the materialized view is to be populated immediately. This is the default.

DEFERRED

Specify DEFERRED to indicate that the materialized view is to be populated by the next REFRESH operation. The first (deferred) refresh must always be a complete refresh. Until then, the materialized view has a staleness value of UNUSABLE, so it cannot be used for query rewrite.

USING INDEX Clause

The USING INDEX clause lets you establish the value of the INITRANS and STORAGE parameters for the default index Oracle Database uses to maintain the materialized view data. If USING INDEX is not specified, then default values are used for the index. Oracle Database uses the default index to speed up incremental (FAST) refresh of the materialized view.

Restriction on USING INDEX clause

You cannot specify the PCTUSED parameter in this clause.

USING NO INDEX Clause

Specify USING NO INDEX to suppress the creation of the default index. You can create an alternative index explicitly by using the CREATE INDEX statement. You should create such an index if you specify USING NO INDEX and you are creating the materialized view with the fast refresh method (REFRESH FAST).

create_mv_refresh

Use the *create_mv_refresh* clause to specify the default methods, modes, and times for the database to refresh the materialized view. If the master tables of a materialized view are modified, then the data in the materialized view must be updated to make the materialized view accurately reflect the data currently in its master tables. This clause lets you schedule the times and specify the method and mode for the database to refresh the materialized view.

Restriction on Synchronous Refresh

If you are using the synchronous refresh method, then you must specify ON DEMAND and USING TRUSTED CONSTRAINTS.

> **Note:**
>
> This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle Database Administrator's Guide* and *Oracle Database Data Warehousing Guide*.

> **See Also:**
>
> - "Periodic Refresh of Materialized Views: Example" and "Automatic Refresh Times for Materialized Views: Example"
> - *Oracle Database PL/SQL Packages and Types Reference* for more information on refresh methods
> - *Oracle Database Data Warehousing Guide* to learn how to use refresh statistics to monitor the performance of materialized view refresh operations

FAST Clause

Specify `FAST` to indicate the fast refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes for conventional DML changes are stored in the materialized view log associated with the master table. The changes for direct-path `INSERT` operations are stored in the direct loader log.

If you specify `REFRESH FAST`, then the `CREATE` statement will fail unless materialized view logs already exist for the materialized view master tables. Oracle Database creates the direct loader log automatically when a direct-path `INSERT` takes place. No user intervention is needed.

For both conventional DML changes and for direct-path `INSERT` operations, other conditions may restrict the eligibility of a materialized view for fast refresh.

Restrictions on FAST Refresh

`FAST` refresh is subject to the following restrictions:

- When you specify `FAST` refresh at create time, Oracle Database verifies that the materialized view you are creating is eligible for fast refresh. When you change the refresh method to `FAST` in an `ALTER MATERIALIZED VIEW` statement, Oracle Database does not perform this verification. If the materialized view is not eligible for fast refresh, then Oracle Database returns an error when you attempt to refresh this view.
- Materialized views are not eligible for fast refresh if the defining query contains an analytic function or the `XMLTable` function.
- Materialized views are not eligible for fast refresh if the defining query references a table on which an `XMLIndex` index is defined.
- You cannot fast refresh a materialized view if any of its columns is encrypted.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for restrictions on fast refresh in replication environments
> - *Oracle Database Data Warehousing Guide* for restrictions on fast refresh in data warehousing environments
> - The `EXPLAIN_MVIEW` procedure of the `DBMS_MVIEW` package for help diagnosing problems with fast refresh and the `TUNE_MVIEW` procedure of the `DBMS_MVIEW` package for correction of query rewrite problems
> - "Analytic Functions "
> - "Creating a Fast Refreshable Materialized View: Example"

COMPLETE Clause

Specify `COMPLETE` to indicate the complete refresh method, which is implemented by executing the defining query of the materialized view. If you request a complete refresh, then Oracle Database performs a complete refresh even if a fast refresh is possible.

FORCE Clause

Specify `FORCE` to indicate that when a refresh occurs, Oracle Database will perform a fast refresh if one is possible or a complete refresh if fast refresh is not possible. If you do not specify a refresh method (`FAST`, `COMPLETE`, or `FORCE`), then `FORCE` is the default.

ON COMMIT Clause

Specify `ON COMMIT` to indicate that a refresh is to occur whenever the database commits a transaction that operates on a master table of the materialized view. This clause may increase the time taken to complete the commit, because the database performs the refresh operation as part of the commit process.

You can specify only one of the `ON COMMIT`, `ON DEMAND`, and `ON STATEMENT` clauses. If you specify `ON COMMIT`, then you cannot also specify `START WITH` or `NEXT`.

Restrictions on Refreshing ON COMMIT

The following restrictions apply to the `ON COMMIT` clause:

- This clause is not supported for materialized views containing object types or Oracle-supplied types.
- This clause is not supported for materialized views with remote tables.
- If you specify this clause, then you cannot subsequently execute a distributed transaction on any master table of this materialized view. For example, you cannot insert into the master by selecting from a remote table. The `ON DEMAND` clause does not impose this restriction on subsequent distributed transactions on master tables.

ON DEMAND Clause

Specify `ON DEMAND` to indicate that database will not refresh the materialized view unless the user manually launches a refresh through one of the three `DBMS_MVIEW` refresh procedures.

You can specify only one of the ON COMMIT, ON DEMAND, and ON STATEMENT clauses. If you omit all three of these clauses, then ON DEMAND is the default. You can override this default setting by specifying the START WITH or NEXT clauses, either in the same CREATE MATERIALIZED VIEW statement or a subsequent ALTER MATERIALIZED VIEW statement.

START WITH and NEXT take precedence over ON DEMAND. Therefore, in most circumstances it is not meaningful to specify ON DEMAND when you have specified START WITH or NEXT.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for information on these procedures
> - *Oracle Database Data Warehousing Guide* on the types of materialized views you can create by specifying REFRESH ON DEMAND

ON STATEMENT Clause

Specify ON STATEMENT to indicate that an automatic refresh is to occur every time a DML operation is performed on any of the materialized view's base tables.

You can specify only one of the ON COMMIT, ON DEMAND, and ON STATEMENT clauses. You can specify ON STATEMENT only when *creating* a materialized view. You cannot subsequently *alter* the materialized view to use ON STATEMENT refresh.

Restrictions on Refreshing ON STATEMENT

The following restrictions apply to the ON STATEMENT clause:

- This clause can be used only with materialized views that are fast refreshable. The ON STATEMENT clause must be specified with the REFRESH FAST clause.
- The base tables referenced in the materialized view's defining query must be connected in a join graph that uses the star schema or snowflake schema model. The query must contain exactly one centralized fact table and one or more dimension tables, with all pairs of joined tables being related using primary key-foreign key constraints.
  - There is no restriction on the depth of the snowflake model.
  - The constraints can be in RELY mode. However, you must include the USING TRUSTED CONSTRAINT clause while creating the materialized view to use the RELY constraint.
- The materialized view's defining query must include the ROWID column of the fact table.
- The materialized view's defining query cannot include any of the following: invisible columns, ANSI join syntax, complex query, inline view as base table, composite primary key, LONG columns, and LOB columns.
- You cannot alter the definition of an existing materialized view that uses the ON STATEMENT refresh mode.
- You cannot alter an existing materialized view and enable ON STATEMENT refresh for it.

- The following operations cause a materialized view with `ON STATEMENT` refresh to become unusable:
  - `UPDATE` operations on one or more dimension tables on which the materialized view is based
  - Partition maintenance operations and `TRUNCATE` operations on any base table

    However, a materialized view with the `ON STATEMENT` refresh mode can be partitioned.
- All the restrictions that apply to the `ON COMMIT` clause apply to `ON STATEMENT`.

START WITH Clause

Specify a datetime expression for the first automatic refresh time.

NEXT Clause

Specify a datetime expression for calculating the interval between automatic refreshes.

Both the `START WITH` and `NEXT` values must evaluate to a time in the future. If you omit the `START WITH` value, then the database determines the first automatic refresh time by evaluating the `NEXT` expression with respect to the creation time of the materialized view. If you specify a `START WITH` value but omit the `NEXT` value, then the database refreshes the materialized view only once. If you omit both the `START WITH` and `NEXT` values, or if you omit the *`create_mv_refresh`* entirely, then the database does not automatically refresh the materialized view.

WITH PRIMARY KEY Clause

Specify `WITH PRIMARY KEY` to create a primary key materialized view. This is the default and should be used in all cases except those described for `WITH ROWID`. Primary key materialized views allow materialized view master tables to be reorganized without affecting the eligibility of the materialized view for fast refresh. The master table must contain an enabled primary key constraint, and the defining query of the materialized view must specify all of the primary key columns directly. In the defining query, the primary key columns cannot be specified as the argument to a function such as `UPPER`.

Restriction on Primary Key Materialized Views

You cannot specify this clause for an object materialized view. Oracle Database implicitly refreshes objects materialized `WITH OBJECT ID`.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for detailed information about primary key materialized views and "Creating Primary Key Materialized Views: Example"

WITH ROWID Clause

Specify `WITH ROWID` to create a rowid materialized view. Rowid materialized views are useful if the materialized view does not include all primary key columns of the master tables. Rowid materialized views must be based on a single table and cannot contain any of the following:

- Distinct or aggregate functions
- `GROUP BY or CONNECT BY` clauses

- Subqueries
- Joins
- Set operations

The WITH ROWID clause has no effect if there are multiple master tables in the defining query.

Rowid materialized views are not eligible for fast refresh after a master table reorganization until a complete refresh has been performed.

Restriction on Rowid Materialized Views

You cannot specify this clause for an object materialized view. Oracle Database implicitly refreshes objects materialized WITH OBJECT ID.

> **See Also:**
>
> "Creating Materialized Aggregate Views: Example" and "Creating Rowid Materialized Views: Example"

USING ROLLBACK SEGMENT Clause

This clause is not valid if your database is in automatic undo mode, because in that mode Oracle Database uses undo tablespaces instead of rollback segments. Oracle strongly recommends that you use automatic undo mode. This clause is supported for backward compatibility with replication environments containing older versions of Oracle Database that still use rollback segments.

For *rollback_segment*, specify the remote rollback segment to be used during materialized view refresh.

DEFAULT

DEFAULT specifies that Oracle Database will choose automatically which rollback segment to use. If you specify DEFAULT, then you cannot specify *rollback_segment*. DEFAULT is most useful when modifying, rather than creating, a materialized view.

> **See Also:**
>
> ALTER MATERIALIZED VIEW

MASTER

MASTER specifies the remote rollback segment to be used at the remote master site for the individual materialized view.

LOCAL

LOCAL specifies the remote rollback segment to be used for the local refresh group that contains the materialized view. This is the default.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information on specifying the local materialized view rollback segment using the `DBMS_REFRESH` package

If you omit *`rollback_segment`*, then the database automatically chooses the rollback segment to be used. One master rollback segment is stored for each materialized view and is validated during materialized view creation and refresh. If the materialized view is complex, then the database ignores any master rollback segment you specify.

USING ... CONSTRAINTS Clause

The `USING` ... `CONSTRAINTS` clause lets Oracle Database choose more rewrite options during the refresh operation, resulting in more efficient refresh execution. The clause lets Oracle Database use unenforced constraints, such as dimension relationships or constraints in the `RELY` state, rather than relying only on enforced constraints during the refresh operation.

The `USING TRUSTED CONSTRAINTS` clause enables you to create a materialized view on top of a table that has a non-NULL Virtual Private Database (VPD) policy on it. In this case, you must ensure that the materialized view behaves correctly. Materialized view results are computed based on the rows and columns filtered by VPD policy. Therefore, you must coordinate the materialized view definition with the VPD policy to ensure the correct results. Without the `USING TRUSTED CONSTRAINTS` clause, any VPD policy on a master table will prevent a materialized view from being created.

Note:

The `USING TRUSTED CONSTRAINTS` clause lets Oracle Database use dimension and constraint information that has been declared trustworthy by the database administrator but that has not been validated by the database. If the dimension and constraint information is valid, then performance may improve. However, if this information is invalid, then the refresh procedure may corrupt the materialized view even though it returns a success status.

If you omit this clause, then the default is `USING ENFORCED CONSTRAINTS`.

NEVER REFRESH Clause

Specify `NEVER REFRESH` to prevent the materialized view from being refreshed with any Oracle Database refresh mechanism or packaged procedure. Oracle Database will ignore any `REFRESH` statement on the materialized view issued from such a procedure. If you specify this clause, then you can perform DML operations on the materialized view. To reverse this clause, you must issue an `ALTER MATERIALIZED VIEW` ... `REFRESH` statement.

evaluation_edition_clause

You must specify this clause if *`subquery`* references an editioned object. Use this clause to specify the edition that is searched during name resolution of the editioned object—the evaluation edition.

- Specify `CURRENT EDITION` to search the edition in which this DDL statement is executed.
- Specify `EDITION` *`edition`* to search *`edition`*.
- Specifying `NULL EDITION` is equivalent to omitting the *`evaluation_edition_clause`*.

If you omit the *`evaluation_edition_clause`*, then editioned objects are invisible during name resolution and an error will result. Dropping the evaluation edition invalidates the materialized view.

> **See Also:**
>
> *Oracle Database Development Guide* for more information on specifying the evaluation edition for a materialized view

{ ENABLE | DISABLE } ON QUERY COMPUTATION

This clause lets you create a real-time materialized view or a regular view. A real-time materialized view provides fresh data to user queries even when the materialized view is not in sync with its base tables due to data changes. Instead of modifying the materialized view, the optimizer writes a query that combines the existing rows in the materialized view with changes recorded in log files (either materialized view logs or the direct loader logs). This is called on-query computation.

- Specify `ENABLE ON QUERY COMPUTATION` to create a real-time materialized view by enabling on-query computation. This allows you to directly query up-to-date data from the materialized view by specifying the `FRESH_MV` hint in the `SELECT` statement. If the materialized view is also enabled for query rewrite, then on-query computation occurs automatically during query rewrite.
- Specify `DISABLE ON QUERY COMPUTATION` to create a regular materialized view by disabling on-query computation. This is the default.

Restrictions on Real-Time Materialized Views

Real-time materialized views are subject to the following restrictions:

- Real-time materialized views cannot be used when one or more materialized view logs created on the base tables are either unusable or nonexistent.
- A real-time materialized view must be refreshable using out-of-place refresh, log-based refresh, or partition change tracking (PCT) refresh.
- A refresh-on-commit materialized view cannot be a real-time materialized view.
- If a real-time materialized view is a nested materialized view that is defined on top of one or more base materialized views, then query rewrite occurs only if all the base materialized views are fresh. If one or more base materialized views are stale, then query rewrite is not performed using this real-time materialized view.
- The cursors of queries that directly access real-time materialized views are not shared.

See Also:

- FRESH_MV Hint
- *Oracle Database Data Warehousing Guide* for more information on real-time materialized views

query_rewrite_clause

The `query_rewrite_clause` lets you specify whether the materialized view is eligible to be used for query rewrite.

ENABLE Clause

Specify `ENABLE` to enable the materialized view for query rewrite. If you also specify the `unusable_editions_clause`, then the materialized view is not enabled for query rewrite in the unusable editions.

Restrictions on Enabling Query Rewrite

Enabling of query rewrite is subject to the following restrictions:

- You can enable query rewrite only if all user-defined functions in the materialized view are `DETERMINISTIC`.
- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include `CURRENT_TIME` or `USER`, sequence values (such as the `CURRVAL` or `NEXTVAL` pseudocolumns), or the `SAMPLE` clause (which may sample different rows as the contents of the materialized view change).

Note:

- Query rewrite is disabled by default, so you must specify this clause to make materialized views eligible for query rewrite.
- After you create the materialized view, you must collect statistics on it using the `DBMS_STATS` package. Oracle Database needs the statistics generated by this package to optimize query rewrite.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on query rewrite
- *Oracle Database Data Warehousing Guide* to learn how to use refresh statistics to monitor the performance of materialized view refresh operations
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_STATS` package
- The `EXPLAIN_MVIEW` procedure of the `DBMS_MVIEW` package for help diagnosing problems with query rewrite and the `TUNE_MVIEW` procedure of the `DBMS_MVIEW` package for correction of query rewrite problems
- CREATE FUNCTION

DISABLE Clause

Specify `DISABLE` to indicate that the materialized view is not eligible for use by query rewrite. A disabled materialized view can be refreshed.

unusable_editions_clause

This clause lets you specify that the materialized view is not eligible for query rewrite in one or more editions. You can specify this clause regardless of whether you specify the `ENABLE` or `DISABLE` clause. If you specify the `DISABLE` clause, then this clause will take effect if the materialized view is subsequently enabled for query rewrite using the `ALTER MATERIALIZED VIEW` ... `ENABLE QUERY REWRITE` statement.

UNUSABLE BEFORE Clause

This clause lets you specify that the materialized view is not eligible for query rewrite in the ancestors of an edition.

- If you specify `CURRENT EDITION`, then the materialized view is not eligible for query rewrite in the ancestors of the current edition.
- If you specify `EDITION` *`edition`*, then the materialized view is not eligible for query rewrite in the ancestors of the specified *`edition`*.

UNUSABLE BEGINNING WITH Clause

This clause lets you specify that the materialized view is not eligible for query rewrite in an edition and its descendants.

- If you specify `CURRENT EDITION`, then the materialized view is not eligible for query rewrite in the current edition and its descendants.
- If you specify `EDITION` *`edition`*, then the materialized view is not eligible for query rewrite in the specified edition and its descendants.
- Specifying `NULL EDITION` is equivalent to omitting the `UNUSABLE BEGINNING WITH` clause.

The materialized view has a dependency on each edition in which it is not eligible for query rewrite. If such an edition is subsequently dropped, then the dependency is removed. However, the materialized view is not invalidated.

AS *subquery*

Specify the defining query of the materialized view. When you create the materialized view, Oracle Database executes this subquery and places the results in the materialized view. This subquery is any valid SQL subquery. However, not all subqueries are fast refreshable, nor are all subqueries eligible for query rewrite.

Notes on the Defining Query of a Materialized View

The following notes apply to materialized views:

- Oracle Database does not execute the defining query immediately if you specify `BUILD DEFERRED`.
- Oracle recommends that you qualify each table and view in the `FROM` clause of the defining query of the materialized view with the schema containing it.
- In order to create a materialized view whose defining query selects from a master table that has a Virtual Private Database (VPD) policy, you must specify the `REFRESH USING TRUSTED CONSTRAINTS` clause.

Restrictions on the Defining Query of a Materialized View

The materialized view query is subject to the following restrictions:

- The defining query of a materialized view can select from tables, views, or materialized views owned by the user `SYS`, but you cannot enable `QUERY REWRITE` on such a materialized view.
- The defining query of a materialized view cannot select from a `V$` view or a `GV$` view.
- You cannot define a materialized view with a subquery in the select list of the defining query. You can, however, include subqueries elsewhere in the defining query, such as in the `WHERE` clause.
- You cannot use the `AS OF` clause of the *`flashback_query_clause`* in the defining query of a materialized view.
- Materialized join views and materialized aggregate views with a `GROUP BY` clause cannot select from an index-organized table.
- Materialized views cannot contain columns of data type `LONG` or `LONG RAW`.
- Materialized views cannot contain virtual columns.
- You cannot create a materialized view log on a temporary table. Therefore, if the defining query references a temporary table, then this materialized view will not be eligible for `FAST` refresh, nor can you specify the `QUERY REWRITE` clause in this statement.
- If the `FROM` clause of the defining query references another materialized view, then you must always refresh the materialized view referenced in the defining query before refreshing the materialized view you are creating in this statement.
- Materialized views with join expressions in the defining query cannot have XML data type columns. The XML data types include `XMLType` and URI data type columns.

If you are creating a materialized view enabled for query rewrite, then:

- The defining query cannot contain, either directly or through a view, references to ROWNUM, USER, SYSDATE, remote tables, sequences, or PL/SQL functions that write or read database or package state.
- Neither the materialized view nor the master tables of the materialized view can be remote.

If you want the materialized view to be eligible for fast refresh using a materialized view log, or synchronous refresh using a staging log, then some additional restrictions apply.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for restrictions relating to using fast refresh and synchronous refresh
> - *Oracle Database Administrator's Guide*for more information on restrictions relating to replication
> - "Creating Materialized Join Views: Example", "Creating Subquery Materialized Views: Example", and "Creating a Nested Materialized View: Example"

Examples

The following examples require the materialized logs that are created in the "Examples" section of CREATE MATERIALIZED VIEW LOG .

Creating a Simple Materialized View: Example

The following statement creates a very simple materialized view based on the employees and table in the hr schema:

```
CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM hr.employees;
```

By default, Oracle Database creates a primary key materialized view with refresh on demand only. If a materialized view log exists on employees, then mv1 can be altered to be capable of fast refresh. If no such log exists, then only full refresh of mv1 is possible. Oracle Database uses default storage properties for mv1. The only privileges required for this operation are the CREATE MATERIALIZED VIEW system privilege, and the READ or SELECT object privilege on hr.employees.

Creating Subquery Materialized Views: Example

The following statement creates a subquery materialized view based on the customers and countries tables in the sh schema at the remote database:

```
CREATE MATERIALIZED VIEW foreign_customers
   AS SELECT * FROM sh.customers@remote cu
   WHERE EXISTS
     (SELECT * FROM sh.countries@remote co
      WHERE co.country_id = cu.country_id);
```

Creating Materialized Aggregate Views: Example

The following statement creates and populates a materialized aggregate view on the sample sh.sales table and specifies the default refresh method, mode, and time. It uses the materialized view log created in "Creating a Materialized View Log for Fast Refresh: Examples", as well as the two additional logs shown here:

```
CREATE MATERIALIZED VIEW LOG ON times
   WITH ROWID, SEQUENCE (time_id, calendar_year)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON products
   WITH ROWID, SEQUENCE (prod_id)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sales_mv
   BUILD IMMEDIATE
   REFRESH FAST ON COMMIT
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id;
```

Creating Materialized Join Views: Example

The following statement creates and populates the materialized aggregate view `sales_by_month_by_state` using tables in the sample `sh` schema. The materialized view will be populated with data as soon as the statement executes successfully. By default, subsequent refreshes will be accomplished by reexecuting the defining query of the materialized view:

```
CREATE MATERIALIZED VIEW sales_by_month_by_state
     TABLESPACE example
     PARALLEL 4
     BUILD IMMEDIATE
     REFRESH COMPLETE
     ENABLE QUERY REWRITE
     AS SELECT t.calendar_month_desc, c.cust_state_province,
        SUM(s.amount_sold) AS sum_sales
        FROM times t, sales s, customers c
        WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
        GROUP BY t.calendar_month_desc, c.cust_state_province;
```

Creating Prebuilt Materialized Views: Example

The following statement creates a materialized aggregate view for the preexisting summary table, `sales_sum_table`:

```
CREATE TABLE sales_sum_table
   (month VARCHAR2(8), state VARCHAR2(40), sales NUMBER(10,2));

CREATE MATERIALIZED VIEW sales_sum_table
   ON PREBUILT TABLE WITH REDUCED PRECISION
   ENABLE QUERY REWRITE
   AS SELECT t.calendar_month_desc AS month,
             c.cust_state_province AS state,
             SUM(s.amount_sold) AS sales
      FROM times t, customers c, sales s
      WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
      GROUP BY t.calendar_month_desc, c.cust_state_province;
```

In the preceding example, the materialized view has the same name and also has the same number of columns with the same data types as the prebuilt table. The `WITH REDUCED`

PRECISION clause allows for differences between the precision of the materialized view columns and the precision of the values returned by the subquery.

Creating Primary Key Materialized Views: Example

The following statement creates the primary key materialized view catalog on the sample table oe.product_information:

```
CREATE MATERIALIZED VIEW catalog
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 1/4096
   WITH PRIMARY KEY
   AS SELECT * FROM product_information;
```

Creating Rowid Materialized Views: Example

The following statement creates a rowid materialized view on the sample table oe.orders:

```
CREATE MATERIALIZED VIEW order_data REFRESH WITH ROWID
   AS SELECT * FROM orders;
```

Periodic Refresh of Materialized Views: Example

The following statement creates the primary key materialized view emp_data and populates it with data from the sample table hr.employees:

```
CREATE MATERIALIZED VIEW LOG ON employees
   WITH PRIMARY KEY
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW emp_data
   PCTFREE 5 PCTUSED 60
   TABLESPACE example
   STORAGE (INITIAL 50K)
   REFRESH FAST NEXT sysdate + 7
   AS SELECT * FROM employees;
```

The preceding statement does not include a START WITH parameter, so Oracle Database determines the first automatic refresh time by evaluating the NEXT value using the current SYSDATE. A materialized view log was created for the employee table, so Oracle Database performs a fast refresh of the materialized view every 7 days, beginning 7 days after the materialized view is created.

Because the materialized view conforms to the conditions for fast refresh, the database will perform a fast refresh. The preceding statement also establishes storage characteristics that the database uses to maintain the materialized view.

Automatic Refresh Times for Materialized Views: Example

The following statement creates the complex materialized view all_customers that queries the employee tables on the remote and local databases:

```
CREATE MATERIALIZED VIEW all_customers
   PCTFREE 5 PCTUSED 60
   TABLESPACE example
   STORAGE (INITIAL 50K)
   USING INDEX STORAGE (INITIAL 25K)
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY') + 15/24
   AS SELECT * FROM sh.customers@remote
```

```
      UNION
   SELECT * FROM sh.customers@local;
```

Oracle Database automatically refreshes this materialized view tomorrow at 11:00 a.m. and subsequently every Monday at 3:00 p.m. The default refresh method is `FORCE`. The defining query contains a `UNION` operator, which is not supported for fast refresh, so the database will automatically perform a complete refresh.

The preceding statement also establishes storage characteristics for both the materialized view and the index that the database uses to maintain it:

- The first `STORAGE` clause establishes the sizes of the first and second extents of the materialized view as 50 kilobytes each.
- The second `STORAGE` clause, appearing with the `USING INDEX` clause, establishes the sizes of the first and second extents of the index as 25 kilobytes each.

Creating a Fast Refreshable Materialized View: Example

The following statement creates a fast-refreshable materialized view that selects columns from the `order_items` table in the sample `oe` schema, using the `UNION` set operator to restrict the rows returned from the `product_information` and `inventories` tables using `WHERE` conditions. The materialized view logs for `order_items` and `product_information` were created in the "Examples " section of `CREATE MATERIALIZED VIEW LOG`. This example also requires a materialized view log on `oe.inventories`.

```
CREATE MATERIALIZED VIEW LOG ON inventories
   WITH (quantity_on_hand);

CREATE MATERIALIZED VIEW warranty_orders REFRESH FAST AS
  SELECT order_id, line_item_id, product_id FROM order_items o
    WHERE EXISTS
    (SELECT * FROM inventories i WHERE o.product_id = i.product_id
      AND i.quantity_on_hand IS NOT NULL)
  UNION
    SELECT order_id, line_item_id, product_id FROM order_items
    WHERE quantity > 5;
```

The materialized view `warranty_orders` requires that materialized view logs be defined on `order_items` (with `product_id` as a join column) and on inventories (with `quantity_on_hand` as a filter column). See "Specifying Filter Columns for Materialized View Logs: Example" and "Specifying Join Columns for Materialized View Logs: Example".

Creating a Nested Materialized View: Example

The following example uses the materialized view from the preceding example as a master table to create a materialized view tailored for a particular sales representative in the sample `oe` schema:

```
CREATE MATERIALIZED VIEW my_warranty_orders
   AS SELECT w.order_id, w.line_item_id, o.order_date
   FROM warranty_orders w, orders o
   WHERE o.order_id = o.order_id
   AND o.sales_rep_id = 165;
```

CREATE MATERIALIZED VIEW LOG

Purpose

Use the `CREATE MATERIALIZED VIEW LOG` statement to create a **materialized view log**, which is a table associated with the master table of a materialized view.

> **Note:**
>
> The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

Materialized view logs are used for two types of materialized view refreshes: fast refresh and synchronous refresh.

Fast refresh uses a conventional materialized view log. During a fast refresh (also called an incremental refresh), when DML changes are made to master table data, Oracle Database stores rows describing those changes in the materialized view log and then uses the materialized view log to refresh materialized views based on the master table.

Synchronous refresh uses a special type of materialized view log called a **staging log**. During a synchronous refresh, DML changes are first described in the staging log and then applied to the master tables and the materialized views simultaneously. This guarantees that the master table data and materialized view data are in sync throughout the refresh process. This refresh method is useful in data warehousing environments.

Without a materialized view log, Oracle Database must reexecute the materialized view query to refresh the materialized view. This process is called a **complete refresh**. Usually, a complete refresh takes more time to complete than a fast refresh or a synchronous refresh.

A materialized view log is located in the master database in the same schema as the master table. A master table can have only one materialized view log defined on it.

To fast refresh or synchronous refresh a materialized join view, you must create a materialized view log for each of the tables referenced by the materialized view.

Fast refresh supports two types of materialized view logs: timestamp-based materialized view logs and commit SCN-based materialized view logs. Timestamp-based materialized view logs use timestamps and require some setup operations when preparing to refresh the materialized view. Commit SCN-based materialized view logs use commit SCN data rather than timestamps, which removes the need for the setup operations and thus can improve the speed of the materialized view refresh. If you specify the `COMMIT SCN` clause, then a commit SCN-based materialized view log is created. Otherwise, a time-stamp based materialized view log is created. Note that only new materialized view logs can take advantage of `COMMIT SCN`. Existing materialized view logs cannot be altered to add `COMMIT SCN` unless they are dropped and recreated. Refer to *Oracle Database Data Warehousing Guide* for more information.

Synchronous refresh supports only timestamp-based staging logs.

See Also:

- CREATE MATERIALIZED VIEW , ALTER MATERIALIZED VIEW , *Oracle Database Concepts*, *Oracle Database Data Warehousing Guide*, and *Oracle Database Administrator's Guide* for information on materialized views in general
- ALTER MATERIALIZED VIEW LOG for information on modifying a materialized view log
- DROP MATERIALIZED VIEW LOG for information on dropping a materialized view log
- *Oracle Database Utilities* for information on using direct loader logs

Prerequisites

The privileges required to create a materialized view log directly relate to the privileges necessary to create the underlying objects associated with a materialized view log.

- If you own the master table, then you can create an associated materialized view log if you have the `CREATE TABLE` privilege.
- If you are creating a materialized view log for a table in another user's schema, then you must have the `CREATE ANY TABLE` and `COMMENT ANY TABLE` system privileges, as well as either the `READ` or `SELECT` object privilege on the master table or the `READ ANY TABLE` or `SELECT ANY TABLE` system privilege.

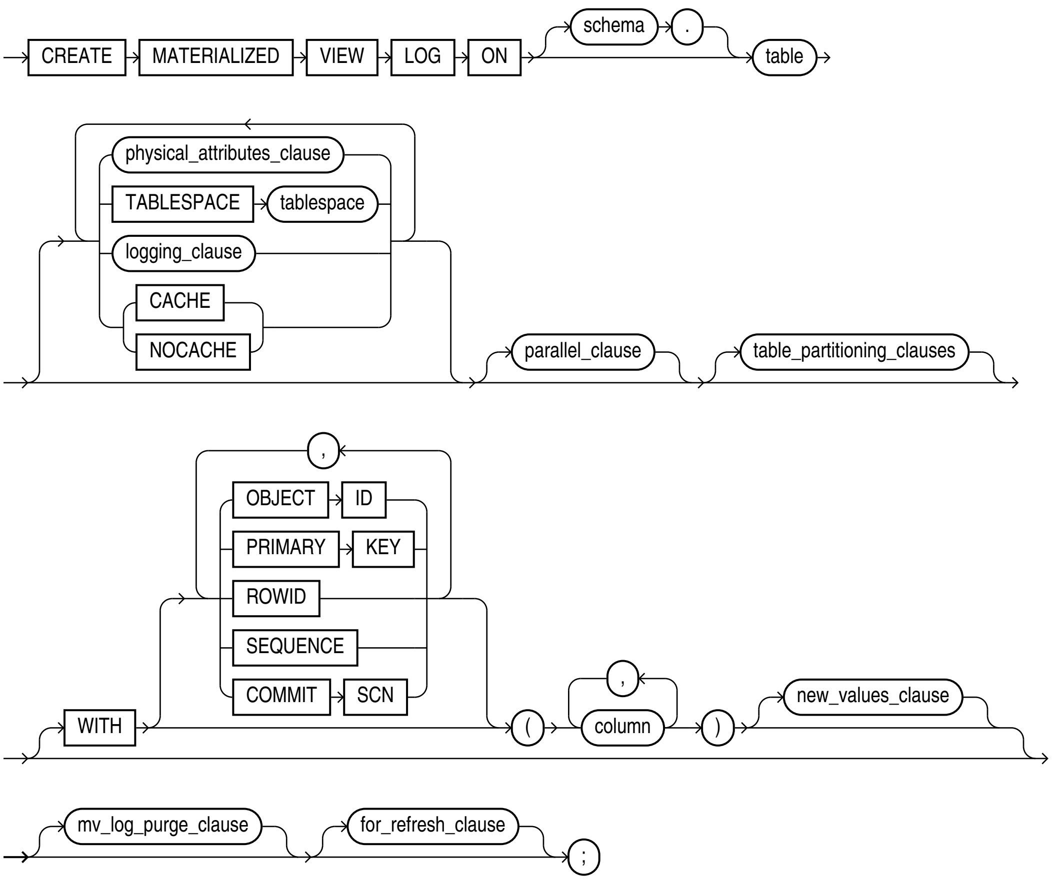
In either case, the owner of the materialized view log must have sufficient quota in the tablespace intended to hold the materialized view log or must have the `UNLIMITED TABLESPACE` system privilege.

See Also:

Oracle Database Data Warehousing Guide for more information about the prerequisites for creating a materialized view log

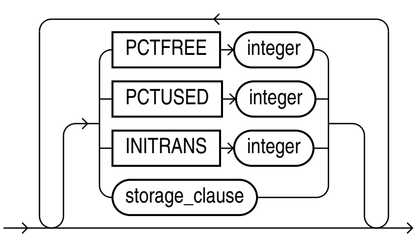
Syntax

***create_materialized_vw_log*::=**



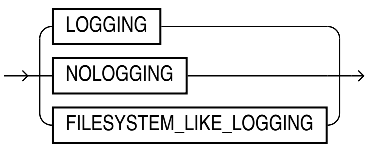
(*physical_attributes_clause*::=, *logging_clause*::=, *parallel_clause*::=, *table_partitioning_clauses*::= (in `CREATE TABLE`), *new_values_clause*::=, *mv_log_purge_clause*::=, *for_refresh_clause*::=.)

***physical_attributes_clause*::=**

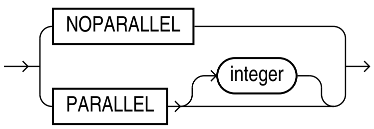


(*storage_clause*::=)

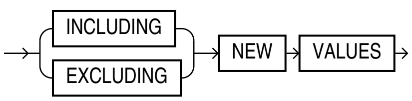
***logging_clause*::=**



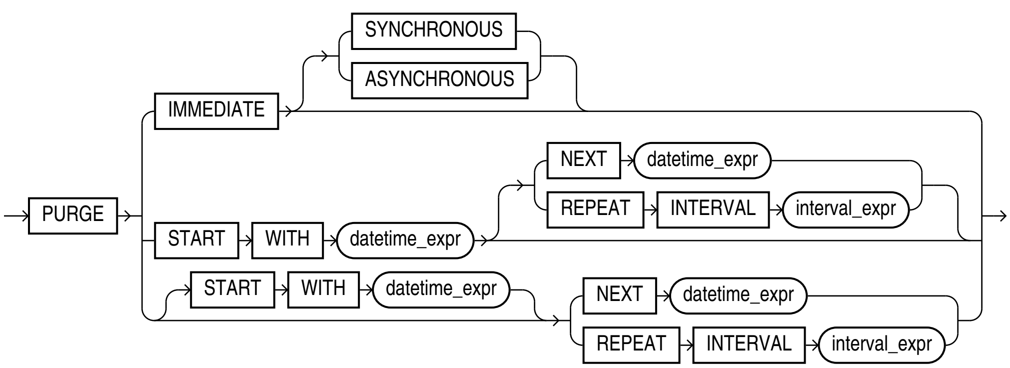
***parallel_clause*::=**



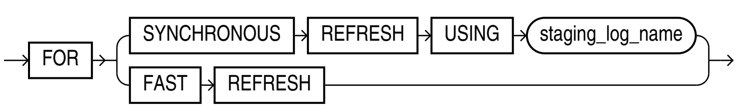
***new_values_clause*::=**



***mv_log_purge_clause*::=**



***for_refresh_clause*::=**



Semantics

schema

Specify the schema containing the materialized view log master table. If you omit *schema*, then Oracle Database assumes the master table is contained in your own schema. Oracle Database creates the materialized view log in the schema of its master table. You cannot create a materialized view log for a table in the schema of the user SYS.

table

Specify the name of the master table for which the materialized view log is to be created. Oracle Database encrypts any columns in the materialized view log that are encrypted in the master table, using the same encryption algorithm.

Restrictions on Master Tables of Materialized View Logs

The following restrictions apply to master tables of materialized view logs:

- You cannot create a materialized view log for a temporary table or for a view.
- You cannot create a materialized view log for a master table with a virtual column.

> **See Also:**
>
> "Creating a Materialized View Log for Fast Refresh: Examples"

physical_attributes_clause

Use the *physical_attributes_clause* to define physical and storage characteristics for the materialized view log.

> **See Also:**
>
> *physical_attributes_clause* and storage_clause for a complete description these clauses, including default values

TABLESPACE Clause

Specify the tablespace in which the materialized view log is to be created. If you omit this clause, then the database creates the materialized view log in the default tablespace of the schema of the materialized view log.

logging_clause

Specify either LOGGING or NOLOGGING to establish the logging characteristics for the materialized view log. The default is the logging characteristic of the tablespace in which the materialized view log resides.

> **See Also:**
>
> *logging_clause* for a full description of this clause

CACHE | NOCACHE

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this log are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

`NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list. The default is `NOCACHE`.

> **Note:**
>
> `NOCACHE` has no effect on materialized view logs for which you specify `KEEP` in the *`storage_clause`*.

> **See Also:**
>
> CREATE TABLE for information about specifying `CACHE` or `NOCACHE`

parallel_clause

The *`parallel_clause`* lets you indicate whether parallel operations will be supported for the materialized view log.

For complete information on this clause, refer to *parallel_clause* in the documentation on `CREATE TABLE`.

table_partitioning_clauses

Use the *`table_partitioning_clauses`* to indicate that the materialized view log is partitioned on specified ranges of values or on a hash function. Partitioning of materialized view logs is the same as partitioning of tables.

> **See Also:**
>
> *table_partitioning_clauses* in the `CREATE TABLE` documentation

WITH Clause

Use the `WITH` clause to indicate whether the materialized view log should record the primary key, rowid, object ID, or a combination of these row identifiers when rows in the master are changed. You can also use this clause to add a sequence to the materialized view log to provide additional ordering information for its records.

This clause also specifies whether the materialized view log records additional columns that might be referenced as **filter columns**, which are non-primary-key columns referenced by subquery materialized views, or **join columns**, which are non-primary-key columns that define a join in the subquery `WHERE` clause.

If you omit this clause, or if you specify the clause without `PRIMARY KEY`, `ROWID`, or `OBJECT ID`, then the database stores primary key values by default. However, the database does not store primary key values implicitly if you specify only `OBJECT ID` or `ROWID` at create time. A primary key log, created either explicitly or by default, performs additional checking on the primary key constraint.

OBJECT ID

Specify `OBJECT ID` to indicate that the system-generated or user-defined object identifier of every modified row should be recorded in the materialized view log.

Restriction on OBJECT ID

You can specify `OBJECT ID` only when creating a log on an object table, and you cannot specify it for storage tables.

PRIMARY KEY

Specify `PRIMARY KEY` to indicate that the primary key of all rows changed should be recorded in the materialized view log.

ROWID

Specify `ROWID` to indicate that the rowid of all rows changed should be recorded in the materialized view log.

SEQUENCE

Specify `SEQUENCE` to indicate that a sequence value providing additional ordering information should be recorded in the materialized view log. Sequence numbers are necessary to support fast refresh after some update scenarios.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on the use of sequence numbers in materialized view logs and for examples that use this clause

COMMIT SCN

Without the `COMMIT SCN` clause, the materialized view log is based on timestamps and requires some setup operations when preparing to refresh the materialized view. Specify `COMMIT SCN` to instruct the database to use commit SCN data rather than timestamps. This setting removes the need for the setup operations and thus can improve the speed of the materialized view refresh.

You can create the following types of local materialized views (including both `ON COMMIT` and `ON DEMAND`) on master tables with commit SCN-based materialized view logs:

- Materialized aggregate views, including materialized aggregate views on a single table
- Materialized join views
- Primary-key-based and rowid-based single table materialized views
- `UNION ALL` materialized views, where each `UNION ALL` branch is one of the above materialized view types

You cannot create remote materialized views on master tables with commit SCN-based materialized view logs.

Restrictions on COMMIT SCN

The following restrictions apply to `COMMIT SCN`:

- Use of `COMMIT SCN` on a table with one or more LOB columns is not supported and causes `ORA-32421`.
- Creating a materialized view on master tables with different types of materialized view logs (that is, a master table with timestamp-based materialized view logs and a master table with commit SCN-based materialized view logs) is not supported and causes `ORA-32414`.
- If you specify `COMMIT SCN`, then you cannot specify `FOR SYNCHRONOUS REFRESH`.

column

Specify the columns whose values you want to be recorded in the materialized view log for all rows that are changed. Typically these columns are filter columns and join columns.

Restrictions on the WITH Clause

This clause is subject to the following restrictions:

- You can specify only one `PRIMARY KEY`, one `ROWID`, one `OBJECT ID`, one `SEQUENCE`, and one column list for each materialized view log.
- Primary key columns are implicitly recorded in the materialized view log. Therefore, you cannot specify any of the following combinations if *`column`* contains one of the primary key columns:

```
WITH ... PRIMARY KEY ... (column)
WITH ... (column) ... PRIMARY KEY
WITH (column)
```

See Also:

- CREATE MATERIALIZED VIEW for information on explicit and implicit inclusion of materialized view log values
- *Oracle Database Administrator’s Guide* for more information about filter columns and join columns
- "Specifying Filter Columns for Materialized View Logs: Example" and "Specifying Join Columns for Materialized View Logs: Example"

NEW VALUES Clause

The NEW VALUES clause lets you determine whether Oracle Database saves both old and new values for update DML operations in the materialized view log.

> **See Also:**
>
> "Including New Values in Materialized View Logs: Example"

INCLUDING

Specify INCLUDING to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, then you must specify INCLUDING.

EXCLUDING

Specify EXCLUDING to disable the recording of new values in the log. This is the default. You can use this clause to avoid the overhead of recording new values. Do not use this clause if you have a fast-refreshable single-table materialized aggregate view defined on the master table.

mv_log_purge_clause

Use this clause to specify the purge time for the materialized view log.

- IMMEDIATE SYNCHRONOUS: the materialized view log is purged immediately after refresh. This is the default.
- IMMEDIATE ASYNCHRONOUS: the materialized view log is purged in a separate Oracle Scheduler job after the refresh operation.
- START WITH, NEXT, and REPEAT INTERVAL set up a scheduled purge that is independent of the materialized view refresh and is initiated during CREATE or ALTER MATERIALIZED VIEW LOG statement. This is very similar to scheduled refresh syntax in a CREATE or ALTER MATERIALIZED VIEW statement:
  - The START WITH datetime expression specifies when the purge starts.
  - The NEXT datetime expression computes the next run time for the purge.

  If you specify REPEAT INTERVAL, then the next run time will be: SYSDATE + *interval_expr*.

  A CREATE MATERIALIZED VIEW LOG statement with a scheduled purge creates an Oracle Scheduler job to perform log purge. The job calls the DBMS_SNAPSHOT.PURGE_LOG procedure to purge the materialized view logs. This process allows you to amortize the purging costs over several materialized view refreshes.

Restriction on *mv_log_purge_clause*

This clause is not valid for materialized view logs on temporary tables.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on purging materialized view logs

for_refresh_clause

Use this clause to specify the refresh method for which the materialized view log will be used. You can specify only one refresh method for any given master table.

FOR SYNCHRONOUS REFRESH

Specify this clause to create a staging log that can be used for synchronous refresh. Use *staging_log_name* to specify the name of the staging log to be created. The staging log will be created in the schema in which the master table resides.

After you create the staging log, you cannot perform DML operations directly on the master table. You must use the procedures in the DBMS_SYNC_REFRESH package to prepare and execute change data operations.

Restrictions on Synchronous Refresh

The following restrictions apply to synchronous refresh:

- If you specify FOR SYNCHRONOUS REFRESH, then you cannot specify COMMIT SCN.
- To be eligible for synchronous refresh, the master table must satisfy the following criteria:
  - If the master table is a fact table, then it must be partitioned.
  - The master table must have a key. If the master table is a dimension table, then it must have a primary key defined on it. If the master table is a fact table, then the set of columns that are the foreign keys of the dimension tables joined to the fact table are deemed to be the key.
  - The master table cannot have a non-NULL Virtual Private Database (VPD) policy or a trigger defined on it.

  Oracle Database may allow you to create a staging log on a master table even if all of the preceding criteria are not met. However, the master table will not be eligible for synchronous refresh.
- Any existing materialized views on the master table must be refresh-on-demand materialized views. If an existing materialized view is a refresh-on-commit materialized view, then you must change it to a refresh-on-demand materialized view with the *alter_mv_refresh* clause of ALTER MATERIALIZED VIEW before you create the staging log.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for the complete steps for using synchronous refresh
> - *Oracle Database PL/SQL Packages and Types Reference* for information on the DBMS_SYNC_REFRESH package

FOR FAST REFRESH

Specify this clause to create a materialized view log that can be used for fast refresh. The materialized view log will be created in the same schema in which the master table resides. This is the default.

Examples

Creating a Materialized View Log for Fast Refresh: Examples

The following statement creates a materialized view log on the `oe.customers` table that specifies physical and storage characteristics:

```
CREATE MATERIALIZED VIEW LOG ON customers
   PCTFREE 5
   TABLESPACE example
   STORAGE (INITIAL 10K);
```

The materialized view log on `customers` supports fast refresh for primary key materialized views only.

The following statement creates another version of the materialized view log with the `ROWID` clause, which enables fast refresh for more types of materialized views:

```
CREATE MATERIALIZED VIEW LOG ON customers WITH PRIMARY KEY, ROWID;
```

This materialized view log on `customers` makes fast refresh possible for rowid materialized views and for materialized join views. To provide for fast refresh of materialized aggregate views, you must also specify the `SEQUENCE` and `INCLUDING NEW VALUES` clauses, as shown in the example that follows.

Specify a Purge Repeat Interval for a Materialized View Log: Example

The following statement creates a materialized view log on the `oe.orders` table. The contents of the log will be purged once every five days, beginning five days after the creation date of the materialized view log:

```
CREATE MATERIALIZED VIEW LOG ON orders
  PCTFREE 5
  TABLESPACE example
  STORAGE (INITIAL 10K)
  PURGE REPEAT INTERVAL '5' DAY;
```

Specifying Filter Columns for Materialized View Logs: Example

The following statement creates a materialized view log on the `sh.sales` table and is used in "Creating Materialized Aggregate Views: Example". It specifies as filter columns all of the columns of the table referenced in that materialized view.

```
CREATE MATERIALIZED VIEW LOG ON sales
   WITH ROWID, SEQUENCE(amount_sold, time_id, prod_id)
   INCLUDING NEW VALUES;
```

Specifying Join Columns for Materialized View Logs: Example

The following statement creates a materialized view log on the `order_items` table of the sample `oe` schema. The log records primary keys and `product_id`, which is used as a join column in "Creating a Fast Refreshable Materialized View: Example".

```
CREATE MATERIALIZED VIEW LOG ON order_items WITH (product_id);
```

Including New Values in Materialized View Logs: Example

The following example creates a materialized view log on the oe.product_information table that specifies INCLUDING NEW VALUES:

```
CREATE MATERIALIZED VIEW LOG ON product_information
   WITH ROWID, SEQUENCE (list_price, min_price, category_id), PRIMARY KEY
   INCLUDING NEW VALUES;
```

You could create the following materialized aggregate view to use the product_information log:

```
CREATE MATERIALIZED VIEW products_mv
   REFRESH FAST ON COMMIT
   AS SELECT SUM(list_price - min_price), category_id
         FROM product_information
         GROUP BY category_id;
```

This materialized view is eligible for fast refresh because the log defined on its master table includes both old and new values.

Creating a Staging Log for Synchronous Refresh: Example

The following statement creates a staging log on the sh.sales fact table. The staging log is named mystage_log and is stored in the sh schema. It can be used for synchronous refresh.

```
CREATE MATERIALIZED VIEW LOG ON sales
   PCTFREE 5
   TABLESPACE example
   STORAGE (INITIAL 10K)
   FOR SYNCHRONOUS REFRESH USING mystage_log;
```

CREATE MATERIALIZED ZONEMAP

Purpose

Use the CREATE MATERIALIZED ZONEMAP statement to create a zone map.

A **zone map** is a special type of materialized view that stores information about zones. A **zone** is a set of contiguous data blocks on disk that stores the values of one or more table columns. Multiple zones are usually required to store all of the values of the table columns. A zone map tracks the minimum and maximum table column values stored in each zone.

Zone maps enable you to reduce the I/O and CPU costs of table scans. When a SQL statement contains predicates on columns in a zone map, the database compares the predicate values to the minimum and maximum table column values stored in each zone to determine which zones to read during SQL execution.

Oracle Database supports the following types of zone maps:

- A **basic zone map** is defined on a single table and maintains zone information for specified columns in that table.

  You can create a basic zone map either by specifying the *create_zonemap_on_table* clause, or by specifying the *create_zonemap_as_subquery* clause where the FROM clause of the defining subquery specifies a single table.
- A **join zone map** is defined on two or more joined tables and maintains zone information for specified columns in any of the joined tables.

You can create a join zone map by specifying the *create_zonemap_as_subquery* clause. The FROM clause of the defining subquery must specify a table that is left outer joined with one or more other tables.

Zone maps are commonly used with star schemas in data warehousing environments. However, a star schema is not a requirement for creating a zone map. In either case, this reference uses star schema terminology to refer to the tables in a zone map. In a join zone map, the outer table of the join(s) is referred to as the **fact table**, and the tables with which this table is joined are referred to as **dimension tables**. Collectively these tables are called the **base tables** of the zone map. In a basic zone map, the single table on which the zone map is defined is referred to as both the fact table and the base table of the zone map.

A base table of a zone map can be a partitioned or composite-partitioned table. In this case, the zone map maintains minimum and maximum column values for each partition (and subpartition) as well as for each zone.

You can create zone maps for use with or without attribute clustering:

- To create a zone map for use with attribute clustering, use either of the following methods:
  - Use the CREATE MATERIALIZED ZONEMAP statement and include attribute clustered columns in the zone map. Refer to the *attribute_clustering_clause* of CREATE TABLE and the *attribute_clustering_clause* clause of ALTER TABLE for more information.
  - Specify the WITH MATERIALIZED ZONEMAP clause while creating or modifying an attribute clustered table. Refer to the zonemap_clause of CREATE TABLE and the MODIFY CLUSTERING clause of ALTER TABLE for more information.
- To create a zone map for use without attribute clustering, use the CREATE MATERIALIZED ZONEMAP statement and include columns that are not attribute clustered in the zone map.

See Also:

Oracle Database Data Warehousing Guide for more information on zone maps

Prerequisites

To create a zone map **in your own schema:**

- You must have the CREATE MATERIALIZED VIEW system privilege and either the CREATE TABLE or CREATE ANY TABLE system privilege.
- You must have access to any base tables of the zone map that you do not own, either through a READ or SELECT object privilege on each of the tables or through the READ ANY TABLE or SELECT ANY TABLE system privilege.

To create a zone map **in another user's schema:**

- You must have the CREATE ANY MATERIALIZED VIEW system privilege.
- The owner of the zone map must have the CREATE TABLE system privilege. The owner must also have access to any base tables of the zone map that the schema

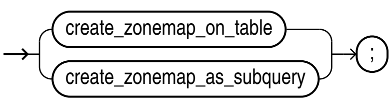
owner does not own, either through a READ or SELECT object privilege on each of the tables or through the READ ANY TABLE or SELECT ANY TABLE system privilege.

To create a refresh-on-commit zone map (REFRESH ON COMMIT clause), in addition to the preceding privileges, you must have the ON COMMIT REFRESH object privilege on any base tables that you do not own or you must have the ON COMMIT REFRESH system privilege. Unlike materialized views, you can create a refresh-on-commit zone map even if there are no materialized view logs on the base tables.

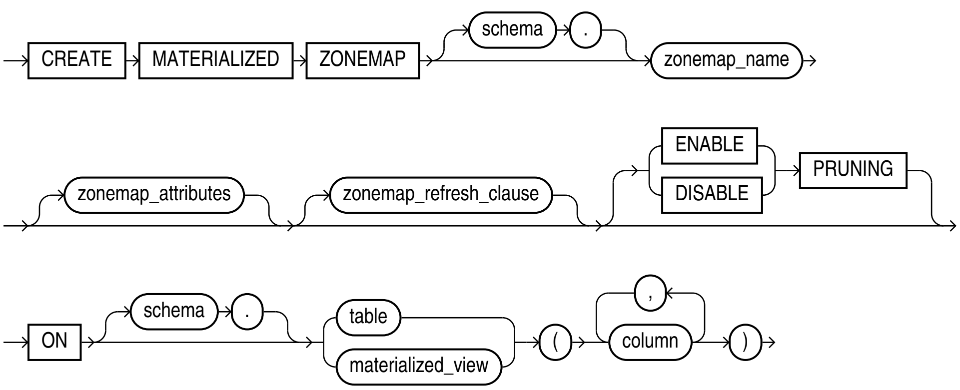
When you create a zone map, Oracle Database creates one internal table and at least one index, all in the schema of the zone map. Oracle Database uses these objects to maintain the zone map data. You must have the privileges necessary to create these objects, and you must have sufficient quota in the target tablespace to store these objects or you must have the UNLIMITED TABLESPACE system privilege.

Syntax

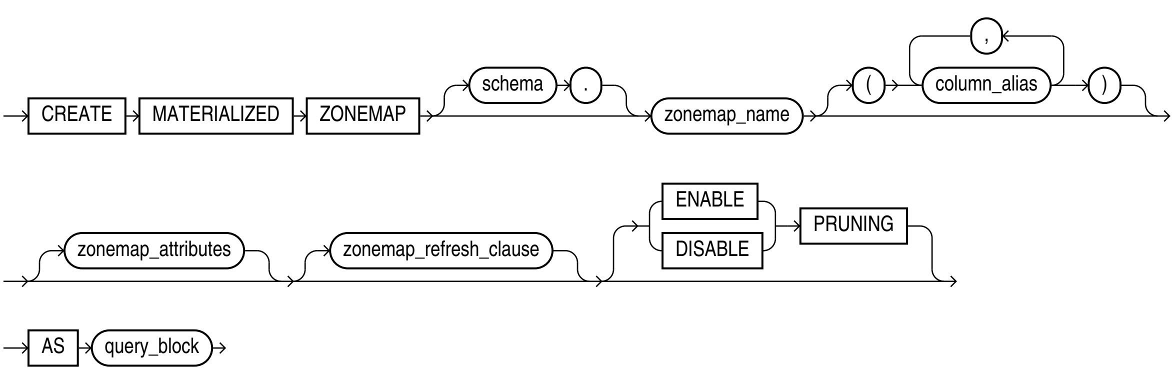
***create_materialized_zonemap*::=**



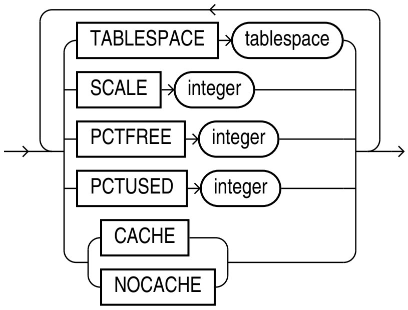
***create_zonemap_on_table*::=**



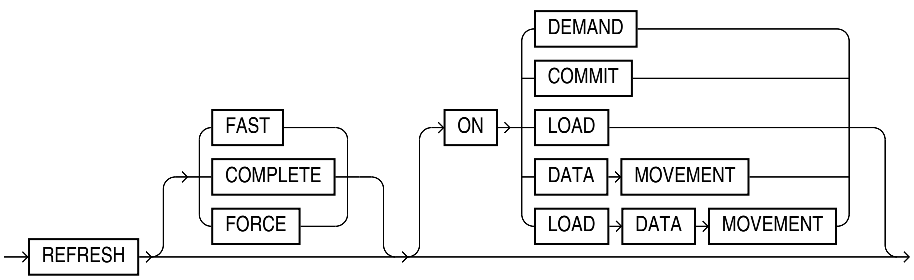
***create_zonemap_as_subquery*::=**



***zonemap_attributes*::=**



***zonemap_refresh_clause*::=**



> **Note:**
>
> When specifying the *`zonemap_refresh_clause`*, you must specify at least one clause after the `REFRESH` keyword.

Semantics

create_zonemap_on_table

Use this clause to create a basic zone map.

ON Clause

In the `ON` clause, first specify the fact table for the zone map, and then inside the parentheses specify one or more columns of the fact table to be included in the zone map.

For each specified fact table *`column`*, Oracle creates two columns in the zone map. These two columns contain the minimum and maximum values of the fact table column in each zone. Oracle generates names for the zone map columns of the form `MIN_1_`*`column`* and `MAX_1_`*`column`* for the first specified fact table *`column`*, `MIN_2_`*`column`* and `MAX_2_`*`column`* for the second specified fact table *`column`*, and so on.

If you omit *`schema`*, then Oracle assumes the fact table is in your own schema. The fact table can be a table or a materialized view

create_zonemap_as_subquery

Use this clause to create a basic zone map or a join zone map. To create a basic zone map, specify a single base table in the `FROM` clause of the defining subquery. To create a join zone map, specify a table that is left outer joined to one or more other tables in the `FROM` clause of the defining subquery.

column_alias

You can specify a column alias for each table column to be included in the zone map. The column alias list explicitly resolves any column name conflict, eliminating the need to specify aliases in the `SELECT` list of the defining subquery. If you specify any column alias in this clause, then you must specify an alias for each column in the `SELECT` list of the defining subquery. The first column alias you specify must be `ZONE_ID$`, which corresponds to the first column in the `SELECT` list, the `SYS_OP_ZONE_ID` function expression.

AS *query_block*

Specify the defining subquery of the zone map. The subquery must consist of a single *`query_block`*. You can specify only the `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses of *`query_block`*, and those clauses must satisfy the following requirements:

- The first column in the `SELECT` list must be the `SYS_OP_ZONE_ID` function expression. Refer to SYS_OP_ZONE_ID for more information.
- The remaining columns in the `SELECT` list must be function expressions that return minimum and maximum values for the columns you want to include in the zone map. For each column, specify a pair of function expressions of the following form:

  ```
  MIN([table.]column), MAX([table.]column)
  ```

  For *`table`*, specify the name or table alias for the table that contains the column. The table can be a fact table or dimension table. For *`column`*, specify the name or column alias for the column.
- The `FROM` clause can specify a fact table alone, or a fact table and one or more dimension tables with each dimension table left outer joined to the fact table. You can specify `LEFT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to dimension table columns in the join condition in the `WHERE` clause. You can optionally specify a table alias for any of the tables in the `FROM` clause. Fact tables and dimension tables can be tables or materialized views.
- In the `WHERE` clause, you can specify only left outer join conditions using the outer join operator(+).
- You must specify a `GROUP BY` clause with the same `SYS_OP_ZONE_ID` function expression that you specified for the first column of the `SELECT` list.

schema

Specify the schema to contain the zone map. If you omit *`schema`*, then Oracle Database creates the zone map in your schema.

zonemap_name

Specify the name of the zone map to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

zonemap_attributes

Use this clause to specify the following attributes for the zone map: `TABLESPACE`, `SCALE`, `PCTFREE`, `PCTUSED`, and `CACHE` or `NOCACHE`.

TABLESPACE

Specify the *`tablespace`* in which the zone map is to be created. If you omit this clause, then Oracle Database creates the zone map in the default tablespace of the schema containing the zone map.

SCALE

This clause lets you specify the zone map scale, which determines the number of contiguous disk blocks that form a zone. The scale is an integer value that represents a power of 2. For example, a scale of 10 means up to 2 raised to the 10th power, or 1024, contiguous disk blocks will form a zone. For *`integer`*, specify a value between 4 and 16, inclusive. The recommended value is 10; this is the default.

PCTFREE

Specify an *`integer`* representing the percentage of space in each data block of the zone map reserved for future updates to rows of the zone map. The integer value must be between 0 and 99, inclusive. The default value is 10. Refer to *physical_attributes_clause* for more information on the `PCTFREE` parameter.

PCTUSED

Specify an *`integer`* representing the minimum percentage of used space that Oracle maintains for each data block of the zone map. The integer value must be between 0 and 99, inclusive. The default value is 40. Refer to *physical_attributes_clause* for more information on the `PCTUSED` parameter.

CACHE | NOCACHE

For data that will be accessed frequently, `CACHE` specifies that the blocks retrieved for this zone map are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed.

`NOCACHE` specifies that the blocks are placed at the least recently used end of the LRU list. The default is `NOCACHE`.

zonemap_refresh_clause

Use this clause to specify the default refresh method and mode for the zone map. If you do not specify a refresh method (`FAST`, `COMPLETE`, or `FORCE`), then `FORCE` is the default method. If you do not specify a refresh mode (`ON` clauses), then `ON LOAD DATA MOVEMENT` is the default mode.

FAST

Specify `FAST` to indicate the fast refresh method, which performs the refresh according to the changes that have occurred to the base tables. While zone maps are internally implemented as a type of materialized view, materialized view logs on base tables are not needed to perform a fast refresh of a zone map

COMPLETE

Specify COMPLETE to indicate the complete refresh method, which is implemented by executing the defining query of the zone map. If you request a complete refresh, then Oracle Database performs a complete refresh even if a fast refresh is possible.

FORCE

Specify FORCE to indicate that when a refresh occurs, Oracle Database will perform a fast refresh if one is possible or a complete refresh if fast refresh is not possible. This is the default.

ON DEMAND

Specify ON DEMAND to indicate that database will not refresh the zone map unless you manually issue an ALTER MATERIALIZED ZONEMAP ... REBUILD statement. If you specify this clause, then the zone map is referred to as a refresh-on-demand zone map. Refer to REBUILD in the documentation on ALTER MATERIALIZED ZONEMAP for more information on rebuilding a zone map.

ON COMMIT

Specify ON COMMIT to indicate that a refresh is to occur whenever the database commits a transaction that operates on a base table of the zone map. If you specify this clause, then the zone map is referred to as a refresh-on-commit zone map. This clause may increase the time taken to complete the commit, because the database performs the refresh operation as part of the commit process.

ON LOAD

Specify ON LOAD to indicate that a refresh is to occur at the end of a direct-path insert (serial or parallel) resulting either from an INSERT or a MERGE operation.

ON DATA MOVEMENT

Specify ON DATA MOVEMENT to indicate that a refresh is to occur at the end of the following data movement operations:

- Data redefinition using the DBMS_REDEFINITION package
- Table partition maintenance operations that are specified by the following clauses of ALTER TABLE: *coalesce_table*, *merge_table_partitions*, *move_table_partition*, and *split_table_partition*

ON LOAD DATA MOVEMENT

Specify ON LOAD DATA MOVEMENT to indicate that a refresh is to occur at the end of a direct-path insert or a data movement operation. This is the default.

ENABLE | DISABLE PRUNING

This clause lets you control the use of the zone map for pruning.

- Specify ENABLE PRUNING to enable use of the zone map for pruning. This is the default.
- Specify DISABLE PRUNING to disable use of the zone map for pruning. The optimizer will not use the zone map for pruning, but the database will continue to maintain the zone map.

If the setting is ENABLE PRUNING, then the optimizer will consider using the zone map for pruning during SQL operations that include any of the following conditions:

- Comparison conditions: =, <=, <, >=, >

The condition must be a simple comparison condition that has a column name on one side and a literal or bind variable on the other side. For example:

```
WHERE country_name = 'United States of America'
WHERE country_name = :country1
WHERE 10000 >= salary
```

- `IN` condition

  The `IN` condition must have a column name on the left side and an expression list of literals or bind variables on the right side. For example:

```
WHERE country_name IN ('Germany', 'India', 'United Kingdom')
WHERE country_name IN (:country1, :country2, :country3)
WHERE prod_id IN (20, 48, 132, 143)
```

- `LIKE` condition

  The `LIKE` condition must have a column name on the left side and a text literal on the right side. The text literal is the pattern for the `LIKE` condition and it must contain at least one pattern matching character. Valid pattern matching characters are the underscore (_), which matches exactly one character, and the percent sign (`%`), which matches zero or more characters. The first character of the pattern cannot be a pattern matching character. For example:

```
WHERE prod_name LIKE 'DVD%'
WHERE prod_name LIKE 'Model%Cordless%Battery'
WHERE prod_name LIKE 'CD%Pack of _'
```

> **See Also:**
>
> Conditions for more information on conditions

Restrictions on Zone Maps

Zone maps are subject to the following restrictions:

- A table can be a fact table for at most one zone map. A table can be a dimension table for multiple zone maps. A table can be a fact table for one zone map and a dimension table for other zone maps.
- A base table of a zone map cannot be an external table, an index-organized table, a remote table, a temporary table, or a view.
- A base table of a zone map cannot be in the schema of the user `SYS`.
- A zone map cannot be partitioned.
- You can define a zone map on a column of any scalar data type other than `BFILE`, `BLOB`, `CLOB`, `LONG`, `LONG RAW`, or `NCLOB`.
- All joins specified in the defining subquery of a zone map must be left outer equijoins with the fact table on the left side.
- If the `FROM` clause of the defining subquery for a zone map references a materialized view, then you must refresh that materialized view before refreshing the zone map.
- You cannot perform DML operations directly on a zone map.

- Each column of the zone map must have one of the following declared collations: BINARY or USING_NLS_COMP.

Examples

The following statement creates a basic zone map called sales_zmap. The zone map tracks columns cust_id and prod_id in the table sales.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  ON sales(cust_id, prod_id);
```

The following statement creates a basic zone map called sales_zmap that is similar to the zone map created in the previous example. However, this statement uses a defining subquery to create the zone map.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(rowid),
            MIN(cust_id), MAX(cust_id),
            MIN(prod_id), MAX(prod_id)
     FROM sales
     GROUP BY SYS_OP_ZONE_ID(rowid);
```

The following statement creates a join zone map called sales_zmap. The fact table for the zone map is sales and the zone map has one dimension table: customers. The zone map tracks two columns in the dimension table: cust_state_province and cust_city.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(s.rowid),
            MIN(cust_state_province), MAX(cust_state_province),
            MIN(cust_city), MAX(cust_city)
     FROM sales s
          LEFT OUTER JOIN customers c ON s.cust_id = c.cust_id
     GROUP BY SYS_OP_ZONE_ID(s.rowid);
```

The following statement creates a join zone map called sales_zmap. The fact table for the zone map is sales and the zone map has two dimension tables: products and customers. The zone map tracks five columns in the dimension tables: prod_category and prod_subcategory in the products table, and country_id, cust_state_province, and cust_city in the customers table.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(s.rowid),
            MIN(prod_category), MAX(prod_category),
            MIN(prod_subcategory), MAX(prod_subcategory),
            MIN(country_id), MAX(country_id),
            MIN(cust_state_province), MAX(cust_state_province),
            MIN(cust_city), MAX(cust_city)
    FROM sales s
       LEFT OUTER JOIN products p ON s.prod_id = p.prod_id
       LEFT OUTER JOIN customers c ON s.cust_id = c.cust_id
    GROUP BY sys_op_zone_id(s.rowid);
```

The following statement creates a join zone map that is identical to the zone map created in the previous example. The only difference is that the previous example uses the LEFT OUTER JOIN syntax in the FROM clause and the following example uses the outer join operator (+) in the WHERE clause.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
  AS SELECT SYS_OP_ZONE_ID(s.rowid),
```

```
        MIN(prod_category), MAX(prod_category),
        MIN(prod_subcategory), MAX(prod_subcategory),
        MIN(country_id), MAX(country_id),
        MIN(cust_state_province), MAX(cust_state_province),
        MIN(cust_city), MAX(cust_city)
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id(+) AND
      s.cust_id = c.cust_id(+)
GROUP BY sys_op_zone_id(s.rowid);
```

CREATE OPERATOR

Purpose

Use the `CREATE OPERATOR` statement to create a new operator and define its bindings.

Operators can be referenced by indextypes and by SQL queries and DML statements. The operators, in turn, reference functions, packages, types, and other user-defined objects.

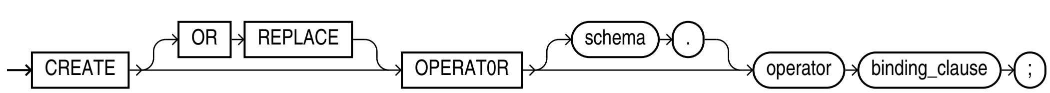
> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* and *Oracle Database Concepts* for a discussion of these dependencies and of operators in general

Prerequisites

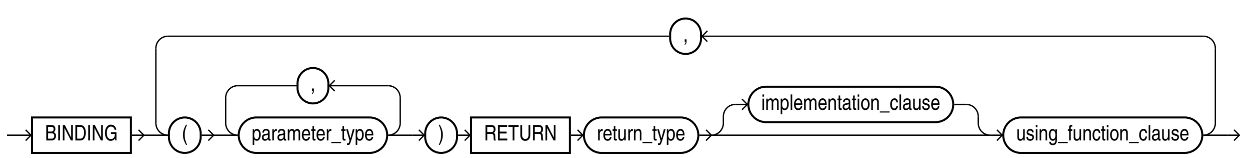
To create an operator in your own schema, you must have the `CREATE OPERATOR` system privilege. To create an operator in another schema, you must have the `CREATE ANY OPERATOR` system privilege. In either case, you must also have the `EXECUTE` object privilege on the functions and operators referenced.

Syntax

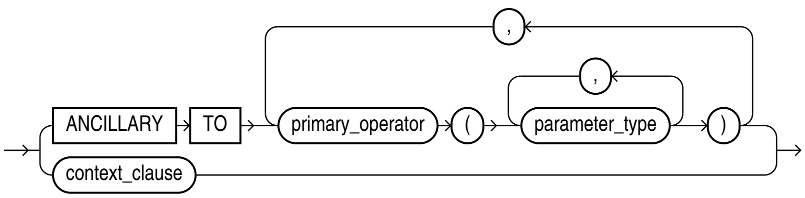
***create_operator*::=**



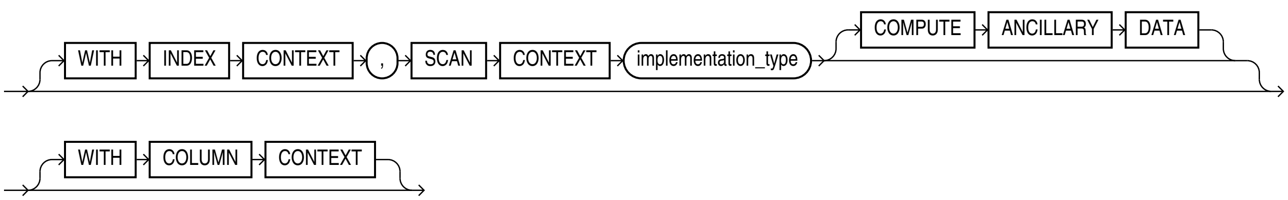
***binding_clause*::=**



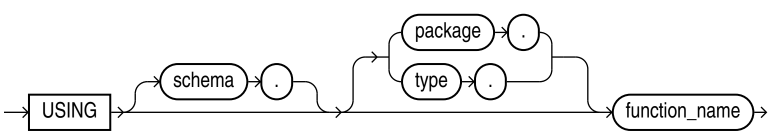
***implementation_clause*::=**



***context_clause*::=**



***using_function_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to replace the definition of the operator schema object.

Restriction on Replacing an Operator

You can replace the definition only if the operator has no dependent objects, such as indextypes supporting the operator.

schema

Specify the schema containing the operator. If you omit *`schema`*, then the database creates the operator in your own schema.

operator

Specify the name of the operator to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

binding_clause

Use the *binding_clause* to specify one or more parameter data types (*parameter_type*) for binding the operator to a function. The signature of each binding—the sequence of the data types of the arguments to the corresponding function—must be unique according to the rules of overloading.

The *parameter_type* can itself be an object type. If it is, then you can optionally qualify it with its schema.

Restriction on Binding Operators

You cannot specify a *parameter_type* of REF, LONG, or LONG RAW.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about overloading

RETURN Clause

Specify the return data type for the binding.

The *return_type* can itself be an object type. If so, then you can optionally qualify it with its schema.

Restriction on Binding Return Data Type

You cannot specify a *return_type* of REF, LONG, or LONG RAW.

implementation_clause

Use this clause to describe the implementation of the binding.

ANCILLARY TO Clause

Use the ANCILLARY TO clause to indicate that the operator binding is ancillary to the specified primary operator binding (*primary_operator*). If you specify this clause, then do not specify a previous binding with just one number parameter.

context_clause

Use the *context_clause* to describe the functional implementation of a binding that is not ancillary to a primary operator binding.

WITH INDEX CONTEXT, SCAN CONTEXT

Use this clause to indicate that the functional evaluation of the operator uses the index and a scan context that is specified by the implementation type.

COMPUTE ANCILLARY DATA

Specify COMPUTE ANCILLARY DATA to indicate that the operator binding computes ancillary data.

WITH COLUMN CONTEXT

Specify `WITH COLUMN CONTEXT` to indicate that Oracle Database should pass the column information to the functional implementation for the operator.

If you specify this clause, then the signature of the function implemented must include one extra `ODCIFuncCallInfo` structure.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for instructions on using the `ODCIFuncCallInfo` routine

using_function_clause

The *`using_function_clause`* lets you specify the function that provides the implementation for the binding. The *`function_name`* can be a standalone function, packaged function, type method, or a synonym for any of these.

If the function is subsequently dropped, then the database marks all dependent objects `INVALID`, including the operator. However, if you then subsequently issue an `ALTER OPERATOR` ... `DROP BINDING` statement to drop the binding, then subsequent queries and DML will revalidate the dependent objects.

Examples

Creating User-Defined Operators: Example

This example creates a very simple functional implementation of equality and then creates an operator that uses the function. For a more complete set of examples, see *Oracle Database Data Cartridge Developer's Guide*.

```
CREATE FUNCTION eq_f(a VARCHAR2, b VARCHAR2) RETURN NUMBER AS
BEGIN
   IF a = b THEN RETURN 1;
   ELSE RETURN 0;
   END IF;
END;
/

CREATE OPERATOR eq_op
   BINDING (VARCHAR2, VARCHAR2)
   RETURN NUMBER
   USING eq_f;
```

CREATE OUTLINE

Purpose

> **Note:**
>
> Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. SQL plan management creates SQL plan baselines, which offer superior SQL performance stability compared with stored outlines.
>
> You can migrate existing stored outlines to SQL plan baselines by using the `MIGRATE_STORED_OUTLINE` function of the `DBMS_SPM` package or Enterprise Manager Cloud Control. When the migration is complete, the stored outlines are marked as migrated and can be removed. You can drop all migrated stored outlines on your system by using the `DROP_MIGRATED_STORED_OUTLINE` function of the `DBMS_SPM` package.
>
> See Also: *Oracle Database SQL Tuning Guide* for more information about SQL plan management and *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SPM` package

Use the `CREATE OUTLINE` statement to create a **stored outline**, which is a set of attributes used by the optimizer to generate an execution plan. You can then instruct the optimizer to use a set of outlines to influence the generation of execution plans whenever a particular SQL statement is issued, regardless of changes in factors that can affect optimization. You can also modify an outline so that it takes into account changes in these factors.

> **Note:**
>
> The SQL statement you want to affect must be an exact string match of the statement specified when creating the outline.

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* for information on execution plans
> - ALTER OUTLINE for information on modifying an outline
> - ALTER SESSION and ALTER SYSTEM for information on the `USE_STORED_OUTLINES` and `USE_PRIVATE_OUTLINES` parameters

Prerequisites

To create a public or private outline, you must have the `CREATE ANY OUTLINE` system privilege.

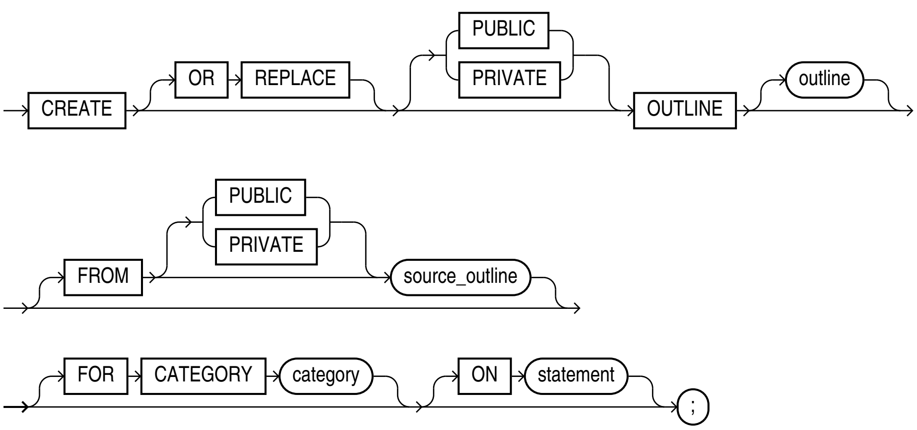
If you are creating a clone outline from a source outline, then you must also have the `SELECT_CATALOG_ROLE` role.

You can enable or disable the use of stored outlines dynamically for an individual session or for the system:

- Enable the `USE_STORED_OUTLINES` parameter to use public outlines.
- Enable the `USE_PRIVATE_OUTLINES` parameter to use private stored outlines.

Syntax

***create_outline*::=**



> **Note:**
>
> None of the clauses after *`outline`* are required. However, you must specify at least one clause after *`outline`*, and it must be either the `FROM` clause or the `ON` clause.

Semantics

OR REPLACE

Specify `OR REPLACE` to replace an existing outline with a new outline of the same name.

PUBLIC | PRIVATE

Specify `PUBLIC` if you are creating an outline for use by `PUBLIC`. This is the default.

Specify `PRIVATE` to create an outline for private use by the current session only. The data of this outline is stored in the current schema.

outline

Specify the unique name to be assigned to the stored outline. The name must satisfy the requirements listed in "Database Object Naming Rules ". If you do not specify *outline*, then the database generates an outline name.

> **See Also:**
>
> "Creating an Outline: Example"

FROM *source_outline* Clause

Use the FROM clause to create a new outline by copying an existing one. By default, Oracle Database looks for *source_category* in the public area. If you specify PRIVATE, then the database looks for the outline in the current schema.

Restriction on Copying an Outline

If you specify the FROM clause, then you cannot specify the ON clause.

> **See Also:**
>
> "Creating a Private Clone Outline: Example" and "Publicizing a Private Outline to the Public Area: Example"

FOR CATEGORY Clause

Specify an optional name used to group stored outlines. For example, you could specify a category of outlines for end-of-week use and another for end-of-quarter use. If you do not specify *category*, then the outline is stored in the DEFAULT category.

ON Clause

Specify the SQL statement for which the database will create an outline when the statement is compiled. This clause is optional only if you are creating a copy of an existing outline using the FROM clause.

You can specify any one of the following statements: SELECT, DELETE, UPDATE, INSERT ... SELECT, CREATE TABLE ... AS SELECT.

Restrictions on the ON Clause

This clause is subject to the following restrictions:

- If you specify the ON clause, then you cannot specify the FROM clause.
- You cannot create an outline on a multitable INSERT statement.
- The SQL statement in the ON clause cannot include any DML operation on a remote object.

> **Note:**
>
> In subsequent statements, you can specify additional outlines for the same SQL statement, but each outline for the same statement must specify a different category in the `CATEGORY` clause.

Examples

Creating an Outline: Example

The following statement creates a stored outline by compiling the `ON` statement. The outline is called `salaries` and is stored in the category `special`.

```
CREATE OUTLINE salaries FOR CATEGORY special
   ON SELECT last_name, salary FROM employees;
```

When this same `SELECT` statement is subsequently compiled, if the `USE_STORED_OUTLINES` parameter is set to `special`, the database generates the same execution plan as was generated when the outline `salaries` was created.

Creating a Private Clone Outline: Example

The following statement creates a stored private outline `my_salaries` based on the public category `salaries` created in the preceding example.

```
CREATE OR REPLACE PRIVATE OUTLINE my_salaries
   FROM salaries;
```

Publicizing a Private Outline to the Public Area: Example

The following statement copies back (publicizes) a private outline to the public area after private editing:

```
CREATE OR REPLACE OUTLINE public_salaries
   FROM PRIVATE my_salaries;
```

CREATE PACKAGE

Purpose

Packages are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE PACKAGE` statement to create the specification for a stored **package**, which is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The **package specification** declares these objects. The **package body**, specified subsequently, defines these objects.

> **See Also:**
>
> - CREATE PACKAGE BODY for information on specifying the implementation of the package
> - CREATE FUNCTION and CREATE PROCEDURE for information on creating standalone functions and procedures
> - ALTER PACKAGE and DROP PACKAGE for information on modifying and dropping a package
> - *Oracle Database Development Guide* and *Oracle Database PL/SQL Packages and Types Reference* for detailed discussions of packages and how to use them

Prerequisites

To create or replace a package in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

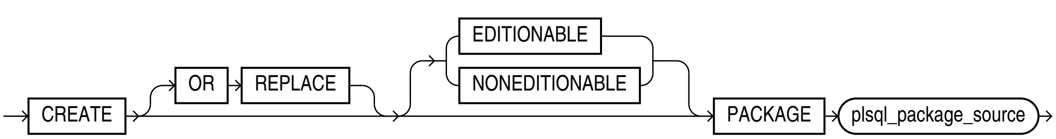
To embed a `CREATE PACKAGE` statement inside an Oracle Database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information

Syntax

Packages are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_package*::=**



(*`plsql_package_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the package specification if it already exists. Use this clause to change the specification of an existing package without dropping, re-creating, and regranting object privileges previously granted on the package. If you change a package specification, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, then the database marks the indexes `DISABLED`.

> **See Also:**
>
> `ALTER PACKAGE` for information on recompiling package specifications

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the package is an editioned or noneditioned object if editioning is enabled for the schema object type `PACKAGE` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_package_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_package_source`*, including examples.

CREATE PACKAGE BODY

Purpose

Package bodies are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE PACKAGE BODY` statement to create the body of a stored **package**, which is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The **package body** defines these objects. The **package specification**, defined in an earlier `CREATE PACKAGE` statement, declares these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

> **See Also:**
>
> - CREATE FUNCTION and CREATE PROCEDURE for information on creating standalone functions and procedures
> - CREATE PACKAGE for a discussion of packages, including how to create packages
> - ALTER PACKAGE for information on modifying a package
> - DROP PACKAGE for information on removing a package from the database

Prerequisites

To create or replace a package in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a package in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege. In both cases, the package body must be created in the same schema as the package.

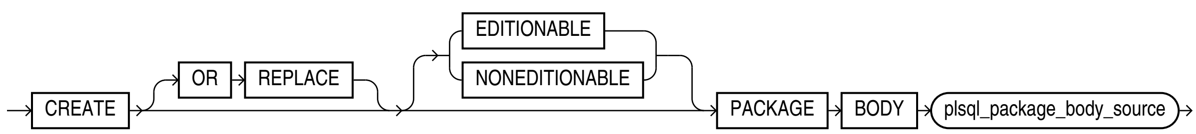
To embed a `CREATE PACKAGE BODY` statement inside an Oracle Database precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference*

Syntax

Package bodies are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

create_package_body::=



(*`plsql_package_body_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the package body if it already exists. Use this clause to change the body of an existing package without dropping, re-creating, and regranting object privileges previously granted on it. If you change a package body, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

> **See Also:**
>
> ALTER PACKAGE for information on recompiling package bodies

[EDITIONABLE | NONEDITIONABLE]

If you do not specify this clause, then the package body inherits `EDITIONABLE` or `NONEDITIONABLE` from the package specification. If you do specify this clause, then it must match that of the package specification.

plsql_package_body_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_package_body_source`*.

CREATE PFILE

Purpose

Use the `CREATE PFILE` statement to export either a binary server parameter file or the current In-Memory parameter settings into a text initialization parameter file. Creating a text parameter file is a convenient way to get a listing of the current parameter settings being used by the database, and it lets you edit the file easily in a text editor and then convert it back into a server parameter file using the `CREATE SPFILE` statement.

Upon successful execution of this statement, Oracle Database creates a text parameter file on the server. In an Oracle Real Application Clusters environment, it will contain all parameter settings of all instances. It will also contain any comments that appeared on the same line with a parameter setting in the server parameter file.

Note on Creating Text Parameter Files in a CDB

When you create a text parameter file in a multitenant container database (CDB), the current container can be the root or a PDB.

- If the current container is the root, then the database creates a text file that contains the parameter settings for the root.
- If the current container is a PDB, then the database creates a text file that contains the parameter settings for the PDB. In this case you must specify a *`pfile_name`*.

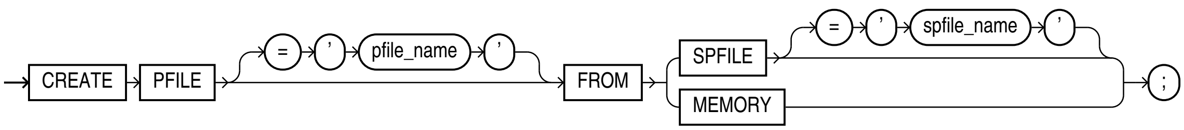
> **See Also:**
>
> - CREATE SPFILE for information on server parameter files
> - *Oracle Database Administrator's Guide* for additional information on text initialization parameter files and binary server parameter files
> - *Oracle Real Application Clusters Administration and Deployment Guide* for information on using server parameter files in an Oracle Real Application Clusters environment

Prerequisites

You must have the `SYSBACKUP`, `SYSDBA`, `SYSDG`, or `SYSOPER` system privilege to execute this statement. You can execute this statement either before or after instance startup.

Syntax

***create_pfile*::=**



Semantics

pfile_name

Specify the name of the text parameter file you want to create. If you do not specify *`pfile_name`*, then Oracle Database uses the platform-specific default initialization parameter file name. *`pfile_name`* can include a path prefix. If you do not specify such a path prefix, then the database adds the path prefix for the default storage location, which is platform dependent.

spfile_name

Specify the name of the binary server parameter from which you want to create a text file.

- If you specify *`spfile_name`*, then the file must exist on the server. If the file does not reside in the default directory for server parameter files on your operating system, then you must specify the full path.
- If you do not specify *`spfile_name`*, then the database uses the spfile that is currently associated with the instance, usually the one that was used a startup. If no spfile is associated with the instance, then the database looks for the platform-specific default server parameter file name. If that file does not exist, then the database returns an error.

> **See Also:**
>
> the appropriate operating-system-specific documentation for default parameter file names

MEMORY

Specify `MEMORY` to create a pfile using the current system-wide parameter settings. In an Oracle RAC environment, the created file will contain the parameter settings from each instance.

Examples

Creating a Parameter File: Example

The following example creates a text parameter file `my_init.ora` from a binary server parameter file `s_params.ora`:

```
CREATE PFILE = 'my_init.ora' FROM SPFILE = 's_params.ora';
```

> **Note:**
>
> Typically you will need to specify the full path and filename for parameter files on your operating system. Refer to your Oracle operating system documentation for path information.

CREATE PLUGGABLE DATABASE

Purpose

Use the `CREATE PLUGGABLE DATABASE` statement to create a pluggable database (PDB).

This statement enables you to perform the following tasks:

- Create a PDB by using the seed as a template

  Use the *`create_pdb_from_seed`* clause to create a PDB by using the seed in the multitenant container database (CDB) as a template. The files associated with the seed are copied to a new location and the copied files are then associated with the new PDB.

- Create a PDB by cloning an existing PDB or non-CDB

  Use the *`create_pdb_clone`* clause to create a PDB by copying an existing PDB or non-CDB and then plugging the copy into the CDB. The files associated with the existing PDB or non-CDB are copied to a new location and the copied files are associated with the new PDB.

- Create a PDB by plugging an unplugged PDB or a non-CDB into a CDB

  Use the *`create_pdb_from_xml`* clause to plug an unplugged PDB or a non-CDB into a CDB, using an XML metadata file.

- Create a proxy PDB by referencing another PDB. A proxy PDB provides fully functional access to the referenced PDB.

  Use the *`create_pdb_clone`* clause and specify `AS PROXY FROM` to create a proxy PDB.

- Create an application container, application seed, or application PDB

  Use the *`create_pdb_from_seed`*, *`create_pdb_clone`*, or *`create_pdb_from_xml clause`*. To create an application container, you must specify the `AS APPLICATION CONTAINER` clause. To create an application seed, you must specify the `AS SEED` clause.

> **Note:**
>
> When a new PDB is established in a CDB, it is possible that the name of a service offered by the new PDB will collide with an existing service name. The namespace in which a collision can occur is that of the listener that gives access to the CDB. Within that namespace, collisions are possible among the names of non-CDB's default services, CDB's default services, PDB's default services, and user-defined services. For example, if two or more CDBs on the same computer system use the same listener, and the newly established PDB has the same service name as another PDB in these CDBs, then a collision occurs.
>
> When you create a PDB, you can specify new names for any potential colliding service names. See the clause *service_name_convert*. If you discover a service name collision after a PDB is created, you must not attempt to operate the PDB that causes a collision with an existing service name. If the colliding name is that of the PDB's default service, then you must rename the PDB. If the colliding name is that of a user-created service within the PDB, then you must drop that service and create one in its place, with a non-colliding name, that has the same purpose and properties.

> **See Also:**
>
> - *Oracle Database Concepts* for general information on PDBs
> - *Oracle Database Administrator's Guide* for information on managing PDBs
> - ALTER PLUGGABLE DATABASE and DROP PLUGGABLE DATABASE for information on modifying and dropping PDBs

Prerequisites

You must be connected to a CDB. The CDB must be open and in `READ WRITE` mode.

To create a PDB or an application container, the current container must be the root and you must have the `CREATE PLUGGABLE DATABASE` system privilege, granted commonly.

To create an application seed or an application PDB, the current container must be an application root, the application container must be open and in `READ WRITE` mode, and you must have the `CREATE PLUGGABLE DATABASE` system privilege, either granted commonly or granted locally in that application container.

To specify the *`create_pdb_clone`* clause:

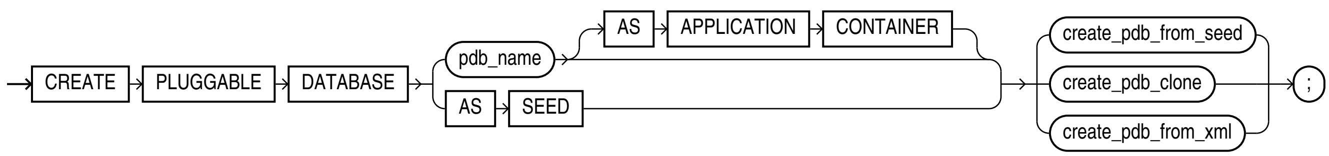
- If *`src_pdb_name`* refers to a PDB in the same CDB, then you must have the `CREATE PLUGGABLE DATABASE` system privilege in the root of the CDB in which the new PDB will be created and in the PDB being cloned.
- If *`src_pdb_name`* refers to a PDB in a remote database or a non-CDB, then you must have the `CREATE PLUGGABLE DATABASE` system privilege in the root of the CDB in which the new PDB will be created and the remote user must have the `CREATE`

`PLUGGABLE DATABASE` system privilege in the PDB or non-CDB to which *`src_pdb_name`* refers.

See *Oracle Multitenant Administrator’s Guide*for more information on the prerequisites to PDB creation.

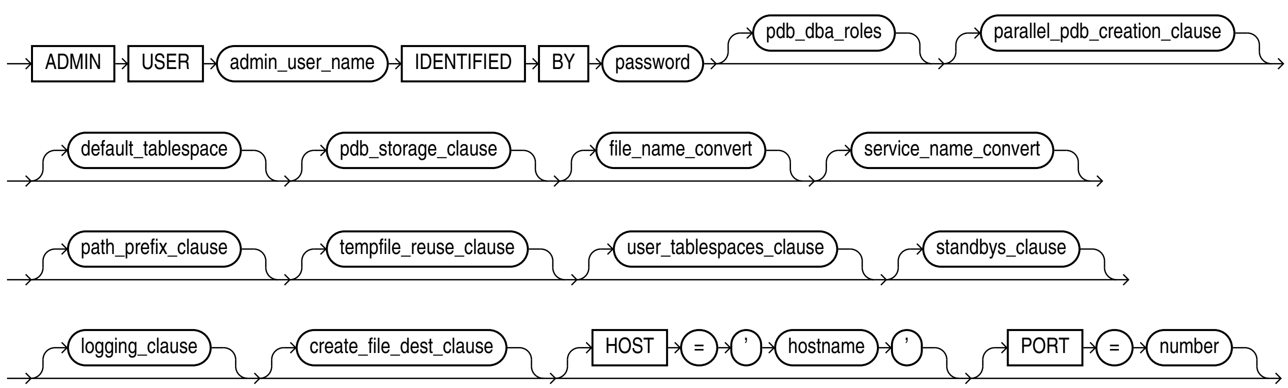
Syntax

***create_pluggable_database*::=**



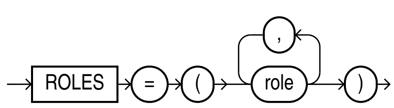
(*create_pdb_from_seed*::=, *create_pdb_clone*::=, *create_pdb_from_xml*::=)

***create_pdb_from_seed*::=**

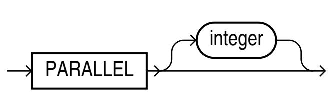


(*pdb_dba_roles*::=, *parallel_pdb_creation_clause*::=, *default_tablespace*::=, *file_name_convert*::=, *service_name_convert*::=, *pdb_storage_clause*::=, *path_prefix_clause*::=, *tempfile_reuse_clause*::=, *user_tablespaces_clause*::=, *standbys_clause*::=, *logging_clause*::=, *create_file_dest_clause*::=)

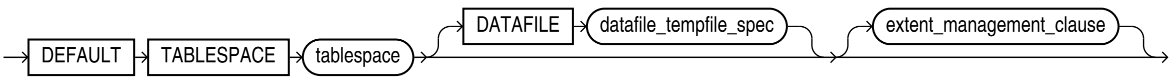
***pdb_dba_roles*::=**



***parallel_pdb_creation_clause*::=**

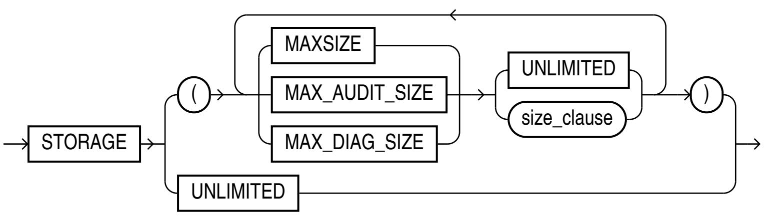


***default_tablespace*::=**



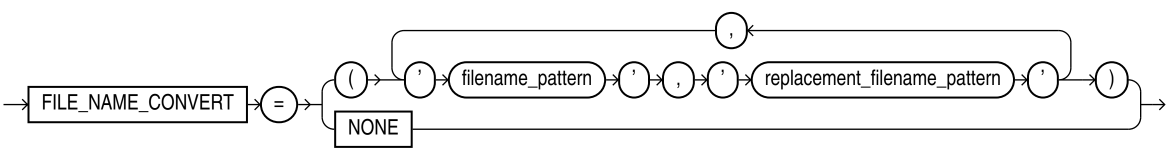
(*datafile_tempfile_spec*::=, *extent_management_clause*::=)

***pdb_storage_clause*::=**

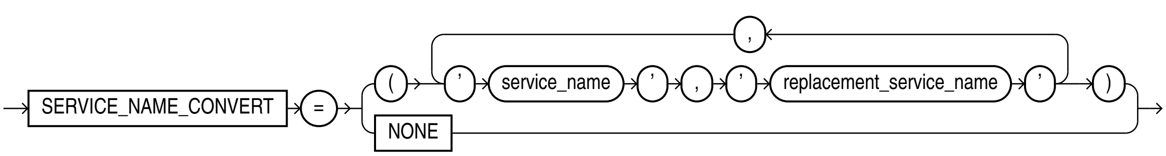


(*size_clause*::=)

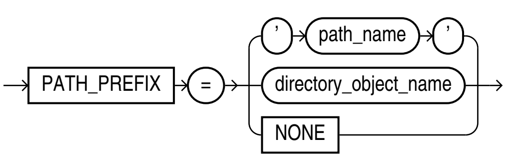
***file_name_convert*::=**



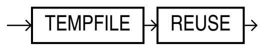
***service_name_convert*::=**



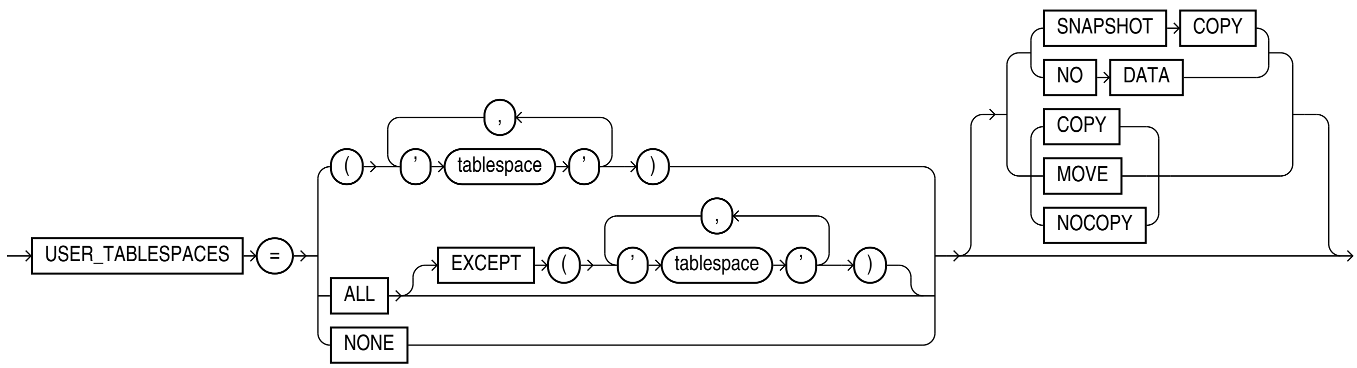
***path_prefix_clause*::=**



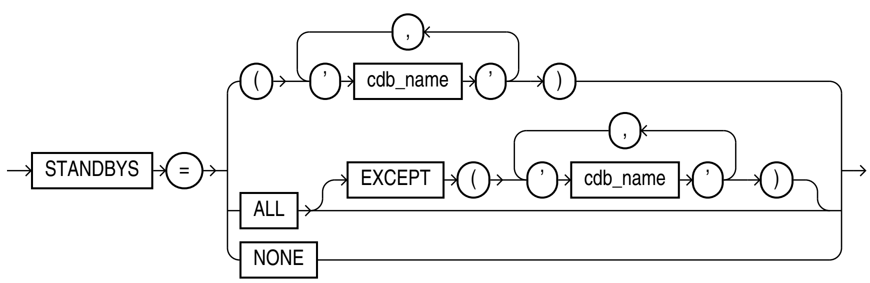
***tempfile_reuse_clause*::=**



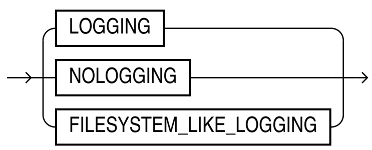
***user_tablespaces_clause*::=**



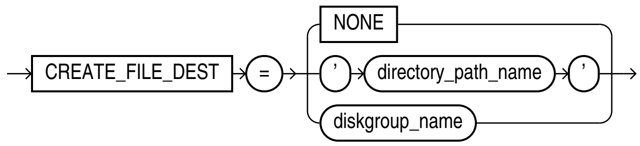
***standbys_clause*::=**



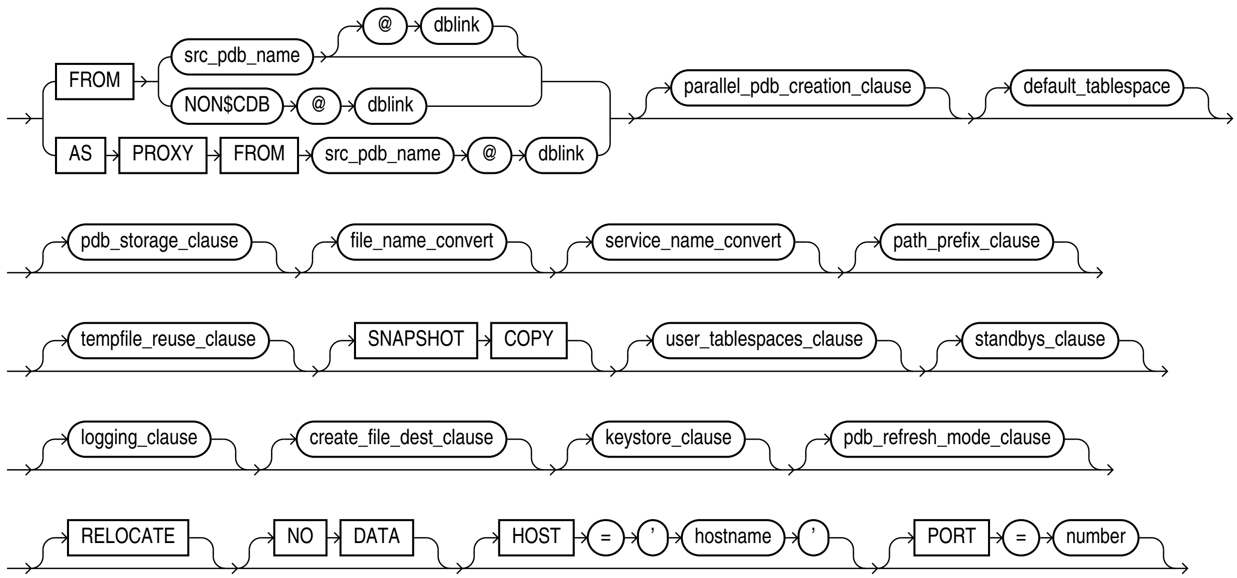
***logging_clause*::=**



***create_file_dest_clause*::=**

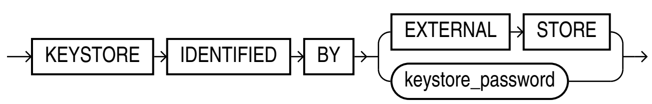


***create_pdb_clone*::=**

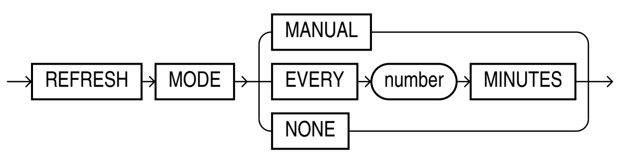


(*parallel_pdb_creation_clause*::=, *default_tablespace*::=, *pdb_storage_clause*::=, *file_name_convert*::=, *service_name_convert*::=, *path_prefix_clause*::=, *tempfile_reuse_clause*::=, *user_tablespaces_clause*::=, *standbys_clause*::=, *logging_clause*::=, *create_file_dest_clause*::=, *keystore_clause*::=, *pdb_refresh_mode_clause*::=)

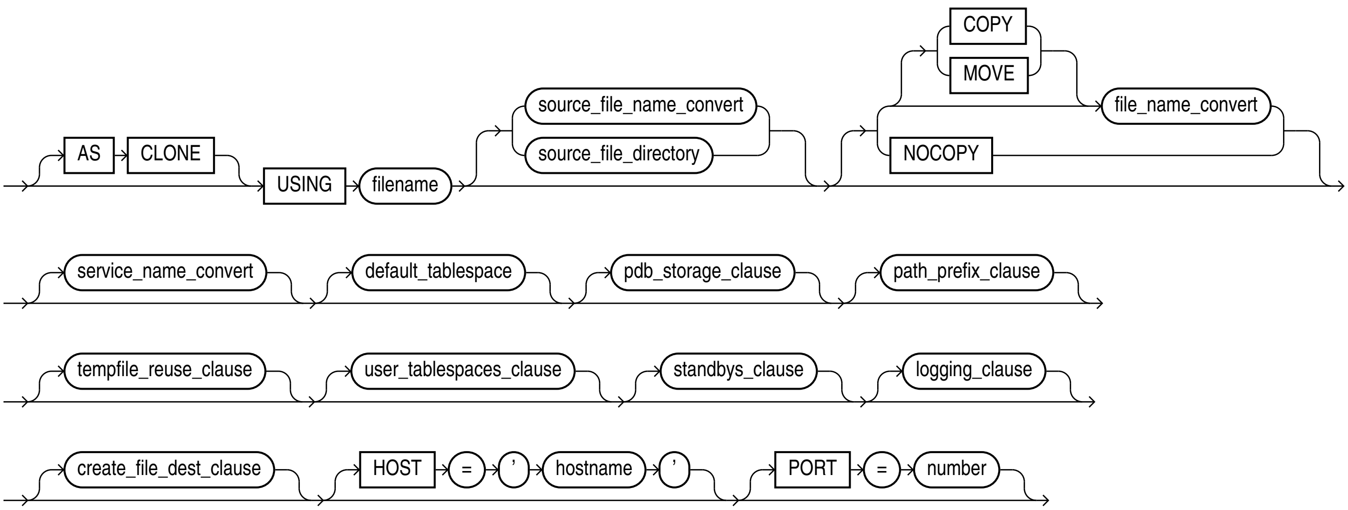
***keystore_clause*::=**



***pdb_refresh_mode_clause*::=**

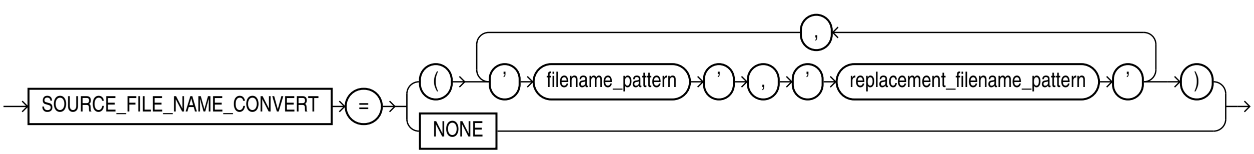


***create_pdb_from_xml*::=**

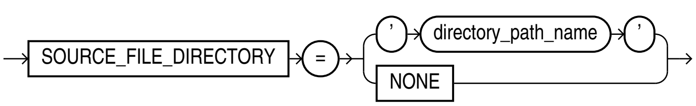


(*source_file_name_convert*::=, *source_file_directory*::=, *file_name_convert*::=, *service_name_convert*::=, *default_tablespace*::=, *pdb_storage_clause*::=, *path_prefix_clause*::=, *tempfile_reuse_clause*::=, *user_tablespaces_clause*::=, *standbys_clause*::=, *logging_clause*::=, *create_file_dest_clause*::=)

***source_file_name_convert*::=**



***source_file_directory*::=**



Semantics

pdb_name

Specify the name of the PDB to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". The first character of a PDB name must be alphanumeric and the remaining characters can be alphanumeric or the underscore character (_).

The PDB name must be unique in the CDB, and it must be unique within the scope of all the CDBs whose instances are reached through a specific listener.

AS APPLICATION CONTAINER

Specify this clause to create an application container.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for the complete steps for creating an application container

AS SEED

Specify this clause to create an application seed. The database assigns the seed a name of the form `application_container_name$SEED`.

An application container can have at most one application seed. The application seed is optional, but, if it exists, you can use it to create application PDBs quickly that match the requirements of the application container. An application seed enables instant provisioning of application PDBs that are created from it.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for the complete steps for creating an application seed

create_pdb_from_seed

This clause enables you to create a PDB by using the seed in the CDB as a template.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for the complete steps for creating a PDB using the seed

ADMIN USER

Use this clause to create an administrative user who can be granted the privileges required to perform administrative tasks on the PDB. For `admin_user_name`, specify name of the user to be created. Use the `IDENTIFIED BY` clause to specify the password for `admin_user_name`. Oracle Database creates a local user in the PDB and grants the `PDB_DBA` local role to that user.

pdb_dba_roles

This clause lets you grant one or more roles to the `PDB_DBA` role. Use this clause to grant roles that have the privileges required by the administrative user of the PDB. For `role`, specify a predefined role. For a list of predefined roles, refer to *Oracle Database Security Guide*.

You can also use the GRANT statement to grant roles to the PDB_DBA role after the PDB has been created. Until you have granted the appropriate privileges to the PDB_DBA role, the SYS and SYSTEM users can perform administrative tasks on a PDB.

parallel_pdb_creation_clause

This clause instructs the CDB to use parallel execution servers to copy the new PDB's data files to a new location. This may result in faster creation of the PDB.

PARALLEL

If you specify PARALLEL, then the CDB automatically chooses the number of parallel execution servers to use. This is the default if the COMPATIBLE initialization parameter is set to 12.2 or higher.

PARALLEL *integer*

Use integer to specify the number of parallel execution servers to use. The CDB can ignore this setting, depending on the current database load and the number of available parallel execution servers. If you specify a value of 0 or 1, then the CDB does not parallelize the creation of the PDB. This can result in a longer PDB creation time.

default_tablespace

If you specify this clause, then Oracle Database creates a smallfile tablespace and sets it as the default permanent tablespace for the PDB. Oracle Database will assign the default tablespace to any non-SYSTEM user for whom a different permanent tablespace is not specified. The *default_tablespace* clause has the same semantics that it has for the CREATE DATABASE statement. For full information, refer to *default_tablespace* in the documentation on CREATE DATABASE.

pdb_storage_clause

Use this clause to specify storage limits for the PDB.

- Use MAXSIZE to limit the amount of storage that can be used by all tablespaces in the PDB to the value specified with *size_clause*. This limit includes the size of data files and temporary files for tablespaces belonging to the PDB. Specify MAXSIZE UNLIMITED to enforce no limit.
- Use MAX_AUDIT_SIZE to limit the amount of storage that can be used by unified audit OS spillover (.bin format) files in the PDB to the value specified with *size_clause*. Specify MAX_AUDIT_SIZE UNLIMITED to enforce no limit.
- Use MAX_DIAG_SIZE to limit the amount of storage for diagnostics (trace files and incident dumps) in the Automatic Diagnostic Repository (ADR) that can be used by the PDB to the value specified with *size_clause*. Specify MAX_DIAG_SIZE UNLIMITED to enforce no limit.

If you omit this clause, or specify STORAGE UNLIMITED, then there are no storage limits for the PDB. This is equivalent to specifying STORAGE (MAXSIZE UNLIMITED MAX_AUDIT_SIZE UNLIMITED MAX_DIAG_SIZE UNLIMITED).

file_name_convert

Use this clause to determine how the database generates the names of files (such as data files and wallet files) for the PDB.

- For *filename_pattern*, specify a string found in names of files associated with the seed (when creating a PDB by using the seed), associated with the source PDB (when cloning a PDB), or listed in the XML file (when plugging a PDB into a CDB).
- For *replacement_filename_pattern*, specify a replacement string.

Oracle Database will replace *filename_pattern* with *replacement_filename_pattern* when generating the names of files associated with the new PDB.

File name patterns cannot match files or directories managed by Oracle Managed Files.

You can specify `FILE_NAME_CONVERT = NONE`, which is the same as omitting this clause. If you omit this clause, then the database first attempts to use Oracle Managed Files to generate file names. If you are not using Oracle Managed Files, then the database uses the `PDB_FILE_NAME_CONVERT` initialization parameter to generate file names. If this parameter is not set, then an error occurs.

service_name_convert

Use this clause to rename the user-defined services of the new PDB based on the service names of the source PDB. When the service name of a new PDB conflicts with an existing service name in the CDB, plug-in violations can result. This clause enables you to avoid these violations.

- For *service_name*, specify the name of a service found in the PDB seed (when creating a PDB in an application container by using the application seed) or in the source PDB (when cloning a PDB or plugging a PDB into a CDB).
- For *replacement_service_name*, specify the replacement name for the service.

Oracle Database will use the replacement service name for the service in the PDB being created.

You can specify `SERVICE_NAME_CONVERT = NONE`, which is the same as omitting this clause.

Restrictions on *service_name_convert*

The *service_name_convert* clause is subject to the following restrictions:

- You cannot change the name of the default service for a PDB. The default service has the same name as the PDB.
- You cannot specify this clause when you use the *create_pdb_from_seed* clause to create a PDB from the CDB seed, because the CDB seed does not have user-defined services. You can, however, specify this clause when you use the *create_pdb_from_seed* clause to create an application PDB from the application seed.

path_prefix_clause

Use this clause to ensure that file paths for directory objects associated with the PDB are restricted to the specified directory or its subdirectories. This clause also ensures that the following files associated with the PDB are restricted to the specified directory: the Oracle XML repository for the PDB, files created with a `CREATE PFILE` statement, and the export directory for Oracle wallets. You cannot modify the setting of this clause after you create the PDB. This clause does not affect files created by Oracle Managed Files.

- For *path_name*, specify the absolute path name of an operating system directory. The single quotation marks are required, with the result that the path name is case sensitive. Oracle Database uses *path_name* as a prefix for all file paths associated with the PDB.

  Be sure to specify *path_name* so that the resulting path name will be properly formed when relative paths are appended to it. For example, on UNIX systems, be sure to end *path_name* with a forward slash (/), such as:

  ```
  PATH_PREFIX = '/disk1/oracle/dba/salespdb/'
  ```

- For *directory_object_name*, specify the name of a directory object that exists in the CDB root (CDB$ROOT). The directory object points to the absolute path to be used for PATH_PREFIX.
- If you specify PATH_PREFIX = NONE, then the relative paths for directory objects associated with the PDB are treated as absolute paths and are not restricted to a particular directory.

Omitting the *path_prefix_clause* is equivalent to specifying PATH_PREFIX = NONE.

After the *path_prefix_clause* is specified for a PDB, existing directory objects might not work as expected, since the PATH_PREFIX string is always added as a prefix to all local directory objects in the PDB. The *path_prefix_clause* only applies to user-created directory objects. It does not apply to Oracle-supplied directory objects.

tempfile_reuse_clause

When you create a PDB, Oracle Database associates temp files with the new PDB. Depending on how you create the PDB, the temp files may already exist and may have been previously used.

Specify TEMPFILE REUSE to instruct the database to format and reuse a temp file associated with the new PDB if it already exists. If you specify this clause and a temp file does not exist, then the database creates the temp file.

If you do not specify TEMPFILE REUSE and a temp file to be associated with the new PDB already exists, then the database returns an error and does not create the PDB.

user_tablespaces_clause

This clause lets you specify the tablespaces to be made available in the new PDB. The SYSTEM, SYSAUX, and TEMP tablespaces are available in all PDBs and cannot be specified in this clause.

You can use this clause to separate the data for multiple schemas into different PDBs. For example, when you move a non-CDB to a PDB, and the non-CDB had a number of schemas that each supported a different application, you can use this clause to separate the data belonging to each schema into a separate PDB, assuming that each schema used a separate tablespace in the non-CDB.

- Specify *tablespace* to make the tablespace available in the new PDB. You can specify more than one tablespace in a comma-separated list.
- Specify ALL to make all tablespaces available in the new PDB. This is the default.
- Specify ALL EXCEPT to make all tablespaces available in the new PDB, except the specified tablespaces.
- Specify NONE to make only the SYSTEM, SYSAUX, and TEMP tablespaces available in the new PDB.

When the compatibility level of the CDB is `12.2` or higher, the tablespaces that are excluded by this clause are created offline in the new PDB, and they have no data files associated with them. When the compatibility level of the CDB is lower than `12.2`, the tablespaces that are excluded by this clause are offline in the new PDB, and all data files that belong to these tablespaces are unnamed and offline.

{ SNAPSHOT COPY | NO DATA }

These clauses apply only when cloning a PDB with the *create_pdb_clone* clause. They do not apply when cloning a non-CDB. By default, the database creates each tablespace to be made available in the new PDB according to the settings specified for cloning the PDB. These clauses allow you to override those settings as follows:

- `SNAPSHOT COPY` - Clone the tablespace using storage snapshots.
- `NO DATA` - Clone the data model definition of the tablespace, but not the tablespace's data.

{ COPY | MOVE | NOCOPY }

These clauses apply when you plug in a PDB with the *create_pdb_from_xml* clause. By default, the database creates each tablespace to be made available in the new PDB according to the settings specified for plugging in the PDB. These clauses allow you to override those settings as follows:

- `COPY` - Copy the tablespace files to the new location.
- `MOVE` - Move the tablespace files to the new location.
- `NOCOPY` - Do not copy or move the tablespace files to the new location.

standbys_clause

Use this clause to specify whether the new PDB is included in one or more standby CDBs. If you include a PDB in a standby CDB, then during standby recovery the standby CDB will search for the data files for the PDB. If the data files are not found, then standby recovery will stop and you must copy the data files to the correct location before you can restart recovery.

- Specify *cdb_name* to include the new PDB in the specified standby CDB. You can specify more than one standby CDB name in a comma-separated list.
- Specify `ALL` to include the new PDB in all standby CDBs. This is the default.
- Specify `ALL EXCEPT` to include the new PDB in all standby CDBs, except the specified standby CDBs.
- Specify `NONE` to exclude the new PDB from all standby CDBs. When a PDB is excluded from all standby CDBs, the PDB's data files are unnamed and marked offline on all of the standby CDBs. Standby recovery will not stop if the data files for the PDB are not found on the standby. If you instantiate a new standby CDB after the PDB is created, then you must explicitly disable the PDB for recovery on the new standby CDB.

You can enable a PDB on a standby CDB after it was excluded on that standby CDB by copying the data files to the correct location, bringing the PDB online, and marking it as enabled for recovery.

logging_clause

Use this clause to specify the default logging attribute for tablespaces created within the PDB. The logging attribute controls whether certain DML operations are logged in the redo log file (`LOGGING`) or not (`NOLOGGING`).The default is `LOGGING`.

When creating a tablespace, you can override the default logging attribute by specifying the *logging_clause* of the `CREATE TABLESPACE` statement.

Refer to *logging_clause* for a full description of this clause.

create_file_dest_clause

By default, a newly created PDB inherits its Oracle Managed Files settings from the root. If the root uses Oracle Managed Files, then the PDB also uses Oracle Managed Files. The PDB shares the same base file system directory for Oracle Managed Files with the root and has its own subdirectory named with the GUID of the PDB. If the root does not use Oracle Managed Files, then the PDB also does not use Oracle Managed Files.

This clause lets you override the default behavior. You can enable or disable Oracle Managed Files for the PDB and you specify a different base file system directory or Oracle ASM disk group for the PDB's files.

- Specify `NONE` to disable Oracle Managed Files for the PDB.
- Specify either *`directory_path_name`* or *`diskgroup_name`* to enable Oracle Managed Files for the PDB.

  Specify *`directory_path_name`* to designate the base file system directory for the PDB's files. Specify the full path name of the operating system directory. The directory must exist and Oracle processes must have appropriate permissions on the directory. The single quotation marks are required, with the result that the path name is case sensitive.

  Specify *`diskgroup_name`* to designate the default Oracle ASM disk group for the PDB's files.

If you specify a value other than `NONE`, then the database implicitly sets the `DB_CREATE_FILE_DEST` initialization parameter with `SCOPE=SPFILE` in the PDB.

HOST and PORT

These clauses are useful only if you are creating a PDB that you plan to reference from a proxy PDB. This type of PDB is called a referenced PDB.

When creating a referenced PDB:

- If the name of the listener is different from the host name of the PDB, then you must specify the `HOST` clause. For *`hostname`*, specify the fully qualified domain name of the listener. Enclose *`hostname`* in single quotation marks. For example: `'myhost.example.com'`.

  In an Oracle Real Application Clusters (Oracle RAC) environment, you can specify for *`hostname`* any of the hosts for the PDB.
- If the port number of the listener is not 1521, then you must specify the `PORT` clause. For *`number`*, specify the port number for the listener.

A proxy PDB uses a database link to establish communication with its referenced PDB. After communication is established, the proxy PDB communicates directly with the referenced PDB without using a database link. The host name and port number of the listener for the referenced PDB must be correct for the proxy PDB to function properly.

> **See Also:**
>
> The clause AS PROXY FROM of `create_pdb_clone` for information on creating a proxy PDB

create_pdb_clone

This clause enables you to create a new PDB by cloning a source to a target PDB. The source can be a PDB in the local CDB, a PDB in a remote CDB, or a non-CDB. The target PDB is the clone of the source.

If the source is a PDB in the local CDB, then the source PDB can be plugged in or unplugged. If the source is a PDB in a remote CDB, then the source PDB must be plugged in.

If the source is a non-CDB or a PDB in a remote CDB, then the source and the CDB that contains the target PDB must meet the following requirements:

- They must have the same endian format.
- They must have compatible character sets and national character sets, which means:
  - Every character in the source character set is available in the local CDB character set.
  - Every character in the source character set has the same code point value in the local CDB character set.
- They must have the same set of database options installed.

Users in the PDB who used the default temporary tablespace of the source non-CDB or PDB use the default temporary tablespace of the new PDB. Users who used non-default temporary tablespaces in the non-CDB or PDB continue to use the same local temporary tablespaces in the new PDB.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for the complete steps for creating a PDB by cloning an existing PDB

FROM

Use this clause to specify the source PDB or non-CDB. The files associated with the source are copied to a new location and these copied files are then associated with the new PDB.

The source PDB or non-CDB cannot be closed. It can be open as follows:

- If the CDB that contains the source PDB (the source CDB) or the source non-CDB is in `ARCHIVELOG` mode and local undo mode, then the source PDB or the source non-CDB can be open in READ WRITE mode and fully functional during the cloning operation. This is called hot PDB cloning.

- If the source CDB or source non-CDB is not in ARCHIVELOG mode, then the source PDB or non-CDB must be open READ ONLY.

Specify the source PDB or non-CDB as follows:

- If the source is a PDB in the local CDB, then use *src_pdb_name* to specify the name of the source PDB. You cannot specify PDB$SEED for *src_pdb_name*. Instead, use the *create_pdb_from_seed* clause to create a PDB by using the seed as a template.
- If the source is a PDB in a remote CDB, then use *src_pdb_name* to specify the name of the source PDB and *dblink* to specify the name of the database link to use to connect to the remote CDB.
- If the source is a non-CDB, then specify NON$CDB@*dblink*, where *dblink* is the name of the database link to use to connect to the non-CDB.

AS PROXY FROM

Use this clause to create a proxy PDB by referencing a different PDB, which is referred to as the referenced PDB. The referenced PDB can be in the same CDB as the proxy PDB or in a different CDB. A local proxy PDB is in the same CDB as its referenced PDB, and a remote proxy PDB is in a different CDB than its referenced PDB.

For *src_pdb_name@dblink*, specify the referenced PDB.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for the complete steps for creating a proxy PDB

default_tablespace

Use this clause to specify a permanent default tablespace for the PDB. Oracle Database will assign the default tablespace to any non-SYSTEM user for whom a different permanent tablespace is not specified. The tablespace must already exist in the source PDB or non-CDB. Because the tablespace already exists, you cannot specify the DATAFILE clause or the *extent_management_clause* when creating a PDB with the *create_pdb_clone* clause.

pdb_storage_clause

Use this clause to specify storage limits for the new PDB. Refer to *pdb_storage_clause* for the full semantics of this clause.

file_name_convert

Use this clause to determine how the database generates the names of files for the new PDB. Refer to *file_name_convert* for the full semantics of this clause.

service_name_convert

Use this clause to determine how the database renames services for the new PDB. Refer to *service_name_convert*::= for the full semantics of this clause.

path_prefix_clause

Use this clause to ensure that all directory object paths associated with the PDB are restricted to the specified directory or its subdirectories. Refer to *path_prefix_clause* for the full semantics of this clause.

tempfile_reuse_clause

Specify `TEMPFILE REUSE` to instruct the database to format and reuse a temp file associated with the new PDB if it already exists. Refer to *tempfile_reuse_clause* for the full semantics of this clause.

SNAPSHOT COPY

You can specify `SNAPSHOT COPY` only when cloning a PDB. This clause is not supported when cloning a non-CDB. The source PDB can be in the local CDB or a remote CDB. The `SNAPSHOT COPY` clause instructs the database to clone the source PDB using storage snapshots. This reduces the time required to create the clone because the database does not need to make a complete copy of the source data files.

When you use the `SNAPSHOT COPY` clause to create a clone of a source PDB and the `CLONEDB` initialization parameter is set to `FALSE`, the underlying file system for the source PDB's files must support storage snapshots. Such file systems include Oracle Automatic Storage Management Cluster File System (Oracle ACFS) and Direct NFS Client storage.

When you use the `SNAPSHOT COPY` clause to create a clone of a source PDB and the `CLONEDB` initialization parameter is set to `TRUE`, the underlying file system for the source PDB's files can be any local file system, network file system (NFS), or clustered file system that has Direct NFS enabled. However, the source PDB must remain in open read-only mode as long as any clones exist.

Direct NFS Client enables an Oracle database to access network attached storage (NAS) devices directly, rather than using the operating system kernel NFS client. If the PDB files are stored on Direct NFS Client storage, then the following additional requirements must be met:

- The source PDB files must be located on an NFS volume.
- Storage credentials must be stored in a Transparent Data Encryption keystore.
- The storage user must have the privileges required to create and destroy snapshots on the volume that hosts the source PDB files.
- Credentials must be stored in the keystore using an `ADMINISTER KEY MANAGEMENT ADD SECRET` SQL statement.

When you use the `SNAPSHOT COPY` clause to create a clone of a source PDB, the following restrictions apply to the source PDB as long as any clones exist:

- It cannot be unplugged.
- It cannot be dropped.

PDB clones created using the `SNAPSHOT COPY` clause cannot be unplugged. They can only be dropped. Attempting to unplug a clone created using the `SNAPSHOT COPY` clause results in an error.

For a PDB created using the SNAPSHOT COPY clause in an Oracle Real Application Clusters (Oracle RAC) environment, each node that must access the PDB's files must be mounted. For Oracle RAC databases running on Linux or UNIX platforms, the underlying NFS volumes must be mounted. If the Oracle RAC database is running on a Windows platform and using Direct NFS for shared storage, then you must update the oranfstab file on all nodes with the created volume export and mount entries.

Storage clones are named and tagged using the new PDB GUID. You can query the CLONETAG column of DBA_PDB_HISTORY view to view clone tags for storage clones.

keystore_clause

Specify this clause if the source database has encrypted data or a keystore set. For *keystore_password*, specify the password for the keystore. You must provide this password even if the source database is using an auto-login software keystore. You can find if the source database has encrypted data by querying the DBA_ENCRYPTED_COLUMNS data dictionary view or the V$ENCRYPTED_TABLESPACES dynamic performance view.

pdb_refresh_mode_clause

The REFRESH MODE clause applies only when cloning a PDB. The source PDB must be in a remote CDB, that is, you must specify the source PDB using the FROM *src_pdb_name@dblink* clause.

This clause lets you specify the refresh mode of the PDB. You can use this clause to create a **refreshable PDB**. Changes in the source PDB can be propagated to the refreshable PDB, either manually or automatically. This operation is called a refresh. You can specify the following refresh modes:

- MANUAL - This mode allows you to refresh the refreshable PDB manually at any time by issuing an ALTER PLUGGABLE DATABASE REFRESH statement.
- EVERY *number* MINUTES - This mode instructs the database to refresh the refreshable PDB every *number* of minutes. This mode also allows you to refresh the PDB manually at any time by issuing an ALTER PLUGGABLE DATABASE REFRESH statement.
- NONE - If you specify this mode, then the clone PDB is not a refreshable PDB. The database cannot refresh the PDB automatically and you cannot refresh the PDB manually. If you specify this mode, then you cannot later change the PDB into a refreshable PDB. This is the default.

A refreshable PDB can be opened only in READ ONLY mode. A refreshable PDB must be closed in order for a refresh to occur. If it is not closed when you attempt to perform a manual refresh, then an error will occur. If it is not closed when the database attempts an automatic refresh, then the refresh will be deferred until the next scheduled refresh.

See Also:

- ALTER PLUGGABLE DATABASE REFRESH for information on refreshing a PDB manually
- ALTER PLUGGABLE DATABASE *pdb_refresh_mode_clause* for information on changing the refresh mode of a PDB
- *Oracle Database Administrator's Guide* for more information on refreshable PDBs

RELOCATE

The RELOCATE clause allows you to relocate a PDB from one CDB to another. The database first clones the source PDB to the target PDB, and then removes the source PDB. The database also moves the files associated with the PDB to a new location. This operation is the fastest way to relocate a PDB with minimal down time. The down time for the PDB is approximately the time required to copy the PDB's files from their old location to their new location. The source PDB can be open in READ WRITE mode and fully functional during the relocation operation.

In the *create_pdb_clone* clause, you must use the FROM *src_pdb_name*@*dblink* syntax to identify the location of the source PDB. For *src_pdb_name*, specify the name of the source PDB. For *dblink*, specify a database link that indicates the location of the source PDB. The database link must have been created in the CDB to which the PDB will be relocated. It can connect either to the root of the remote CDB or to the remote PDB.

See Also:

Oracle Database Administrator's Guide for the complete steps for relocating a PDB

NO DATA

The NO DATA clause applies only when cloning a PDB. This clause specifies that the source PDB's data model definition is cloned, but not the PDB's data. The dictionary data in the source PDB is cloned, but all user-created table and index data from the source PDB is discarded.

Restrictions on the NO DATA Clause

The following restrictions apply to the NO DATA clause:

- You cannot specify NO DATA when cloning a non-CDB.
- You cannot specify NO DATA if the source PDB contains clustered tables, Advanced Queuing (AQ) tables, index-organized tables, or tables that contain abstract data type columns.

HOST and PORT

These clauses are useful only if you are creating a PDB that you plan to reference from a proxy PDB. This type of PDB is called a referenced PDB. Refer to HOST and PORT for the full semantics of these clauses.

create_pdb_from_xml

This clause enables you to create a PDB by plugging an unplugged PDB or a non-CDB (the source database) into a CDB (the target CDB). If the source database is an unplugged PDB, then it may have been unplugged from the target CDB or a different CDB.

The source database and the target CDB must meet the following requirements:

- They must have the same endian format.
- They must have compatible character sets and national character sets, which means:
  - Every character in the source database character set is available in the target CDB character set.
  - Every character in the source database character set has the same code point value in the target CDB character set.
- They must have the same set of database options installed.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for the complete steps for creating a PDB by plugging an unplugged PDB into a CDB and creating a PDB using a non-CDB
> - *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_PDB` package

AS CLONE

Specify this clause only if the target CDB already contains a PDB that was created using the same set of data files. The source files remain as an unplugged PDB and can be used again. Specifying `AS CLONE` also ensures that Oracle Database generates new identifiers, such as DBID and GUID, for the new PDB.

USING

This clause lets you specify a file that contains information about the source database that your are plugging in. For *`filename`*, specify the full path name of the file. You can obtain this file in one of the following ways:

- If the source database is an unplugged PDB, then the file was created by the *`pdb_unplug_clause`* of `ALTER PLUGGABLE DATABASE` as follows:
  - If the filename ends with the extension .xml, then it is an XML file containing metadata about the PDB. In this case, you must ensure that the XML metadata file, as well as the PDB's data files, are in a location that is accessible to the CDB.
  - If the filename ends with the extension .pdb, then it is a PDB archive file. This is a compressed file that includes an XML file containing metadata about the PDB, as

well as the PDB's data files. The PDB archive file must exist in a location that is accessible to the CDB. When you use a .pdb archive file, this file is extracted when you plug in the PDB, and the PDB's files are placed in the same directory as the .pdb archive file. Therefore, the *source_file_directory* clause is not required.

- If the source database is a non-CDB, then you must create the XML metadata file using the DBMS_PDB package, and ensure that the XML metadata file, as well as the source non-CDB's data files, are in a location that is accessible to the CDB.

> **See Also:**
>
> - *pdb_unplug_clause* of ALTER PLUGGABLE DATABASE
> - *Oracle Database PL/SQL Packages and Types Reference* for more information on the DBMS_PDB package

source_file_name_convert

Specify this clause only if the contents of the XML file do not accurately describe the locations of the source files. If the files that must be used to plug in the source database are no longer in the location specified in the XML file, then use this clause to map the specified file names to the actual file names.

- For *filename_pattern*, specify the string for the location of the files as specified in the XML file.
- For *replacement_filename_pattern*, specify the string for the actual location that contains the files that must be used to create the PDB.

Oracle Database will replace *filename_pattern* with *replacement_filename_pattern* when searching for the source database files.

File name patterns cannot match files or directories managed by Oracle Managed Files.

If the files that must be used to create the PDB exist in the location specified in the XML file, you can either omit this clause or specify SOURCE_FILE_NAME_CONVERT=NONE.

source_file_directory

Specify this clause only if the contents of the XML file do not accurately describe the locations of the source files *and* the source files are all present in a single directory. This clause is convenient when you have a large number of data files and specifying a replacement file name pattern for each file using the *source_file_name_convert* clause is not feasible.

- For *directory_path_name*, specify the absolute path of the directory that contains the source files. The directory is scanned to find the appropriate files based on the unplugged PDB's XML file.

You can specify this clause for configurations that use Oracle Managed Files and for configurations that do not use Oracle Managed Files.

If the files that must be used to create the PDB exist in the location specified in the XML file, you can either omit this clause or specify SOURCE_FILE_DIRECTORY=NONE.

COPY

Specify `COPY` if you want the files listed in the XML file to be copied to the new location and used for the new PDB. This is the default. You can use the optional *`file_name_convert`* clause to use pattern replacement in the new file names. Refer to *file_name_convert* for the full semantics of this clause.

MOVE

Specify `MOVE` if you want the files listed in the XML file to be moved, rather than copied, to the new location and used for the new PDB. You can use the optional *`file_name_convert`* clause to use pattern replacement in the new file names. Refer to *file_name_convert* for the full semantics of this clause.

NOCOPY

Specify `NOCOPY` if you want the files for the PDB to remain in their current locations. Use this clause if there is no need to copy or move the files required to plug in the PDB.

service_name_convert

Use this clause to determine how the database renames services for the new PDB. Refer to *service_name_convert*::= for the full semantics of this clause.

default_tablespace

Use this clause to specify a permanent default tablespace for the PDB. Oracle Database will assign the default tablespace to any non-`SYSTEM` user for whom a different permanent tablespace is not specified. The *`tablespace`* must already exist in the source database. Because the tablespace already exists, you cannot specify the `DATAFILE` clause or the *`extent_management_clause`* when creating a PDB with the *`create_pdb_from_xml`* clause.

pdb_storage_clause

Use this clause to specify storage limits for the new PDB. Refer to *pdb_storage_clause* for the full semantics of this clause.

path_prefix_clause

Use this clause to ensure that all directory object paths associated with the PDB are restricted to the specified directory or its subdirectories. Refer to *path_prefix_clause* for the full semantics of this clause.

tempfile_reuse_clause

Specify `TEMPFILE REUSE` to instruct the database to format and reuse a temp file associated with the new PDB if it already exists. Refer to *tempfile_reuse_clause* for the full semantics of this clause.

HOST and PORT

These clauses are useful only if you are creating a PDB that you plan to reference from a proxy PDB. This type of PDB is called a referenced PDB. Refer to HOST and PORT for the full semantics of these clauses.

Examples

Creating a PDB by Using the Seed: Example

The following statement creates a PDB salespdb by using the seed in the CDB as a template. The administrative user salesadm is created and granted the dba role. The default tablespace assigned to any non-SYSTEM users for whom no permanent tablespace is assigned is sales. File names for the new PDB will be constructed by replacing /disk1/oracle/dbs/pdbseed/ in the file names in the seed with /disk1/oracle/dbs/salespdb/. All tablespaces that belong to sales must not exceed 2G. The location of all directory object paths associated with salespdb are restricted to the directory /disk1/oracle/dbs/salespdb/.

```
CREATE PLUGGABLE DATABASE salespdb
  ADMIN USER salesadm IDENTIFIED BY password
  ROLES = (dba)
  DEFAULT TABLESPACE sales
    DATAFILE '/disk1/oracle/dbs/salespdb/sales01.dbf' SIZE 250M AUTOEXTEND ON
  FILE_NAME_CONVERT = ('/disk1/oracle/dbs/pdbseed/',
                       '/disk1/oracle/dbs/salespdb/')
  STORAGE (MAXSIZE 2G)
  PATH_PREFIX = '/disk1/oracle/dbs/salespdb/';
```

Cloning a PDB From an Existing PDB: Example

The following statement creates a PDB newpdb by cloning PDB salespdb. PDBs newpdb and salespdb are in the same CDB. Because no storage limits are explicitly specified, there is no limit on the amount of storage for newpdb. The files are copied from /disk1/oracle/dbs/salespdb/ to /disk1/oracle/dbs/newpdb/. The location of all directory object paths associated with newpdb are restricted to the directory /disk1/oracle/dbs/newpdb/.

```
CREATE PLUGGABLE DATABASE newpdb FROM salespdb
  FILE_NAME_CONVERT = ('/disk1/oracle/dbs/salespdb/', '/disk1/oracle/dbs/
newpdb/')
  PATH_PREFIX = '/disk1/oracle/dbs/newpdb';
```

Plugging a PDB into a CDB: Example

The following statement plugs the PDB salespdb, which was previously unplugged, into the CDB. The details about the metadata describing salespdb are stored in the XML file /disk1/usr/salespdb.xml. The XML file does not accurately describe the current locations of the files. Therefore, the SOURCE_FILE_NAME_CONVERT clause is used to indicate that the files are in /disk2/oracle/dbs/salespdb/, not /disk1/oracle/dbs/salespdb/. The NOCOPY clause indicates that the files are already in the correct location. All tablespaces that belong to sales must not exceed 2G. A file with the same name as the temp file specified in the XML file exists in the target location. Therefore, the TEMPFILE REUSE clause is required.

```
CREATE PLUGGABLE DATABASE salespdb
  USING '/disk1/usr/salespdb.xml'
  SOURCE_FILE_NAME_CONVERT =
    ('/disk1/oracle/dbs/salespdb/', '/disk2/oracle/dbs/salespdb/')
  NOCOPY
  STORAGE (MAXSIZE 2G)
  TEMPFILE REUSE;
```

CREATE PROCEDURE

Purpose

Procedures are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE PROCEDURE` statement to create a standalone stored procedure or a call specification.

A **procedure** is a group of PL/SQL statements that you can call by name. A **call specification** (sometimes called call spec) declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells Oracle Database which Java method to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

Stored procedures offer advantages in the areas of development, integrity, security, performance, and memory allocation.

> **See Also:**
>
> - *Oracle Database Development Guide* for more information on stored procedures, including how to call stored procedures and for information about registering external procedures.
> - CREATE FUNCTION for information specific to functions, which are similar to procedures in many ways.
> - CREATE PACKAGE for information on creating packages. The `CREATE PROCEDURE` statement creates a procedure as a standalone schema object. You can also create a procedure as part of a package.
> - ALTER PROCEDURE and DROP PROCEDURE for information on modifying and dropping a standalone procedure.
> - CREATE LIBRARY for more information about shared libraries.

Prerequisites

To create or replace a procedure in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a procedure in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

To invoke a call spec, you may need additional privileges, for example, the `EXECUTE` object privilege on the C library for a C call spec.

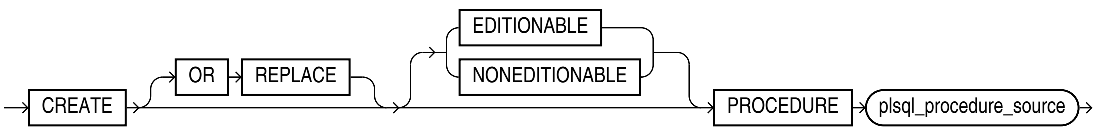
To embed a `CREATE PROCEDURE` statement inside an Oracle precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* or *Oracle Database Java Developer's Guide* for more information

Syntax

Procedures are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_procedure*::=**



(`plsql_procedure_source`: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping, re-creating, and regranting object privileges previously granted on it. If you redefine a procedure, then Oracle Database recompiles it.

Users who had previously been granted privileges on a redefined procedure can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the procedure, then Oracle Database marks the indexes `DISABLED`.

> **See Also:**
>
> ALTER PROCEDURE for information on recompiling procedures

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the procedure is an editioned or noneditioned object if editioning is enabled for the schema object type `PROCEDURE` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_procedure_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *`plsql_procedure_source`*.

CREATE PROFILE

> **Note:**
>
> Oracle recommends that you use the Database Resource Manager rather than this SQL statement to establish resource limits. The Database Resource Manager offers a more flexible means of managing and tracking resource use. For more information on the Database Resource Manager, refer to *Oracle Database Administrator's Guide.*

Purpose

Use the `CREATE PROFILE` statement to create a **profile**, which is a set of limits on database resources. If you assign the profile to a user, then that user cannot exceed these limits.

To specify resource limits for a user, you must:

- Enable resource limits dynamically with the `ALTER SYSTEM` statement or with the initialization parameter `RESOURCE_LIMIT`. This parameter does not apply to password resources. Password resources are always enabled.
- Create a profile that defines the limits using the `CREATE PROFILE` statement
- Assign the profile to the user using the `CREATE USER` or `ALTER USER` statement

In a multitenant environment, different profiles can be assigned to a common user in the root and in a PDB. When the common user logs in to the PDB, a profile whose setting applies to the session depends on whether the settings are password-related or resource-related.

- Password-related profile settings are fetched from the profile that is assigned to the common user in the root. For example, suppose you assign a common profile `c##prof` (in which `FAILED_LOGIN_ATTEMPTS` is set to 1) to common user `c##admin` in the root. In a PDB that user is assigned a local profile`local_prof` (in which `FAILED_LOGIN_ATTEMPTS` is set to 6.) Common user `c##admin` is allowed only one failed login attempt when he or she tries to log in to the PDB where `loc_prof` is assigned to him.
- Resource-related profile settings specified in the profile assigned to a user in a PDB get used without consulting resource-related settings in a profile assigned to the common user in the root. For example, if the profile `local_prof` that is assigned to user `c##admin` in a PDB has `SESSIONS_PER_USER` set to 2, then `c##admin` is only allowed only 2 concurrent sessions when he or she logs in to the PDB `loc_prof` is assigned to him, regardless of value of this setting in a profile assigned to him in the root.

> **See Also:**
>
> *Oracle Database Security Guide* for a detailed description and explanation of how to use password management and protection

Prerequisites

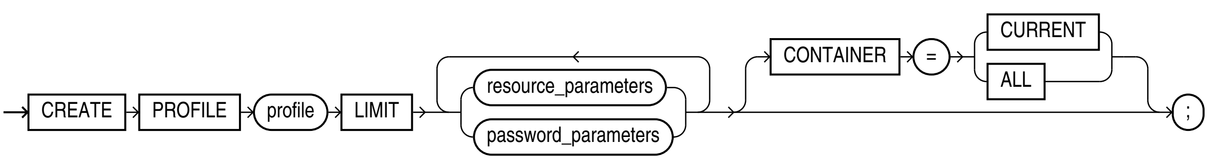
To create a profile, you must have the `CREATE PROFILE` system privilege.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB).

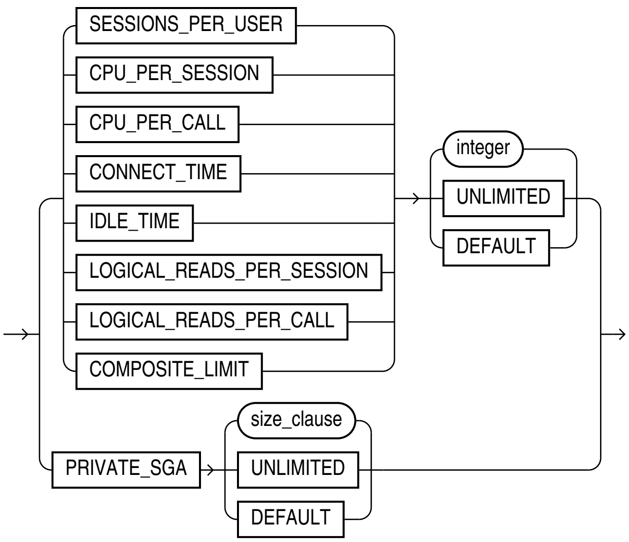
> **See Also:**
>
> - ALTER SYSTEM for information on enabling resource limits dynamically
> - *Oracle Database Reference* for information on the `RESOURCE_LIMIT` parameter
> - CREATE USER and ALTER USER for information on profiles

Syntax

***create_profile*::=**

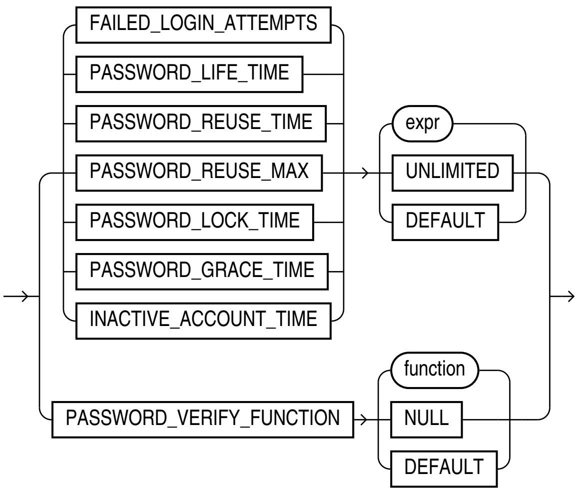


***resource_parameters*::=**



(*size_clause*::=

***password_parameters*::=**



Semantics

profile

Specify the name of the profile to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". Use profiles to limit the database resources available to a user for a single call or a single session.

In a non-CDB, a profile name cannot begin with `C##` or `c##`.

In a CDB, the requirements for a profile name are as follows:

- The name of a **common profile** must begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. By default, the prefix is `C##`.
- The name of a **local profile** must not begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. Regardless of the value of `COMMON_USER_PREFIX`, the name of a local profile can never begin with `C##` or `c##`.

> **Note:**
>
> If the value of `COMMON_USER_PREFIX` is an empty string, then there are no requirements for common or local profile names with one exception: the name of a local profile can never begin with `C##` or `c##`. Oracle recommends against using an empty string value because it might result in conflicts between the names of local and common profiles when a PDB is plugged into a different CDB, or when opening a PDB that was closed when a common user was created.

Oracle Database enforces resource limits in the following ways:

- If a user exceeds the `CONNECT_TIME` or `IDLE_TIME` session resource limit, then the database rolls back the current transaction and ends the session. When the user process next issues a call, the database returns an error.
- If a user attempts to perform an operation that exceeds the limit for other session resources, then the database aborts the operation, rolls back the current statement, and immediately returns an error. The user can then commit or roll back the current transaction, and must then end the session.
- If a user attempts to perform an operation that exceeds the limit for a single call, then the database aborts the operation, rolls back the current statement, and returns an error, leaving the current transaction intact.

> **Note:**
>
> - You can use fractions of days for all parameters that limit time, with days as units. For example, 1 hour is 1/24 and 1 minute is 1/1440.
> - You can specify resource limits for users regardless of whether the resource limits are enabled. However, Oracle Database does not enforce the limits until you enable them.

> **See Also:**
>
> "Creating a Profile: Example"

UNLIMITED

When specified with a resource parameter, `UNLIMITED` indicates that a user assigned this profile can use an unlimited amount of this resource. When specified with a password parameter, `UNLIMITED` indicates that no limit has been set for the parameter.

DEFAULT

Specify `DEFAULT` if you want to omit a limit for this resource in this profile. A user assigned this profile is subject to the limit for this resource specified in the `DEFAULT` profile. The `DEFAULT` profile initially defines unlimited resources. You can change those limits with the `ALTER PROFILE` statement.

Any user who is not explicitly assigned a profile is subject to the limits defined in the `DEFAULT` profile. Also, if the profile that is explicitly assigned to a user omits limits for some resources or specifies `DEFAULT` for some limits, then the user is subject to the limits on those resources defined by the `DEFAULT` profile.

resource_parameters

SESSIONS_PER_USER

Specify the number of concurrent sessions to which you want to limit the user.

CPU_PER_SESSION

Specify the CPU time limit for a session, expressed in hundredth of seconds.

CPU_PER_CALL

Specify the CPU time limit for a call (a parse, execute, or fetch), expressed in hundredths of seconds.

CONNECT_TIME

Specify the total elapsed time limit for a session, expressed in minutes.

IDLE_TIME

Specify the permitted periods of continuous inactive time during a session, expressed in minutes. Long-running queries and other operations are not subject to this limit.

When you set an idle timeout of X minutes, note that the session will take a couple of additional minutes to be terminated.

On the client application side, the error message shows up only the next time, when the idle client attempts to issue a new command.

LOGICAL_READS_PER_SESSION

Specify the permitted number of data blocks read in a session, including blocks read from memory and disk.

LOGICAL_READS_PER_CALL

Specify the permitted number of data blocks read for a call to process a SQL statement (a parse, execute, or fetch).

PRIVATE_SGA

Specify the amount of private space a session can allocate in the shared pool of the system global area (SGA). Refer to *size_clause* for information on that clause.

> **Note:**
>
> This limit applies only if you are using shared server architecture. The private space for a session in the SGA includes private SQL and PL/SQL areas, but not shared SQL and PL/SQL areas.

COMPOSITE_LIMIT

Specify the total resource cost for a session, expressed in **service units**. Oracle Database calculates the total service units as a weighted sum of `CPU_PER_SESSION`, `CONNECT_TIME`, `LOGICAL_READS_PER_SESSION`, and `PRIVATE_SGA`.

> **See Also:**
>
> - ALTER RESOURCE COST for information on how to specify the weight for each session resource
> - "Setting Profile Resource Limits: Example"

password_parameters

Use the following clauses to set password parameters. Parameters that set lengths of time—that is, all the password parameters except `FAILED_LOGIN_ATTEMPTS` and `PASSWORD_REUSE_MAX`—are interpreted in number of days. For testing purposes you can specify minutes (*n*/1440) or even seconds (*n*/86400) for these parameters. You can also use a decimal value for this purpose (for example .0833 for approximately one hour). The minimum value is 1 second. The maximum value is 24855 days. For `FAILED_LOGIN_ATTEMPTS` and `PASSWORD_REUSE_MAX`, you must specify an integer.

FAILED_LOGIN_ATTEMPTS

Specify the number of consecutive failed attempts to log in to the user account before the account is locked. If you omit this clause, then the default is 10 times.

PASSWORD_LIFE_TIME

Specify the number of days the same password can be used for authentication. If you also set a value for `PASSWORD_GRACE_TIME`, then the password expires if it is not changed within the grace period, and further connections are rejected. If you omit this clause, then the default is 180 days.

> **See Also:**
>
> *Oracle Database Security Guide* for information on setting `PASSWORD_LIFE_TIME` to a low value

PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX

These two parameters must be set in conjunction with each other. `PASSWORD_REUSE_TIME` specifies the number of days before which a password cannot be reused. `PASSWORD_REUSE_MAX` specifies the number of password changes required before the current password can be reused. For these parameter to have any effect, you must specify a value for both of them.

- If you specify a value for both of these parameters, then the user cannot reuse a password until the password has been changed the number of times specified for `PASSWORD_REUSE_MAX` during the number of days specified for `PASSWORD_REUSE_TIME`.

  For example, if you specify `PASSWORD_REUSE_TIME` to 30 and `PASSWORD_REUSE_MAX` to 10, then the user can reuse the password after 30 days if the password has already been changed 10 times.
- If you specify a value for either of these parameters and specify `UNLIMITED` for the other, then the user can never reuse a password.
- If you specify `DEFAULT` for either parameter, then Oracle Database uses the value defined in the `DEFAULT` profile. By default, all parameters are set to `UNLIMITED` in the `DEFAULT` profile. If you have not changed the default setting of `UNLIMITED` in the `DEFAULT` profile, then the database treats the value for that parameter as `UNLIMITED`.
- If you set both of these parameters to `UNLIMITED`, then the database ignores both of them. This is the default if you omit both parameters.

PASSWORD_LOCK_TIME

Specify the number of days an account will be locked after the specified number of consecutive failed login attempts. If you omit this clause, then the default is 1 day.

PASSWORD_GRACE_TIME

Specify the number of days after the grace period begins during which a warning is issued and login is allowed. If you omit this clause, then the default is 7 days.

INACTIVE_ACCOUNT_TIME

Specify the permitted number of consecutive days of no logins to the user account, after which the account will be locked. The minimum value is 15 days. The maximum value is 24855. If you omit this clause, then the default is `UNLIMITED`.

PASSWORD_VERIFY_FUNCTION

The `PASSWORD_VERIFY_FUNCTION` clause lets a PL/SQL password complexity verification script be passed as an argument to the `CREATE PROFILE` statement. Oracle Database provides a default script, but you can create your own routine or use third-party software instead.

- For *`function`*, specify the name of the password complexity verification routine. The function must exist in the `SYS` schema and you must have `EXECUTE` privilege on the function.
- Specify `NULL` to indicate that no password verification is performed.

If you specify *`expr`* for any of the password parameters, then the expression can be of any form except scalar subquery expression.

Restriction on Password Parameters

When you assign a profile to an external user or a global user, the password parameters do not take effect for that user.

> **See Also:**
>
> "Setting Profile Password Limits: Example"

CONTAINER Clause

The `CONTAINER` clause applies when you are connected to a CDB. However, it is not necessary to specify the `CONTAINER` clause because its default values are the only allowed values.

- To create a common profile, you must be connected to the root. You can optionally specify `CONTAINER = ALL`, which is the default when you are connected to the root.
- To create a local profile, you must be connected to a PDB. You can optionally specify `CONTAINER = CURRENT`, which is the default when you are connected to a PDB.

Examples

Creating a Profile: Example

The following statement creates the profile `new_profile`:

```
CREATE PROFILE new_profile
  LIMIT PASSWORD_REUSE_MAX 10
        PASSWORD_REUSE_TIME 30;
```

Setting Profile Resource Limits: Example

The following statement creates the profile `app_user`:

```
CREATE PROFILE app_user LIMIT
   SESSIONS_PER_USER          UNLIMITED
   CPU_PER_SESSION            UNLIMITED
   CPU_PER_CALL               3000
   CONNECT_TIME               45
   LOGICAL_READS_PER_SESSION  DEFAULT
   LOGICAL_READS_PER_CALL     1000
   PRIVATE_SGA                15K
   COMPOSITE_LIMIT            5000000;
```

If you assign the `app_user` profile to a user, then the user is subject to the following limits in subsequent sessions:

- The user can have any number of concurrent sessions.
- In a single session, the user can consume an unlimited amount of CPU time.
- A single call made by the user cannot consume more than 30 seconds of CPU time.
- A single session cannot last for more than 45 minutes.
- In a single session, the number of data blocks read from memory and disk is subject to the limit specified in the `DEFAULT` profile.
- A single call made by the user cannot read more than 1000 data blocks from memory and disk.
- A single session cannot allocate more than 15 kilobytes of memory in the SGA.
- In a single session, the total resource cost cannot exceed 5 million service units. The formula for calculating the total resource cost is specified by the `ALTER RESOURCE COST` statement.
- Since the `app_user` profile omits a limit for `IDLE_TIME` and for password limits, the user is subject to the limits on these resources specified in the `DEFAULT` profile.

Setting Profile Password Limits: Example

The following statement creates the `app_user2` profile with password limits values set:

```
CREATE PROFILE app_user2 LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LIFE_TIME 60
   PASSWORD_REUSE_TIME 60
   PASSWORD_REUSE_MAX 5
   PASSWORD_VERIFY_FUNCTION ora12c_verify_function
   PASSWORD_LOCK_TIME 1/24
   PASSWORD_GRACE_TIME 10
   INACTIVE_ACCOUNT_TIME 30;
```

This example uses the default Oracle Database password verification function, `ora12c_verify_function`. Refer to *Oracle Database Security Guide* for information on using this verification function provided or designing your own verification function.

CREATE RESTORE POINT

Purpose

Use the CREATE RESTORE POINT statement to create a **restore point**, which is a name associated with a timestamp or an SCN of the database. A restore point can be used to flash back a table or the database to the time specified by the restore point without the need to determine the SCN or timestamp. Restore points are also useful in various RMAN operations, including backups and database duplication. You can use RMAN to create restore points in the process of implementing an archival backup.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for more information on creating and using restore points and guaranteed restore points, for information on database duplication, and for information on archival backups
> - FLASHBACK DATABASE, FLASHBACK TABLE , and DROP RESTORE POINT for information on using and dropping restore points

Prerequisites

To create a normal restore point, you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege. To create a guaranteed restore point, you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege.

To view or use a restore point, you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege or the SELECT_CATALOG_ROLE role.

You can create a restore point on a primary or standby database. The database can be open, or mounted but not open. If the database is mounted, then it must have been shut down consistently before being mounted unless it is a physical standby database.

You must have created a fast recovery area before creating a guaranteed restore point. You need not enable flashback database before you create the restore point. The database must be in ARCHIVELOG mode if you are creating a guaranteed restore point.

You can create, use, or view a restore point when connected to a multitenant container database (CDB) as follows:

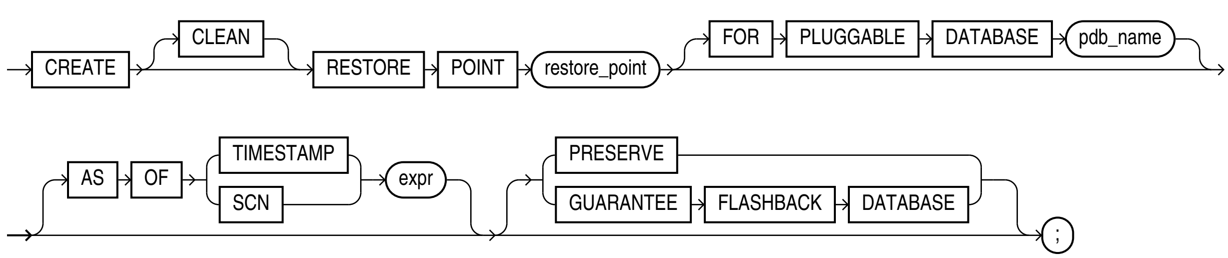
- To create a normal CDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly.
- To create a guaranteed CDB restore point, the current container must be the root and you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly.
- To view a CDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege or the SELECT_CATALOG_ROLE role, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly, or the current container must be a PDB and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE,

SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.

- To use a CDB restore point, you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege or the SELECT_CATALOG_ROLE role, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly.
- To create a normal PDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly, or the current container must be the PDB for which you want to create the restore point and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.
- To create a guaranteed PDB restore point, the current container must be the root and you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly, or the current container must be the PDB for which you want to create the restore point and you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.
- To view a PDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege or the SELECT_CATALOG_ROLE role, either granted commonly or granted locally in the root, or the SYSDBA, SYSBACKUP, or SYSDG system privilege granted commonly, or the current container must be the PDB for the restore point and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.
- To use a PDB restore point, the current container must be the PDB for the restore point and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.

Syntax

***create_restore_point*::=**



Semantics

CLEAN

You can specify CLEAN only when creating a PDB restore point. The PDB must use shared undo and must be closed with no outstanding transactions. Flashing back a PDB using shared undo to a clean PDB restore point does not require restoring backups or creating a clone instance. Therefore, it is faster than flashing back a PDB using shared undo to an SCN or other type of restore point.

restore_point

Specify the name of the restore point. The name must satisfy the requirements listed in "Database Object Naming Rules ".

In a multitenant environment, the CDB and PDBs have their own namespaces for restore points. Therefore, the CDB and each PDB can have a restore point with the same name. When you specify a restore point name in a PDB or for a PDB operation, the name is first interpreted as a PDB restore point for the concerned PDB. If a PDB restore point with the specified name is not found, then it is interpreted as a CDB restore point.

The database can retain at least 2048 normal restore points. In a Multitenant environment, a CDB can retain at least 2048 normal restore points across the entire CDB, including PDB restore points. Normal restore points are retained in the database for at least the number of days specified for the CONTROL_FILE_RECORD_KEEP_TIME initialization parameter. The default value of that parameter is 7 days. Guaranteed restore points are retained in the database until explicitly dropped by the user.

If you specify neither PRESERVE nor GUARANTEE FLASHBACK DATABASE, then the resulting restore point enables you to flash the database back to a restore point within the time period determined by the DB_FLASHBACK_RETENTION_TARGET initialization parameter. The database automatically manages such restore points. When the maximum number of restore points is reached, according to the rules described in *restore_point* above, the database automatically drops the oldest restore point. Under some circumstances the restore points will be retained in the RMAN recovery catalog for use in restoring long-term backups. You can explicitly drop a restore point using the DROP RESTORE POINT statement.

FOR PLUGGABLE DATABASE

This clause enables you to create a PDB restore point when you are connected to the root. For *pdb_name*, specify the name of the PDB.

If you are connected to the PDB for which you want to create the restore point, then it is not necessary to specify this clause. However, if you specify this clause, then you must specify the name of the PDB to which you are connected.

AS OF Clause

Use this clause to create a restore point at a specified datetime or SCN in the past. If you specify TIMESTAMP, then *expr* must be a valid datetime expression resolving to a time in the past. If you specify SCN, then *expr* must be a valid SCN in the database in the past. In either case, *expr* must refer to a datetime or SCN in the current incarnation of the database.

PRESERVE

Specify PRESERVE to indicate that the restore point must be explicitly deleted. Such restore points are useful for preserving a flashback database.

GUARANTEE FLASHBACK DATABASE

A guaranteed restore point enables you to flash the database back deterministically to the restore point regardless of the DB_FLASHBACK_RETENTION_TARGET initialization parameter setting. The guaranteed ability to flash back depends on sufficient space being available in the fast recovery area.

Guaranteed restore points guarantee only that the database will maintain enough flashback logs to flashback the database to the guaranteed restore point. It does not guarantee that the database will have enough undo to flashback any table to the same restore point.

Guaranteed restore points are always preserved. They must be dropped explicitly by the user using the `DROP RESTORE POINT` statement. They do not age out. Guaranteed restore points can use considerable space in the fast recovery area. Therefore, Oracle recommends that you create guaranteed restore points only after careful consideration.

Examples

Creating and Using a Restore Point: Example

The following example creates a normal restore point, updates a table, and then flashes back the altered table to the restore point. The example assumes the user `hr` has the appropriate system privileges to use each of the statements.

```
CREATE RESTORE POINT good_data;

SELECT salary FROM employees WHERE employee_id = 108;

    SALARY
----------
     12000

UPDATE employees SET salary = salary*10
   WHERE employee_id = 108;

SELECT salary FROM employees
   WHERE employee_id = 108;

    SALARY
----------
    120000

COMMIT;

FLASHBACK TABLE employees TO RESTORE POINT good_data;

SELECT salary FROM employees
   WHERE employee_id = 108;

    SALARY
----------
     12000
```

CREATE ROLE

Purpose

Use the `CREATE ROLE` statement to create a **role**, which is a set of privileges that can be granted to users or to other roles. You can use roles to administer database privileges. You can add privileges to a role and then grant the role to a user. The user can then enable the role and exercise the privileges granted by the role.

A role contains all privileges granted to the role and all privileges of other roles granted to it. A new role is initially empty. You add privileges to a role with the `GRANT` statement.

If you create a role that is NOT IDENTIFIED or is IDENTIFIED EXTERNALLY or BY *password*, then Oracle Database grants you the role with ADMIN OPTION. However, if you create a role IDENTIFIED GLOBALLY, then the database does not grant you the role. A global role cannot be granted to a user or role directly. Global roles can be granted only through enterprise roles.

> **See Also:**
>
> - GRANT for information on granting roles
> - ALTER USER for information on enabling roles
> - ALTER ROLE and DROP ROLE for information on modifying or removing a role from the database
> - SET ROLE for information on enabling and disabling roles for the current session
> - *Oracle Database Security Guide* for general information about roles
> - *Oracle Database Enterprise User Security Administrator's Guide* for details on enterprise roles

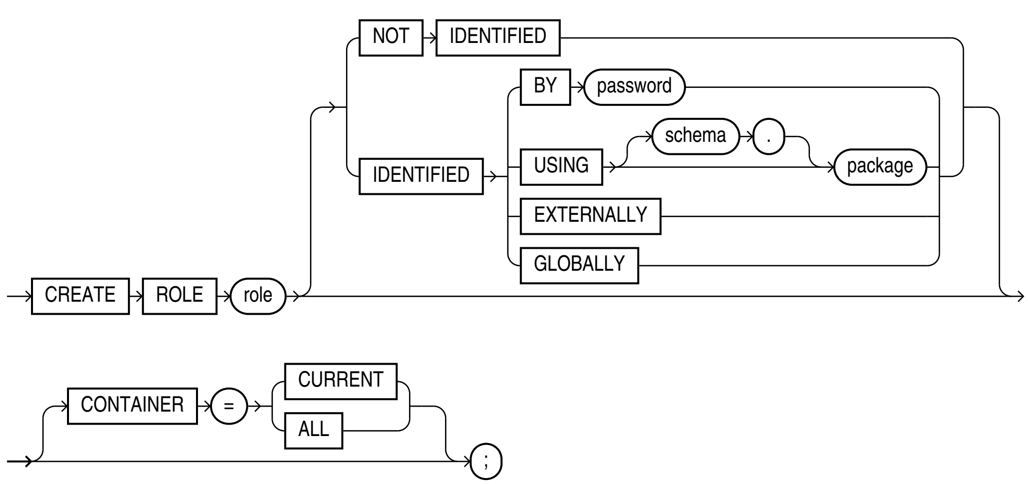
Prerequisites

You must have the CREATE ROLE system privilege.

To specify the CONTAINER clause, you must be connected to a multitenant container database (CDB). To specify CONTAINER = ALL, the current container must be the root. To specify CONTAINER = CURRENT, the current container must be a pluggable database (PDB).

Syntax

***create_role*::=**



Semantics

role

Specify the name of the role to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ". Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters. The maximum length of the role name is 128 bytes. The maximum number of user-defined roles that can be enabled for a single user at one time is 148.

In a non-CDB, a role name cannot begin with `C##` or `c##`.

In a CDB, the requirements for a role name are as follows:

- The name of a **common role** must begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. By default, the prefix is `C##`.
- The name of a **local role** must not begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. Regardless of the value of `COMMON_USER_PREFIX`, the name of a local role can never begin with `C##` or `c##`.

> **Note:**
>
> If the value of `COMMON_USER_PREFIX` is an empty string, then there are no requirements for common or local role names with one exception: the name of a local role can never begin with `C##` or `c##`. Oracle recommends against using an empty string value because it might result in conflicts between the names of local and common roles when a PDB is plugged into a different CDB, or when opening a PDB that was closed when a common user was created.

Some roles are defined by SQL scripts provided on your distribution media.

> **See Also:**
>
> GRANT for a list of these predefined roles and SET ROLE for information on enabling and disabling roles for a user

NOT IDENTIFIED Clause

Specify `NOT IDENTIFIED` to indicate that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED Clause

Use the `IDENTIFIED` clause to indicate that a user must be authorized by the specified method before the role is enabled with the `SET ROLE` statement.

BY *password*

The `BY` *`password`* clause lets you create a **local role** and indicates that the user must specify the password to the database when enabling the role. The password can contain single-byte or multibyte characters. If you use a multibyte password, you must ensure that your database and client are both configured to support NLS.

USING *package*

The `USING` *`package`* clause lets you create a **secure application role**, which is a role that can be enabled only by applications using an authorized package. If you do not specify *`schema`*, then the database assumes the package is in your own schema.

> **See Also:**
>
> *Oracle Database Security Guide* for information on creating a secure application role

EXTERNALLY

Specify `EXTERNALLY` to create an **external role.** An external user must be authorized by an external service, such as an operating system or third-party service, before enabling the role.

Depending on the operating system, the user may have to specify a password to the operating system before the role is enabled.

GLOBALLY

Specify `GLOBALLY` to create a **global role** . A global user must be authorized to use the role by the enterprise directory service before the role is enabled at login.

Specify `GLOBALLY` with `AS` to map a directory group to a global role when using centrally managed users. The directory group is identified by its domain name.

Example: Map a Directory User to a Global User

```
  CREATE USER scott_global IDENTIFIED GLOBALLY AS ‘cn=scott
taylor,ou=sales,dc=abccorp,dc=com’;
```

This effectively maps a directory user named ‘scott taylor’ in the ‘sales’ organization unit of the abccorp.com domain to a database global user ‘scott_global’.

CONTAINER Clause

The `CONTAINER` clause applies when you are connected to a CDB. However, it is not necessary to specify the `CONTAINER` clause because its default values are the only allowed values.

- To create a common role, you must be connected to the root. You can optionally specify `CONTAINER = ALL`, which is the default when you are connected to the root.
- To create a local role, you must be connected to a PDB. You can optionally specify `CONTAINER = CURRENT`, which is the default when you are connected to a PDB.

Examples

Creating a Role: Example

The following statement creates the role dw_manager:

```
CREATE ROLE dw_manager;
```

Users who are subsequently granted the dw_manager role will inherit all of the privileges that have been granted to this role.

You can add a layer of security to roles by specifying a password, as in the following example:

```
CREATE ROLE dw_manager
   IDENTIFIED BY warehouse;
```

Users who are subsequently granted the dw_manager role must specify the password warehouse to enable the role with the SET ROLE statement.

The following statement creates global role warehouse_user:

```
CREATE ROLE warehouse_user IDENTIFIED GLOBALLY;
```

The following statement creates the same role as an external role:

```
CREATE ROLE warehouse_user IDENTIFIED EXTERNALLY;
```

The following statement creates local role role1 in the current PDB. The current container must be a PDB when you issue this statement:

```
CREATE ROLE role1 CONTAINER = CURRENT;
```

The following statement creates common role c##role1. The current container must be the root when you issue this statement:

```
CREATE ROLE c##role1 CONTAINER = ALL;
```

CREATE ROLLBACK SEGMENT

> **Note:**
>
> Oracle strongly recommends that you run your database in automatic undo management mode instead of using rollback segments. Do not use rollback segments unless you must do so for compatibility with earlier versions of Oracle Database. Refer to *Oracle Database Administrator's Guide* for information on automatic undo management.

Purpose

Use the CREATE ROLLBACK SEGMENT statement to create a **rollback segment**, which is an object that Oracle Database uses to store data necessary to reverse, or undo, changes made by transactions.

The information in this section assumes that your database is not running in automatic undo mode (the `UNDO_MANAGEMENT` initialization parameter is set to `MANUAL` or not set at all). If your database is running in automatic undo mode (the `UNDO_MANAGEMENT` initialization parameter is set to `AUTO`, which is the default), then rollback segments are not permitted. However, errors generated in rollback segment operations are suppressed.

Further, if your database has a locally managed `SYSTEM` tablespace, then you cannot create rollback segments in any dictionary-managed tablespace. Instead, you must either use the automatic undo management feature or create locally managed tablespaces to hold the rollback segments.

> **Note:**
>
> A tablespace can have multiple rollback segments. Generally, multiple rollback segments improve performance.
>
> - The tablespace must be online for you to add a rollback segment to it.
> - When you create a rollback segment, it is initially offline. To make it available for transactions by your Oracle Database instance, bring it online using the `ALTER ROLLBACK SEGMENT` statement. To bring it online automatically whenever you start up the database, add the segment name to the value of the `ROLLBACK_SEGMENT` initialization parameter.

To use objects in a tablespace other than the `SYSTEM` tablespace:

- If you are using rollback segments for undo, then at least one rollback segment (other than the `SYSTEM` rollback segment) must be online.
- If you are running the database in automatic undo mode, then at least one `UNDO` tablespace must be online.

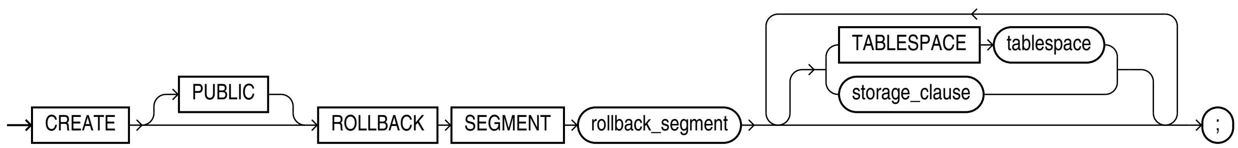
> **See Also:**
>
> - ALTER ROLLBACK SEGMENT for information on altering a rollback segment
> - DROP ROLLBACK SEGMENT for information on removing a rollback segment
> - *Oracle Database Reference* for information on the `UNDO_MANAGEMENT` parameter
> - *Oracle Database Administrator's Guide* for information on automatic undo mode

Prerequisites

To create a rollback segment, you must have the `CREATE ROLLBACK SEGMENT` system privilege.

Syntax

***create_rollback_segment*::=**



(*storage_clause*)

Semantics

PUBLIC

Specify `PUBLIC` to indicate that the rollback segment is public and is available to any instance. If you omit this clause, then the rollback segment is private and is available only to the instance naming it in its initialization parameter `ROLLBACK_SEGMENTS`.

rollback_segment

Specify the name of the rollback segment to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

TABLESPACE

Use the `TABLESPACE` clause to identify the tablespace in which the rollback segment is created. If you omit this clause, then the database creates the rollback segment in the `SYSTEM` tablespace.

> **Note:**
>
> Oracle Database must access rollback segments frequently. Therefore, Oracle strongly recommends that you do not create rollback segments in the `SYSTEM` tablespace, either explicitly or implicitly by omitting this clause. In addition, to avoid high contention for the tablespace containing the rollback segment, it should not contain other objects such as tables and indexes, and it should require minimal extent allocation and deallocation.
>
> To achieve these goals, create rollback segments in locally managed tablespaces with autoallocation disabled—in tablespaces created with the `EXTENT MANAGEMENT LOCAL` clause with the `UNIFORM` setting. The `AUTOALLOCATE` setting is not supported.

> **See Also:**
>
> CREATE TABLESPACE

storage_clause

The *storage_clause* lets you specify storage characteristics for the rollback segment.

- The OPTIMAL parameter of the *storage_clause* is of particular interest, because it applies only to rollback segments.
- You cannot specify the PCTINCREASE parameter of the *storage_clause* with CREATE ROLLBACK SEGMENT.

> **See Also:**
>
> *storage_clause*

Examples

Creating a Rollback Segment: Example

The following statement creates a rollback segment with default storage values in an appropriately configured tablespace:

```
CREATE TABLESPACE rbs_ts
   DATAFILE 'rbs01.dbf' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 100K;

/* This example and the next will fail if your database is in
   automatic undo mode.
*/
CREATE ROLLBACK SEGMENT rbs_one
   TABLESPACE rbs_ts;
```

The preceding statement is equivalent to the following:

```
CREATE ROLLBACK SEGMENT rbs_one
   TABLESPACE rbs_ts
   STORAGE
   ( INITIAL 10K );
```

CREATE SCHEMA

Purpose

Use the CREATE SCHEMA statement to create multiple tables and views and perform multiple grants in your own schema in a single transaction.

To execute a CREATE SCHEMA statement, Oracle Database executes each included statement. If all statements execute successfully, then the database commits the transaction. If any statement results in an error, then the database rolls back all the statements.

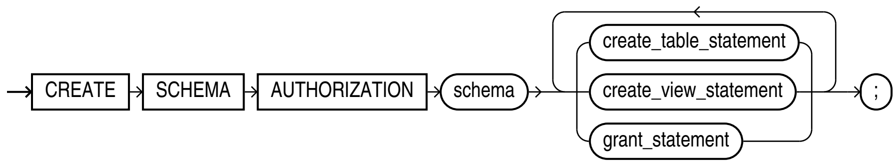
> **Note:**
>
> This statement does not actually create a schema. Oracle Database automatically creates a schema when you create a user (see CREATE USER). This statement lets you populate your schema with tables and views and grant privileges on those objects without having to issue multiple SQL statements in multiple transactions.

Prerequisites

The `CREATE SCHEMA` statement can include `CREATE TABLE`, `CREATE VIEW`, and `GRANT` statements. To issue a `CREATE SCHEMA` statement, you must have the privileges necessary to issue the included statements.

Syntax

***create_schema*::=**



Semantics

schema

Specify the name of the schema. The schema name must be the same as your Oracle Database username.

create_table_statement

Specify a `CREATE TABLE` statement to be issued as part of this `CREATE SCHEMA` statement. Do not end this statement with a semicolon (or other terminator character).

> **See Also:**
>
> CREATE TABLE

create_view_statement

Specify a `CREATE VIEW` statement to be issued as part of this `CREATE SCHEMA` statement. Do not end this statement with a semicolon (or other terminator character).

> **See Also:**
>
> CREATE VIEW

grant_statement

Specify a `GRANT` statement to be issued as part of this `CREATE SCHEMA` statement. Do not end this statement with a semicolon (or other terminator character). You can use this clause to grant object privileges on objects you own to other users. You can also grant system privileges to other users if you were granted those privileges `WITH ADMIN OPTION`.

> **See Also:**
>
> GRANT

The `CREATE SCHEMA` statement supports the syntax of these statements only as defined by standard SQL, rather than the complete syntax supported by Oracle Database.

The order in which you list the `CREATE TABLE`, `CREATE VIEW`, and `GRANT` statements is unimportant. The statements within a `CREATE SCHEMA` statement can reference existing objects or objects you create in other statements within the same `CREATE SCHEMA` statement.

Restriction on Granting Privileges to a Schema

The syntax of the *`parallel_clause`* is allowed for a `CREATE TABLE` statement in `CREATE SCHEMA`, but parallelism is not used when creating the objects.

> **See Also:**
>
> The *parallel_clause* in the `CREATE TABLE` documentation

Examples

Creating a Schema: Example

The following statement creates a schema named `oe` for the sample order entry user `oe`, creates the table `new_product`, creates the view `new_product_view`, and grants the `SELECT` object privilege on `new_product_view` to the sample human resources user `hr`.

```
CREATE SCHEMA AUTHORIZATION oe
   CREATE TABLE new_product
      (color VARCHAR2(10)  PRIMARY KEY, quantity NUMBER)
   CREATE VIEW new_product_view
      AS SELECT color, quantity FROM new_product WHERE color = 'RED'
   GRANT select ON new_product_view TO hr;
```

15 SQL Statements: CREATE SEQUENCE to DROP CLUSTER

This chapter contains the following SQL statements:

- CREATE SEQUENCE
- CREATE SPFILE
- CREATE SYNONYM
- CREATE TABLE
- CREATE TABLESPACE
- CREATE TABLESPACE SET
- CREATE TRIGGER
- CREATE TYPE
- CREATE TYPE BODY
- CREATE USER
- CREATE VIEW
- DELETE
- DISASSOCIATE STATISTICS
- DROP ANALYTIC VIEW
- DROP ATTRIBUTE DIMENSION
- DROP AUDIT POLICY (Unified Auditing)
- DROP CLUSTER

CREATE SEQUENCE

Purpose

Use the `CREATE SEQUENCE` statement to create a **sequence**, which is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, then the sequence numbers each user acquires may have gaps, because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. After a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

After a sequence is created, you can access its values in SQL statements with the `CURRVAL` pseudocolumn, which returns the current value of the sequence, or the `NEXTVAL` pseudocolumn, which increments the sequence and returns the new value.

> **See Also:**
>
> - Pseudocolumns for more information on using the `CURRVAL` and `NEXTVAL`
> - "How to Use Sequence Values " for information on using sequences
> - ALTER SEQUENCE or DROP SEQUENCE for information on modifying or dropping a sequence

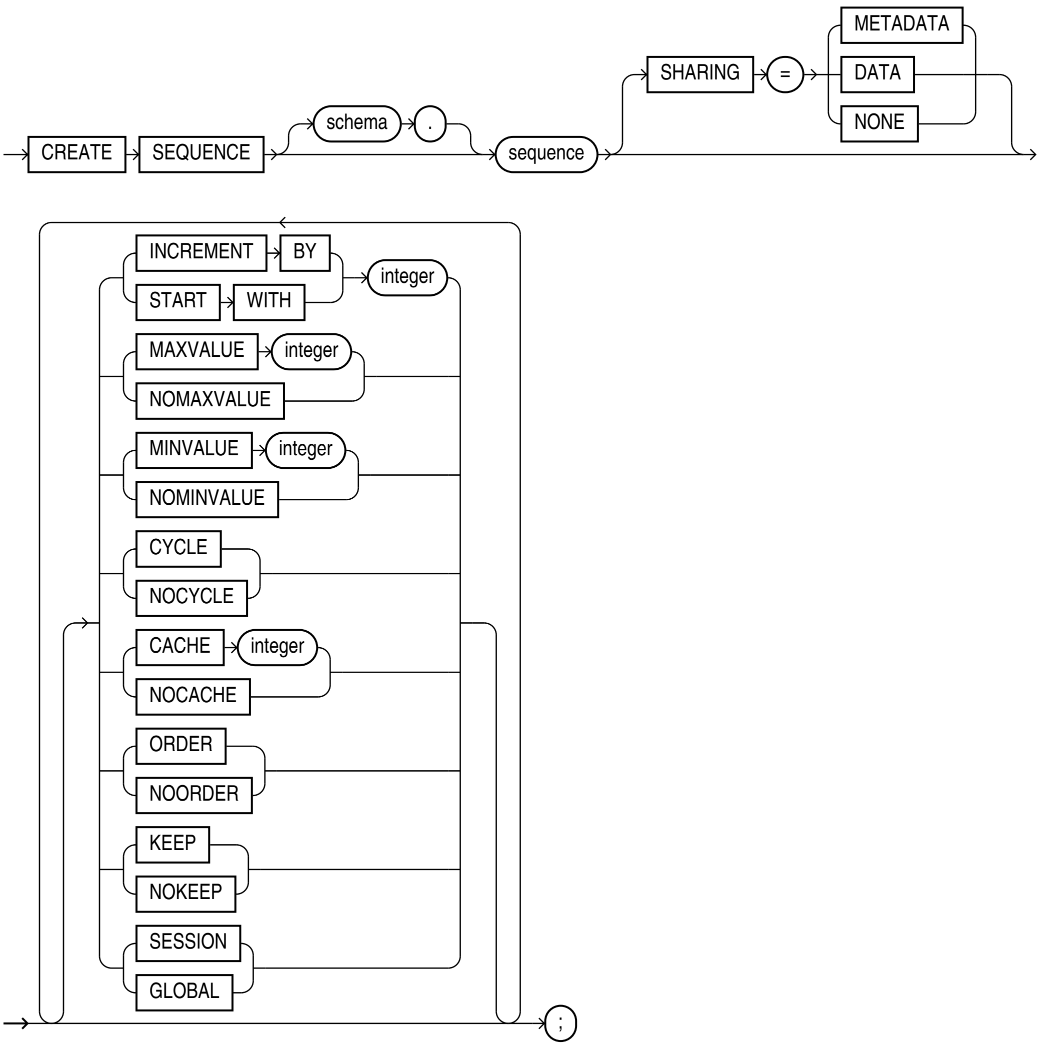
Prerequisites

To create a sequence in your own schema, you must have the `CREATE SEQUENCE` system privilege.

To create a sequence in another user's schema, you must have the `CREATE ANY SEQUENCE` system privilege.

Syntax

***create_sequence*::=**



Semantics

schema

Specify the schema to contain the sequence. If you omit `schema`, then Oracle Database creates the sequence in your own schema.

sequence

Specify the name of the sequence to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

If you specify none of the clauses `INCREMENT BY` through `GLOBAL`, then you create an ascending sequence that starts with 1 and increases by 1 with no upper limit. Specifying only `INCREMENT BY` -1 creates a descending sequence that starts with -1 and decreases with no lower limit.

- To create a sequence that increments without bound, for ascending sequences, omit the `MAXVALUE` parameter or specify `NOMAXVALUE`. For descending sequences, omit the `MINVALUE` parameter or specify the `NOMINVALUE`.
- To create a sequence that stops at a predefined limit, for an ascending sequence, specify a value for the `MAXVALUE` parameter. For a descending sequence, specify a value for the `MINVALUE` parameter. Also specify `NOCYCLE`. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- To create a sequence that restarts after reaching a predefined limit, specify values for both the `MAXVALUE` and `MINVALUE` parameters. Also specify `CYCLE`.

SHARING

This clause applies only when creating a sequence in an application root. This type of sequence is called an application common object and it can be shared with the application PDBs that belong to the application root. To determine how the sequence is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the sequence's metadata, but its data is unique to each container. This type of sequence is referred to as a **metadata-linked application common object**.
- `DATA` - A data link shares the sequence, and its data is the same for all containers in the application container. Its data is stored only in the application root. This type of sequence is referred to as a **data-linked application common object**.
- `NONE` - The sequence is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the sequence. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

You cannot change the sharing attribute of a sequence after it is created.

> **See Also:**
>
> - *Oracle Database Reference* for more information on the `DEFAULT_SHARING` initialization parameter
> - *Oracle Database Administrator's Guide* for complete information on creating application common objects

INCREMENT BY

Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits for an ascending sequence and 27 or fewer digits for a descending sequence. The absolute of this value must be less than the difference of `MAXVALUE` and `MINVALUE`. If this value is negative, then the sequence descends. If the value is positive, then the sequence ascends. If you omit this clause, then the interval defaults to 1.

START WITH

Specify the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence. For descending sequences, the default value is the maximum value of the sequence. This integer value can have 28 or fewer digits for positive values and 27 or fewer digits for negative values.

> **Note:**
>
> This value is not necessarily the value to which an ascending or descending cycling sequence cycles after reaching its maximum or minimum value, respectively.

MAXVALUE

Specify the maximum value the sequence can generate. This integer value can have 28 or fewer digits for positive values and 27 or fewer digits for negative values. `MAXVALUE` must be equal to or greater than `START WITH` and must be greater than `MINVALUE`.

NOMAXVALUE

Specify `NOMAXVALUE` to indicate a maximum value of 10^{28}-1 for an ascending sequence or -1 for a descending sequence. This is the default.

MINVALUE

Specify the minimum value of the sequence. This integer value can have 28 or fewer digits for positive values and 27 or fewer digits for negative values. `MINVALUE` must be less than or equal to `START WITH` and must be less than `MAXVALUE`.

NOMINVALUE

Specify `NOMINVALUE` to indicate a minimum value of 1 for an ascending sequence or -(10^{27} -1) for a descending sequence. This is the default.

CYCLE

Specify `CYCLE` to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum value.

NOCYCLE

Specify `NOCYCLE` to indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.

CACHE

Specify how many values of the sequence the database preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for `CACHE` must be less than the value determined by the following formula:

```
CEIL ((MAXVALUE - MINVALUE) / ABS (INCREMENT))
```

If a system failure occurs, then all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the `CACHE` parameter.

> **Note:**
>
> Oracle recommends using the `CACHE` setting to enhance performance if you are using sequences in an Oracle Real Application Clusters environment.

NOCACHE

Specify `NOCACHE` to indicate that values of the sequence are not preallocated. If you omit both `CACHE` and `NOCACHE`, then the database caches 20 sequence numbers by default.

ORDER

Specify `ORDER` to guarantee that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys.

NOORDER

Specify `NOORDER` if you do not want to guarantee sequence numbers are generated in order of request. This is the default.

KEEP

Specify `KEEP` if you want `NEXTVAL` to retain its original value during replay for Application Continuity. This behavior will occur only if the user running the application is the owner of the schema containing the sequence. This clause is useful for providing bind variable consistency at replay after recoverable errors. Refer to *Oracle Database Development Guide* for more information on Application Continuity.

NOKEEP

Specify `NOKEEP` if you do not want `NEXTVAL` to retain its original value during replay for Application Continuity. This is the default.

> **Note:**
>
> The `KEEP` and `NOKEEP` clauses apply only to the owner of the schema containing the sequence. You can control whether `NEXTVAL` retains its original value for other users during replay for Application Continuity by granting or revoking the `KEEP SEQUENCE` object privilege on the sequence. Refer to Table 18-2 for more information on the `KEEP SEQUENCE` object privilege.

SESSION

Specify `SESSION` to create a session sequence, which is a special type of sequence that is specifically designed to be used with global temporary tables that have session visibility. Unlike the existing regular sequences (referred to as "global" sequences for the sake of comparison), a session sequence returns a unique range of sequence numbers only within a session, but not across sessions. Another difference is that session sequences are not persistent. If a session goes away, so does the state of the session sequences that were accessed during the session.

Session sequences must be created by a read-write database but can be accessed on any read-write or read-only databases (either a regular database temporarily open read-only or a standby database).

The `CACHE`, `NOCACHE`, `ORDER`, or `NOORDER` clauses are ignored when specified with the `SESSION` clause.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for more information on session sequences

GLOBAL

Specify `GLOBAL` to create a global, or regular, sequence. This is the default.

Examples

Creating a Sequence: Example

The following statement creates the sequence `customers_seq` in the sample schema `oe`. This sequence could be used to provide customer ID numbers when rows are added to the `customers` table.

```
CREATE SEQUENCE customers_seq
 START WITH     1000
 INCREMENT BY   1
 NOCACHE
 NOCYCLE;
```

The first reference to `customers_seq.nextval` returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

CREATE SPFILE

Purpose

Use the `CREATE SPFILE` statement to create a server parameter file either from a traditional plain-text initialization parameter file or from the current system-wide settings. Server parameter files are binary files that exist only on the server and are called from client locations to start up the database.

Server parameter files let you make persistent changes to individual parameters. When you use a server parameter file, you can specify in an `ALTER SYSTEM SET` *`parameter`* statement that the new parameter value should be persistent. This means that the new value applies not only in the current instance, but also to any instances that are started up subsequently. Traditional plain-text parameter files do not let you make persistent changes to parameter values.

Server parameter files are located on the server, so they allow for automatic database tuning by Oracle Database and for backup by Recovery Manager (RMAN).

To use a server parameter file when starting up the database, you must create it using the `CREATE SPFILE` statement.

All instances in an Oracle Real Application Clusters environment must use the same server parameter file. However, when otherwise permitted, individual instances can have different settings of the same parameter within this one file. Instance-specific parameter definitions are specified as *`SID.parameter = value`*, where *`SID`* is the instance identifier.

The method of starting up the database with a server parameter file depends on whether you create a default or nondefault server parameter file. Refer to "Creating a Server Parameter File: Examples" for examples of how to use server parameter files.

Note on Creating Server Parameter Files in a CDB

When you create a server parameter file in a multitenant container database (CDB), the current container can be the root or a PDB.

- If the current container is the root, then the values that you set for initialization parameters in the root are used as default values for all other containers.
- If the current container is a PDB, then the database stores the PDB's initialization parameter values internally, rather than in a file. Therefore, you cannot specify an *`spfile_name`*. The values that you set for initialization parameters in the PDB are persistent and override any values set for those parameters in the root.

You can subsequently use the `ALTER SYSTEM` statement to modify initialization parameter values for the root or a PDB.

See Also:

- CREATE PFILE for information on creating a regular text parameter file from a binary server parameter file
- *Oracle Database Administrator's Guide* for information on traditional plain-text initialization parameter files and server parameter files
- *Oracle Real Application Clusters Administration and Deployment Guide* for information on using server parameter files in an Oracle Real Application Clusters environment

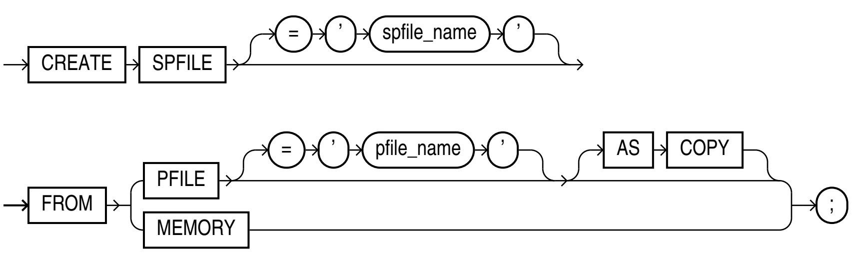
Prerequisites

You must have the SYSBACKUP, SYSDBA, SYSDG, or SYSOPER system privilege to execute this statement. You can execute this statement before or after instance startup. However, if you have already started an instance using *spfile_name*, you cannot specify the same *spfile_name* in this statement.

To create a server parameter file in a CDB, the current container must be the root and you must have the commonly granted SYSBACKUP, SYSDBA, SYSDG, or SYSOPER system privilege.

Syntax

***create_spfile*::=**



Semantics

spfile_name

This clause lets you specify a name for the server parameter file you are creating.

If you specify *spfile_name*, then Oracle Database creates a nondefault server parameter file.

- For *spfile_name*, you can specify a traditional filename, a file in an Oracle ACFS file system, or an Oracle Storage Management (Oracle ASM) filename.
- If you specify a traditional filename or a file in an Oracle ACFS file system, then *spfile_name* can include a path prefix. If you do not specify such a path prefix, then the database adds the path prefix for the default storage location, which is platform dependent.
- If you specify the Oracle ASM filename syntax, then the database creates the spfile in an Oracle ASM disk group.

- When using a nondefault server parameter file, you must specify the server parameter filename in the `STARTUP` command when you start up the database. The exception to this rule is as follows:
  - If the database is defined as a resource in Oracle Clusterware, the instance from which the command is issued is running, and you specify the *spfile_name*, specify the `FROM PFILE` clause, and omit the `AS COPY` clause, then this statement automatically updates the SPFILE in the database resource. In this case, you can start up the database without referring to the server parameter file by name. If the instance from which the command is issued is *not* running, then the SPFILE in the database resource must be updated manually using `srvctl modify database -d` *dbname* `-spfile` *spfile_path*.

If you omit *spfile_name*, then Oracle Database uses the platform-specific default server parameter filename. If such a file already exists on the server, then this statement overwrites it. When using a default server parameter file, you can start up the database without referring to the file by name.

Restriction on *spfile_name*

You cannot specify *spfile_name* when creating a server parameter file while connected to a PDB.

> **See Also:**
>
> - "Creating a Server Parameter File: Examples" for information on starting up the database with default and nondefault server parameter files
> - *file_specification* for the syntax of traditional and Oracle ASM filenames and ALTER DISKGROUP for information on modifying the characteristics of an Oracle ASM file
> - The appropriate operating-system-specific documentation for default parameter file names

pfile_name

Specify the traditional plain-text initialization parameter file from which you want to create a server parameter file. The traditional parameter file must reside on the server.

- If you specify *pfile_name* and the traditional parameter file does not reside in the default directory for parameter files on your operating system, then you must specify the full path.
- If you do not specify *pfile_name*, then Oracle Database looks in the default directory for parameter files on your operating system for the default parameter filename and uses that file. If that file does not exist in the expected directory, then the database returns an error.

> **Note:**
>
> In an Oracle Real Application Clusters environment, you must first combine all instance parameter files into one file before specifying that filename in this statement to create a server parameter file. For information on accomplishing this step, see *Oracle Real Application Clusters Administration and Deployment Guide*.

AS COPY

This clause applies only if the database is defined as a resource in Oracle Clusterware. By default, if you specify both the `spfile_name` and the `FROM PFILE` clause, then the `CREATE SPFILE` statement automatically updates the SPFILE in the database resource. You can specify `AS COPY` to prevent the database from updating the SPFILE in the database resource.

MEMORY

Specify `MEMORY` to create an spfile using the current system-wide parameter settings. In an Oracle RAC environment, the created file will contain the parameter settings from each instance.

Examples

Creating a Server Parameter File: Examples

The following example creates a default server parameter file from a traditional plain-text parameter file named `t_init1.ora`:

```
CREATE SPFILE
   FROM PFILE = '$ORACLE_HOME/work/t_init1.ora';
```

> **Note:**
>
> Typically you will need to specify the full path and filename for parameter files on your operating system.

When you create a default server parameter file, you subsequently start up the database using that server parameter file by using the SQL*Plus command `STARTUP` without the `PFILE` parameter, as follows:

```
STARTUP
```

The following example creates a nondefault server parameter file `s_params.ora` from a traditional plain-text parameter file named `t_init1.ora`:

```
CREATE SPFILE = 's_params.ora'
   FROM PFILE = '$ORACLE_HOME/work/t_init1.ora';
```

When you create a nondefault server parameter file, you subsequently start up the database by first creating a traditional parameter file containing the following single line:

```
spfile = 's_params.ora'
```

The name of this parameter file must comply with the naming conventions of your operating system. You then use the single-line parameter file in the `STARTUP` command. The following example shows how to start up the database, assuming that the single-line parameter file is named `new_param.ora`:

```
STARTUP PFILE=new_param.ora
```

CREATE SYNONYM

Purpose

Use the `CREATE SYNONYM` statement to create a **synonym**, which is an alternative name for a table, view, sequence, operator, procedure, stored function, package, materialized view, Java class schema object, user-defined object type, or another synonym. A synonym places a dependency on its target object and becomes invalid if the target object is changed or dropped.

Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view. However, synonyms are not a substitute for privileges on database objects. Appropriate privileges must be granted to a user before the user can use the synonym.

You can refer to synonyms in the following DML statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `FLASHBACK TABLE` , `EXPLAIN PLAN` , `LOCK TABLE`, `MERGE`, and `CALL`.

You can refer to synonyms in the following DDL statements: `AUDIT`, `NOAUDIT`, `GRANT`, `REVOKE`, and `COMMENT`.

> **See Also:**
>
> *Oracle Database Concepts* for general information on synonyms

Prerequisites

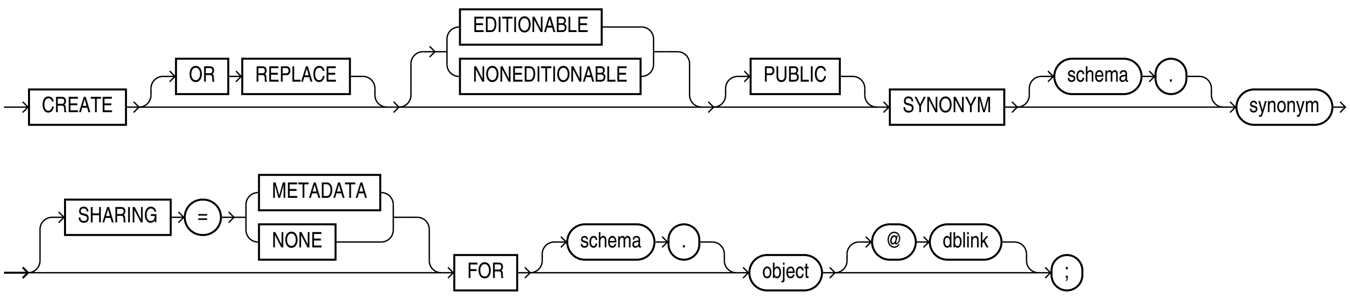
To create a private synonym in your own schema, you must have the `CREATE SYNONYM` system privilege.

To create a private synonym in another user's schema, you must have the `CREATE ANY SYNONYM` system privilege.

To create a `PUBLIC` synonym, you must have the `CREATE PUBLIC SYNONYM` system privilege.

Syntax

***create_synonym*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the synonym if it already exists. Use this clause to change the definition of an existing synonym without first dropping it.

Restriction on Replacing a Synonym

You cannot use the `OR REPLACE` clause for a type synonym that has any dependent tables or dependent valid user-defined object types.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the synonym is an editioned or noneditioned object if editioning is enabled for the schema object type `SYNONYM` in *`schema`*. For private synonyms, the default is `EDITIONABLE`. For public synonyms, the default is `NONEDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

PUBLIC

Specify `PUBLIC` to create a public synonym. Public synonyms are accessible to all users. However each user must have appropriate privileges on the underlying object in order to use the synonym.

When resolving references to an object, Oracle Database uses a public synonym only if the object is not prefaced by a schema and is not followed by a database link.

If you omit this clause, then the synonym is private. A private synonym name must be unique in its schema. A private synonym is accessible to users other than the owner only if those users have appropriate privileges on the underlying database object and specify the schema along with the synonym name.

Notes on Public Synonyms

The following notes apply to public synonyms:

- If you create a public synonym and it subsequently has dependent tables or dependent valid user-defined object types, then you cannot create another database object of the same name as the synonym in the same schema as the dependent objects.

- Take care not to create a public synonym with the same name as an existing schema. If you do so, then all PL/SQL units that use that name will be invalidated.

schema

Specify the schema to contain the synonym. If you omit `schema`, then Oracle Database creates the synonym in your own schema. You cannot specify a schema for the synonym if you have specified `PUBLIC`.

synonym

Specify the name of the synonym to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

> **Note:**
>
> The maximum length of a synonym name is subject to the following rules:
>
> - If the `COMPATIBLE` initialization parameter is set to a value of `12.2` or higher, then the maximum length of a synonym name is 128 bytes. The database will allow you to create and drop synonyms of length 129 to 4000 bytes. However, unless these longer synonym names represent a Java name they will not work in any other SQL command.
> - If the `COMPATIBLE` initialization parameter is set to a value lower than `12.2`, then the maximum length of a synonym name is 30 bytes. The database will allow you to create and drop synonyms of length 31 to 128 bytes. However, unless these longer synonym names represent a Java name they will not work in any other SQL command.
>
> The longer synonym names are transformed into obscure shorter strings for storage in the data dictionary.

> **See Also:**
>
> "CREATE SYNONYM: Examples" and "Oracle Database Resolution of Synonyms: Example"

SHARING

This clause applies only when creating a synonym in an application root. This type of synonym is called an application common object and it can be shared with the application PDBs that belong to the application root. To determine how the synonym is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the synonym's metadata, but its data is unique to each container. This type of synonym is referred to as a **metadata-linked application common object**.
- `NONE` - The synonym is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the synonym. If the

DEFAULT_SHARING initialization parameter does not have a value, then the default is METADATA.

You cannot change the sharing attribute of a synonym after it is created.

> **See Also:**
>
> - *Oracle Database Reference* for more information on the DEFAULT_SHARING initialization parameter
> - *Oracle Database Administrator's Guide* for complete information on creating application common objects

FOR Clause

Specify the object for which the synonym is created. The schema object for which you are creating the synonym can be of the following types:

- Table or object table
- View or object view
- Sequence
- Stored procedure, function, or package
- Materialized view
- Java class schema object
- User-defined object type
- Synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restriction on the FOR Clause

The schema object cannot be contained in a package.

schema

Specify the schema in which the object resides. If you do not qualify object with *schema*, then the database assumes that the schema object is in your own schema.

If you are creating a synonym for a procedure or function on a remote database, then you must specify *schema* in this CREATE statement. Alternatively, you can create a local public synonym on the database where the object resides. However, the database link must then be included in all subsequent calls to the procedure or function.

dblink

You can specify a complete or partial database link to create a synonym for a schema object on a remote database where the object is located. If you specify *dblink* and omit *schema*, then the synonym refers to an object in the schema specified by the database link. Oracle recommends that you specify the schema containing the object in the remote database.

If you omit *dblink*, then Oracle Database assumes the object is located on the local database.

Restriction on Database Links

You cannot specify *dblink* for a Java class synonym.

> **See Also:**
>
> - "References to Objects in Remote Databases " for more information on referring to database links
> - CREATE DATABASE LINK for more information on creating database links

Examples

CREATE SYNONYM: Examples

To define the synonym `offices` for the table `locations` in the schema `hr`, issue the following statement:

```
CREATE SYNONYM offices
   FOR hr.locations;
```

To create a `PUBLIC` synonym for the `employees` table in the schema `hr` on the `remote` database, you could issue the following statement:

```
CREATE PUBLIC SYNONYM emp_table
   FOR hr.employees@remote.us.example.com;
```

A synonym may have the same name as the underlying object, provided the underlying object is contained in another schema.

Oracle Database Resolution of Synonyms: Example

Oracle Database attempts to resolve references to objects at the schema level before resolving them at the `PUBLIC` synonym level. For example, the schemas `oe` and `sh` both contain tables named `customers`. In the next example, user `SYSTEM` creates a `PUBLIC` synonym named `customers` for `oe.customers`:

```
CREATE PUBLIC SYNONYM customers FOR oe.customers;
```

If the user `sh` then issues the following statement, then the database returns the count of rows from `sh.customers`:

```
SELECT COUNT(*) FROM customers;
```

To retrieve the count of rows from `oe.customers`, the user `sh` must preface `customers` with the schema name. (The user `sh` must have select permission on `oe.customers` as well.)

```
SELECT COUNT(*) FROM oe.customers;
```

If the user `hr`'s schema does not contain an object named `customers`, and if `hr` has select permission on `oe.customers`, then `hr` can access the `customers` table in `oe`'s schema by using the public synonym `customers`:

```
SELECT COUNT(*) FROM customers;
```

CREATE TABLE

Purpose

Use the CREATE TABLE statement to create one of the following types of tables:

- A **relational table**, which is the basic structure to hold user data.
- An **object table**, which is a table that uses an object type for a column definition. An object table is explicitly defined to hold object instances of a particular type.

You can also create an object type and then use it in a column when creating a relational table.

Tables are created with no data unless a subquery is specified. You can add rows to a table with the INSERT statement. After creating a table, you can define additional columns, partitions, and integrity constraints with the ADD clause of the ALTER TABLE statement. You can change the definition of an existing column or partition with the MODIFY clause of the ALTER TABLE statement.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* and CREATE TYPE for more information about creating objects
> - ALTER TABLE and DROP TABLE for information on modifying and dropping tables

Prerequisites

To create a **relational table** in your own schema, you must have the CREATE TABLE system privilege. To create a table in another user's schema, you must have the CREATE ANY TABLE system privilege. Also, the owner of the schema to contain the table must have either space quota on the tablespace to contain the table or the UNLIMITED TABLESPACE system privilege.

In addition to these table privileges, to create an object table or a relational table with an object type column, the owner of the table must have the EXECUTE object privilege in order to access all types referenced by the table, or you must have the EXECUTE ANY TYPE system privilege. These privileges must be granted explicitly and not acquired through a role.

Additionally, if the table owner intends to grant access to the table to other users, then the owner must have been granted the EXECUTE object privilege on the referenced types WITH GRANT OPTION, or have the EXECUTE ANY TYPE system privilege WITH ADMIN OPTION. Without these privileges, the table owner has insufficient privileges to grant access to the table to other users.

To enable a unique or primary key constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle Database creates an index on the columns of the unique or primary key in the schema containing the table.

To specify an edition in the *evaluation_edition_clause* or the *unusable_editions_clause*, you must have the USE privilege on the edition.

To specify the *zonemap_clause*, you must have the permissions necessary to create a zone map. Refer to the "Prerequisites" section in the documentation on CREATE MATERIALIZED ZONEMAP.

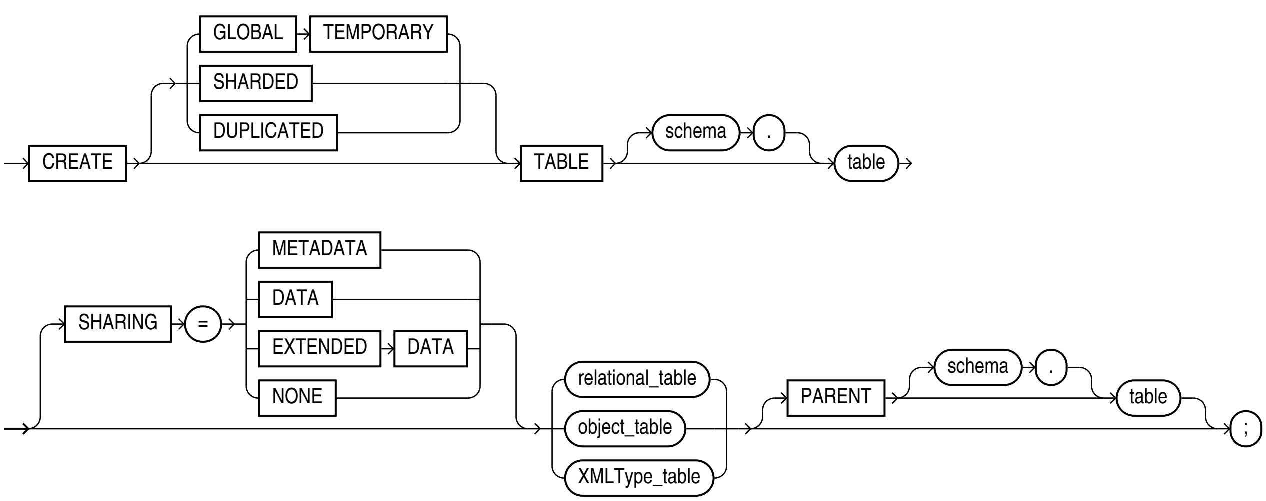
To create an **external table**, you must have the required read and write operating system privileges on the appropriate operating system directories. You must have the READ object privilege on the database directory object corresponding to the operating system directory in which the external data resides. You must also have the WRITE object privilege on the database directory in which the files will reside if you specify a log file or bad file in the *opaque_format_spec* or if you unload data into an external table from a database table by specifying the AS *subquery* clause.

To create an **XMLType table** in a different database schema from your own, you must have not only privilege CREATE ANY TABLE but also privilege CREATE ANY INDEX. This is because a unique index is created on column OBJECT_ID when you create the table. Column OBJECT_ID stores a system-generated object identifier.

> **See Also:**
>
> - CREATE INDEX
> - *Oracle Database Administrator's Guide* for more information about the privileges required to create tables using types

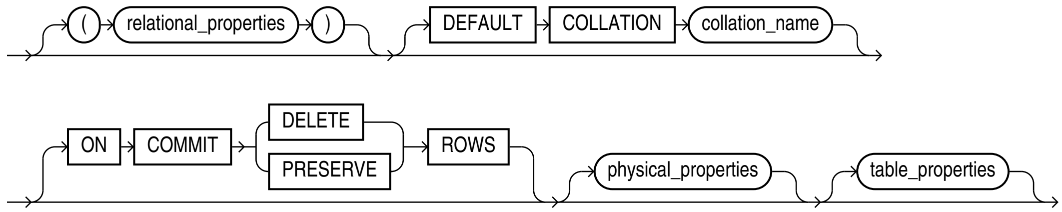
Syntax

***create_table*::=**



(*relational_table*::=, *object_table*::=, *XMLType_table*::=)

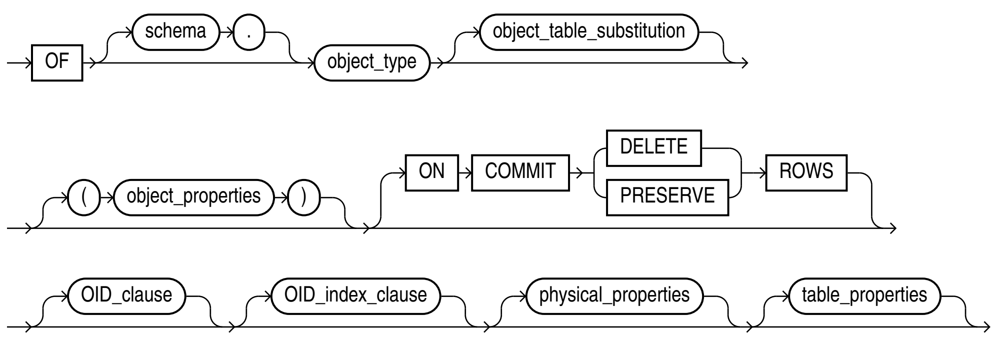
***relational_table*::=**



> **Note:**
>
> Each of the clauses following the table name is optional for any given relational table. However, for every table you must at least specify either column names and data types using the `relational_properties` clause or an `AS subquery` clause using the `table_properties` clause.

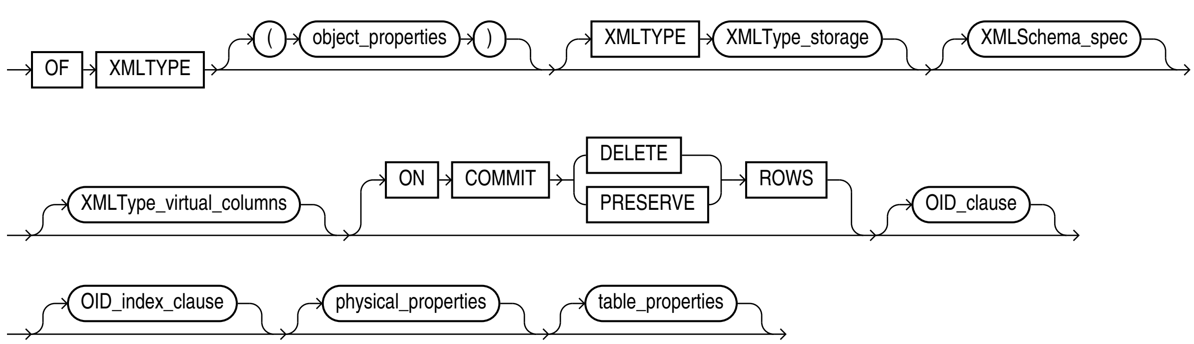
(*relational_properties*::=, *physical_properties*::=, *table_properties*::=)

***object_table*::=**



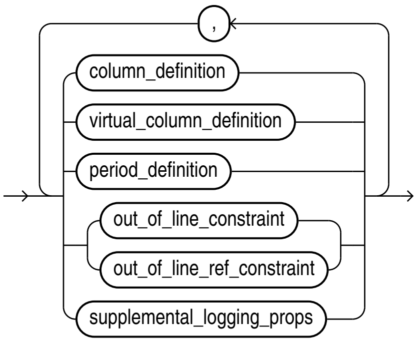
(*object_table_substitution*::=, *object_properties*::=, *oid_clause*::=, *oid_index_clause*::=, *physical_properties*::=, *table_properties*::=)

***XMLType_table*::=**



(*XMLType_storage*::=, *XMLSchema_spec*::=, *XMLType_virtual_columns*::=, *oid_clause*::=, *oid_index_clause*::=, *physical_properties*::=, *table_properties*::=)

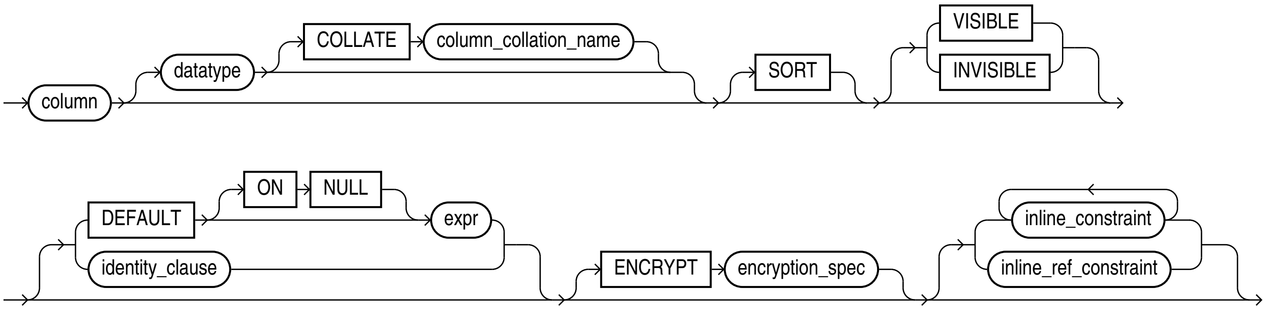
***relational_properties*::=**



> **Note:**
>
> You can specify these clauses in any order with the following exception: You must specify at least one `column_definition` or `virtual_column_definition` before you specify `period_definition`. You can specify `period_definition` only once.

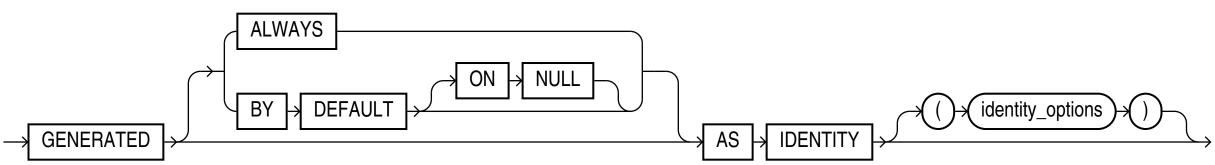
(*column_definition*::=, *virtual_column_definition*::=, *period_definition*::=, *constraint*::=, *supplemental_logging_props*::=)

***column_definition*::=**

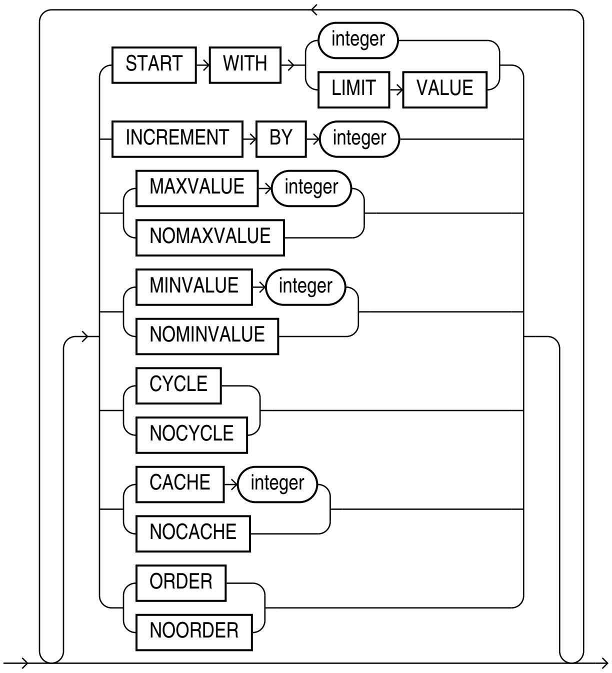


(*identity_clause*::=, *encryption_spec*::=, *constraint*::=)

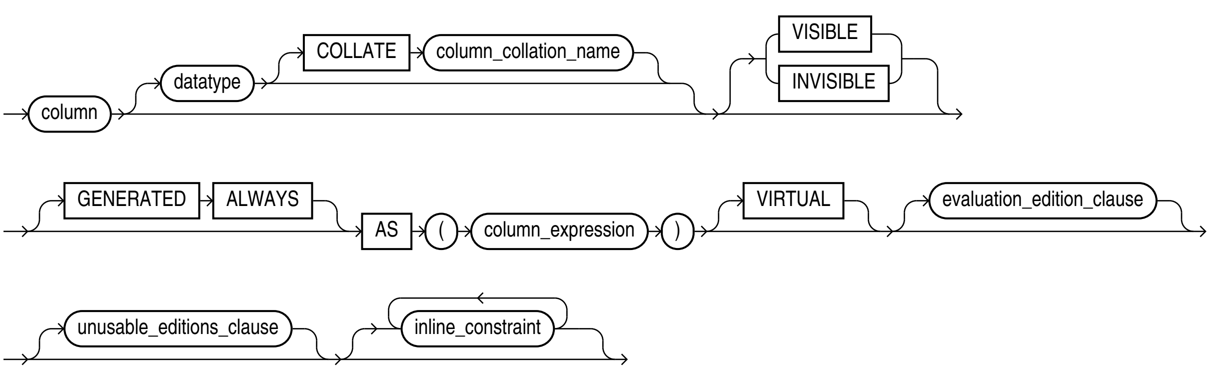
***identity_clause*::=**



***identity_options*::=**

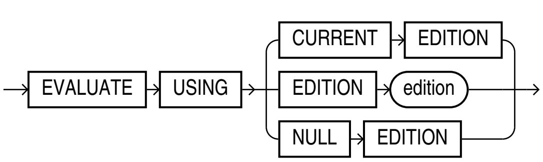


***virtual_column_definition*::=**

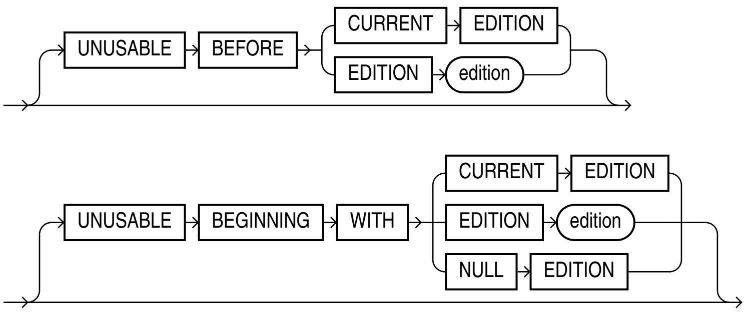


(*evaluation_edition_clause*::=, *unusable_editions_clause*::=, *constraint*::=)

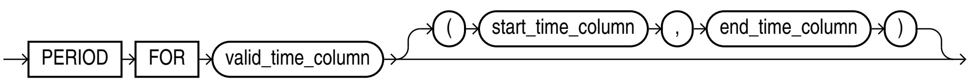
***evaluation_edition_clause*::=**



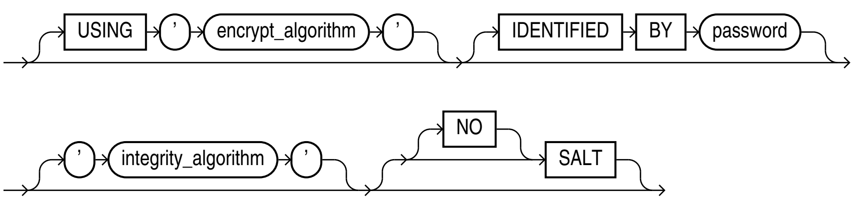
***unusable_editions_clause*::=**



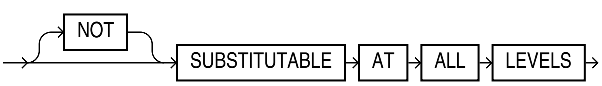
***period_definition*::=**



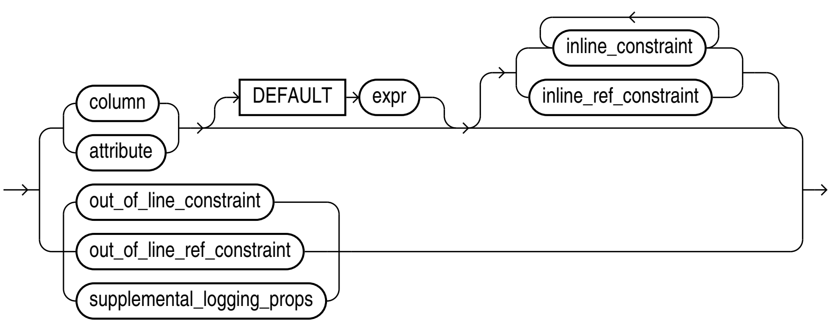
***encryption_spec*::=**



***object_table_substitution*::=**

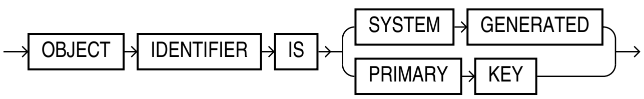


***object_properties*::=**

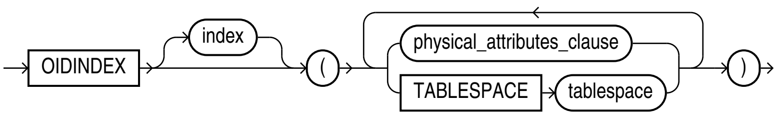


(*constraint::=*, *supplemental_logging_props::=*)

***oid_clause*::=**

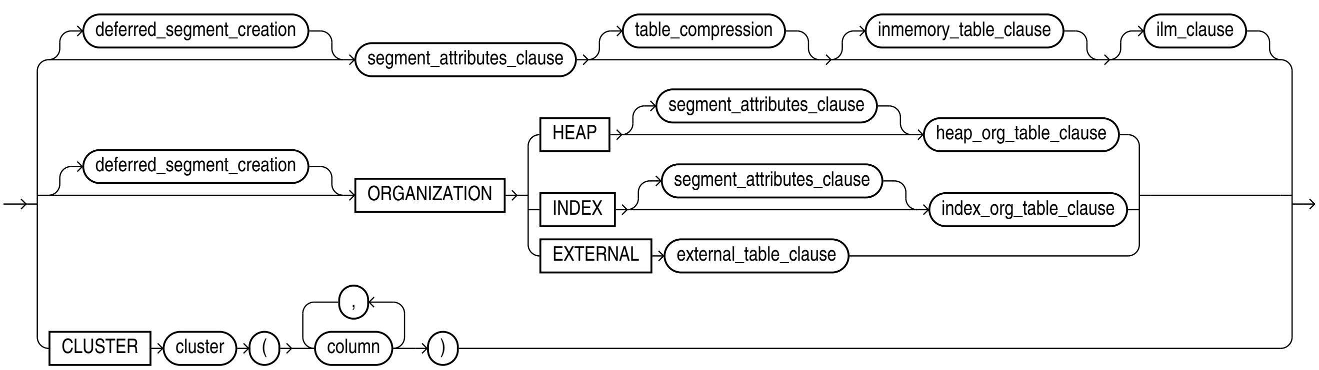


***oid_index_clause*::=**



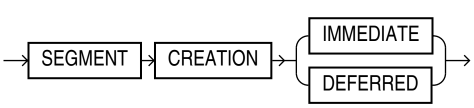
(*physical_attributes_clause::=*)

***physical_properties*::=**

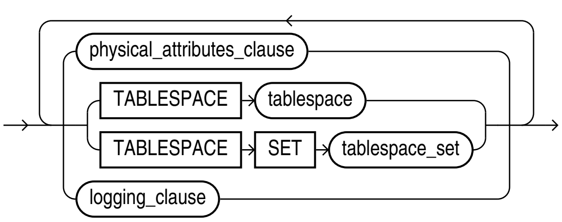


(*deferred_segment_creation::=*, *segment_attributes_clause::=*, *table_compression::=*, *inmemory_table_clause::=*, *ilm_clause::=*, *heap_org_table_clause::=*, *index_org_table_clause::=*, *external_table_clause::=*)

***deferred_segment_creation*::=**

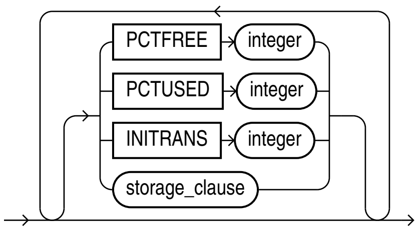


***segment_attributes_clause*::=**



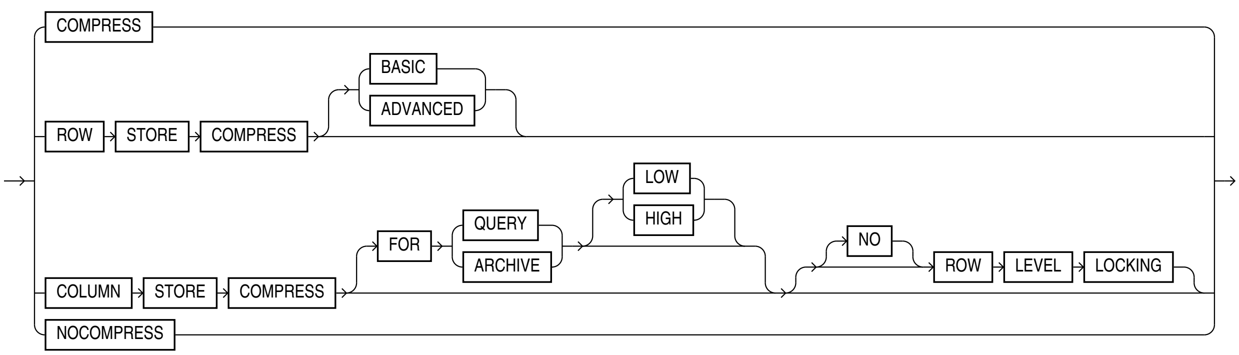
(*physical_attributes_clause*::=, *logging_clause*::=)

***physical_attributes_clause*::=**

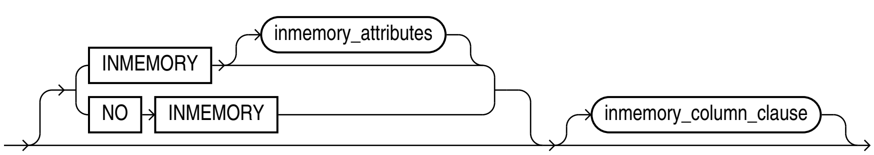


(*storage_clause*::=)

***table_compression*::=**

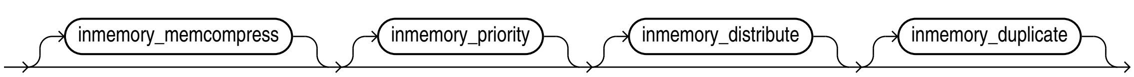


***inmemory_table_clause*::=**



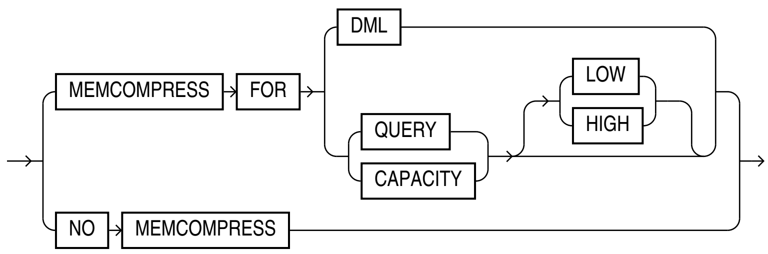
(*inmemory_attributes*::=, *inmemory_column_clause*::=)

***inmemory_attributes*::=**

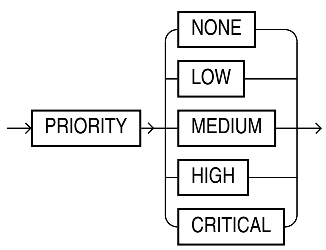


(*inmemory_memcompress*::=, *inmemory_priority*::=, *inmemory_distribute*::=, *inmemory_duplicate*::=)

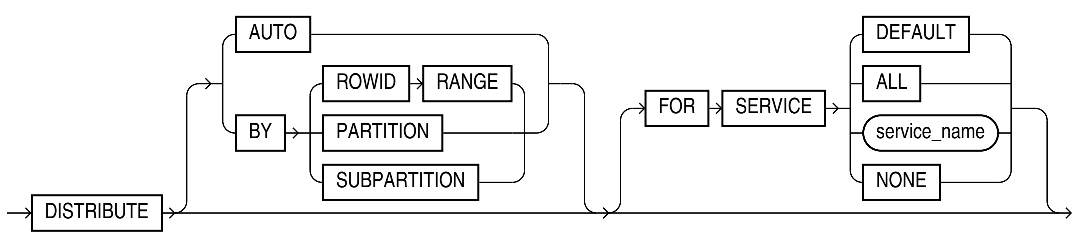
***inmemory_memcompress*::=**



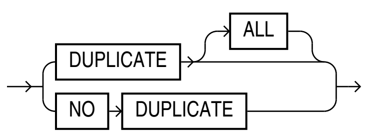
***inmemory_priority*::=**



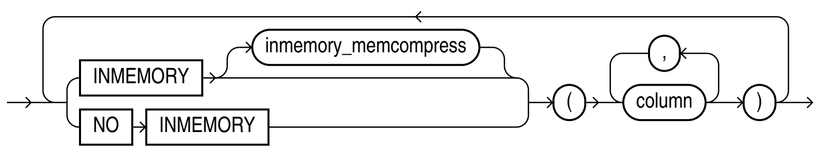
***inmemory_distribute*::=**



***inmemory_duplicate*::=**

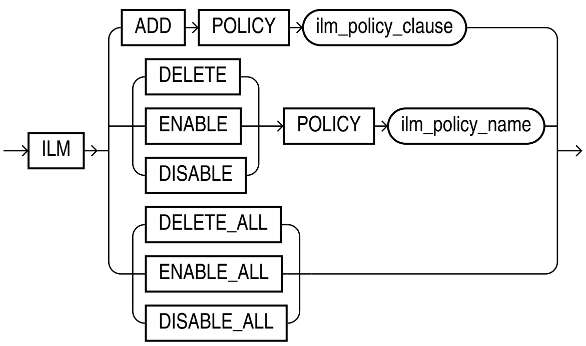


***inmemory_column_clause*::=**

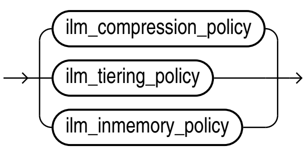


(*inmemory_memcompress::=*)

ilm_clause::=

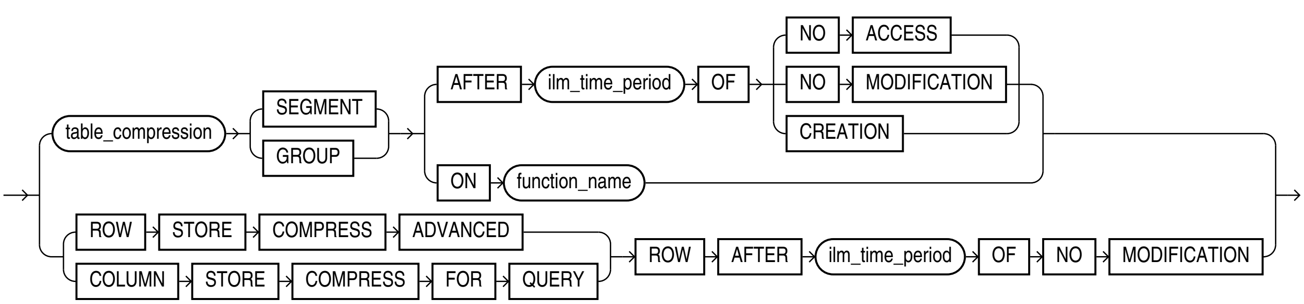


ilm_policy_clause::=



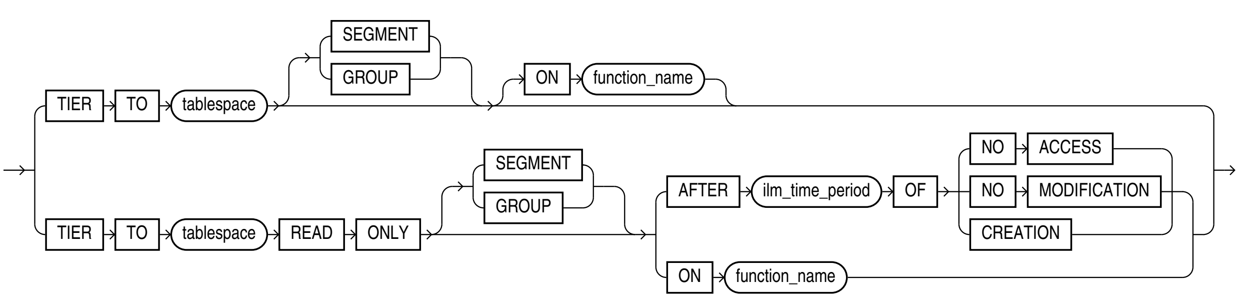
(*ilm_compression_policy::=*, *ilm_tiering_policy::=*, *ilm_inmemory_policy::=*)

ilm_compression_policy::=



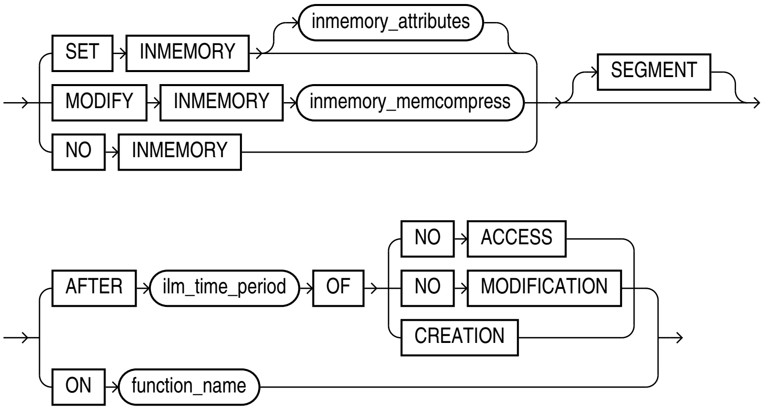
(*table_compression::=*, *ilm_time_period::=*)

ilm_tiering_policy::=

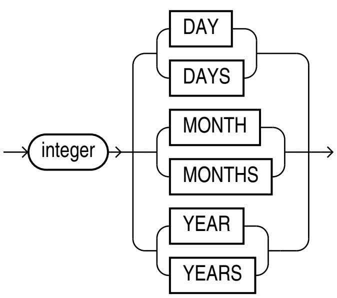


(*ilm_time_period*::=)

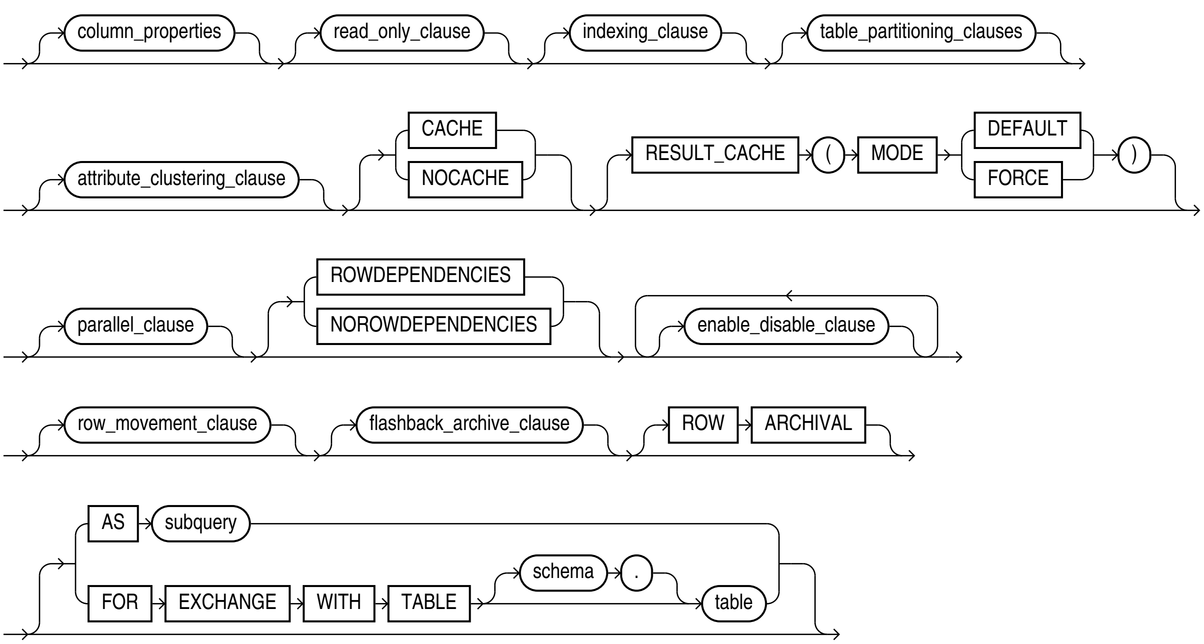
***ilm_inmemory_policy*::=**



***ilm_time_period*::=**

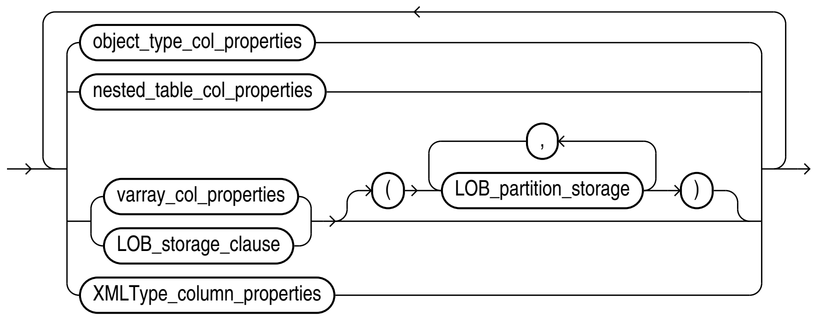


***table_properties*::=**



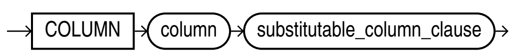
(*column_properties*::=, *read_only_clause*::=, *indexing_clause*::=, *table_partitioning_clauses*::=, *attribute_clustering_clause*::=, *parallel_clause*::=, *enable_disable_clause*::=, *row_movement_clause*::=, *flashback_archive_clause*::= , *subquery*::=)

***column_properties*::=**

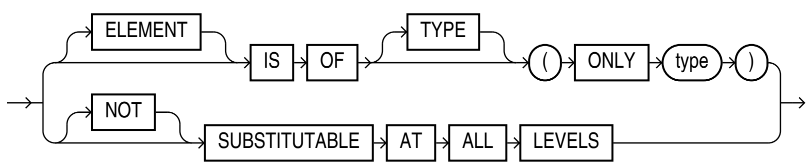


(*object_type_col_properties*::=, *nested_table_col_properties*::=, *varray_col_properties*::=, *LOB_storage_clause*::=, *LOB_partition_storage*::=, *XMLType_column_properties*::=)

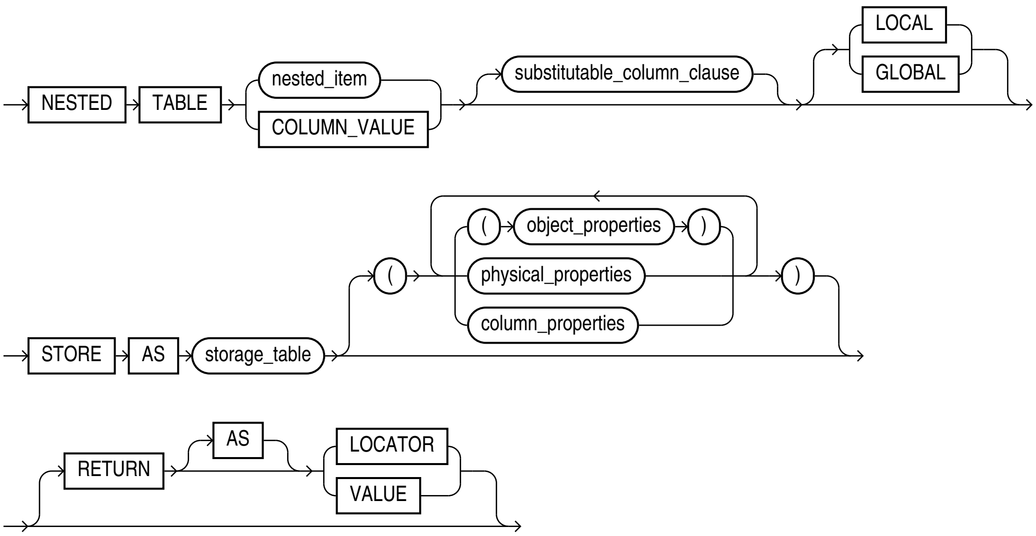
***object_type_col_properties*::=**



***substitutable_column_clause*::=**

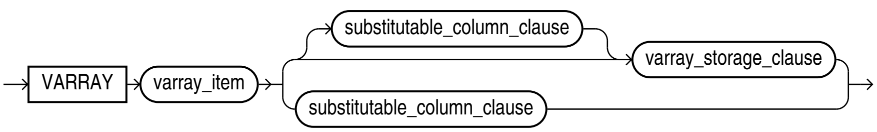


***nested_table_col_properties*::=**



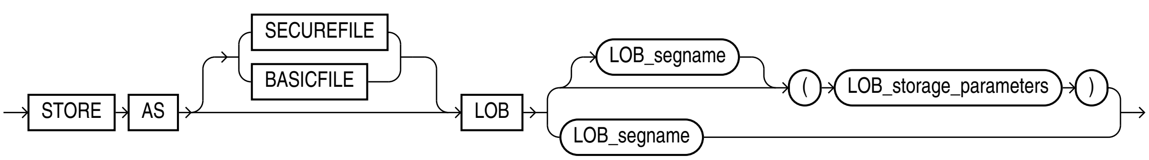
(*substitutable_column_clause::=*, *object_properties::=*, *physical_properties::=*, *column_properties::=*)

***varray_col_properties*::=**



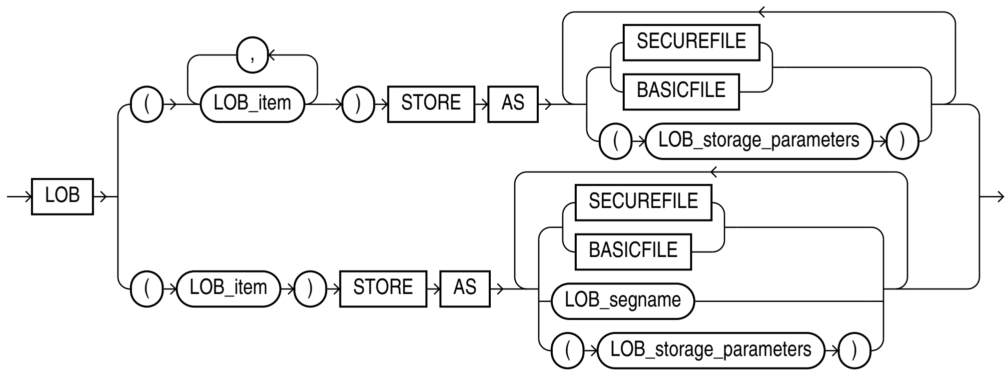
(*substitutable_column_clause::=*, *varray_storage_clause::=*)

***varray_storage_clause*::=**



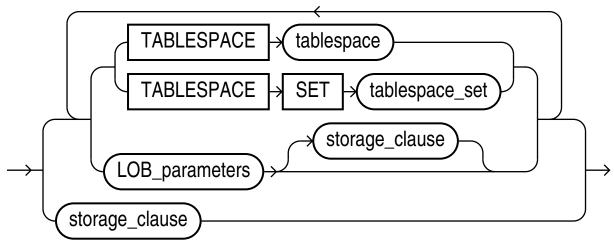
(*LOB_parameters::=*)

***LOB_storage_clause*::=**



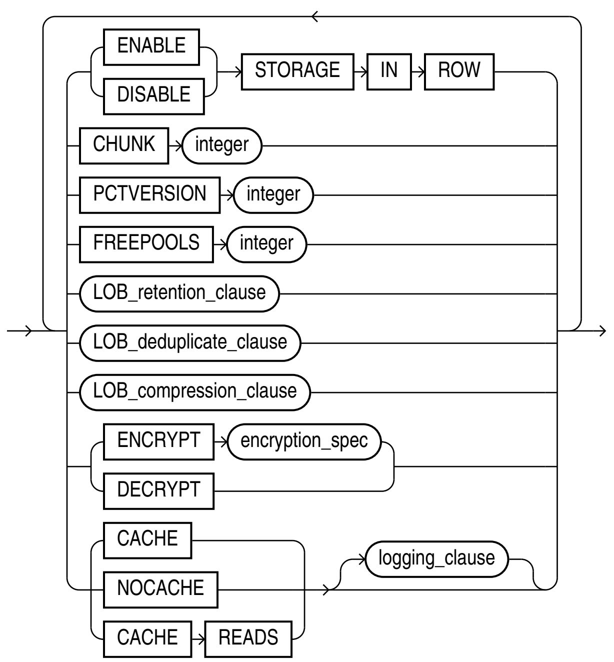
(*LOB_storage_parameters::=*)

***LOB_storage_parameters*::=**



(*LOB_parameters::=*, *storage_clause::=*)

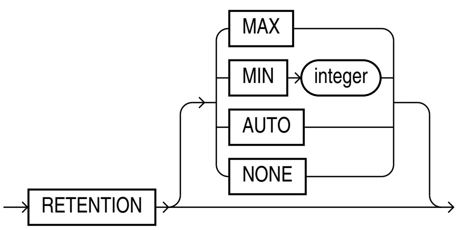
***LOB_parameters*::=**



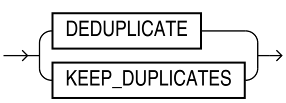
(*LOB_deduplicate_clause*::=, *LOB_compression_clause*::=, *encryption_spec*::=, *logging_clause*::=)

> **Note:**
>
> Several of the LOB parameters are no longer needed if you are using SecureFiles for LOB storage. Refer to *LOB_storage_parameters* for more information.

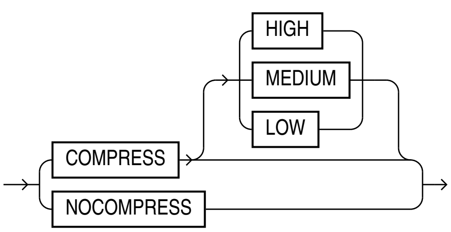
***LOB_retention_clause*::=**



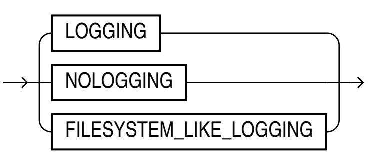
***LOB_deduplicate_clause*::=**



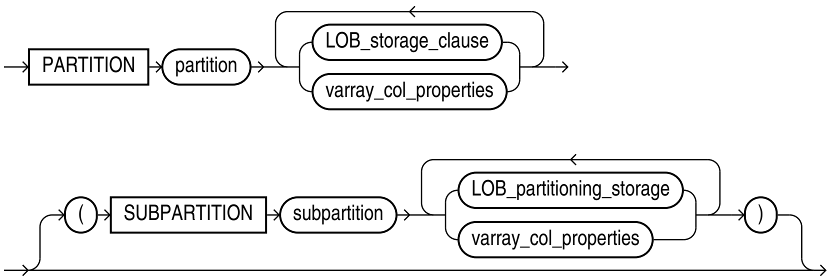
***LOB_compression_clause*::=**



***logging_clause*::=**

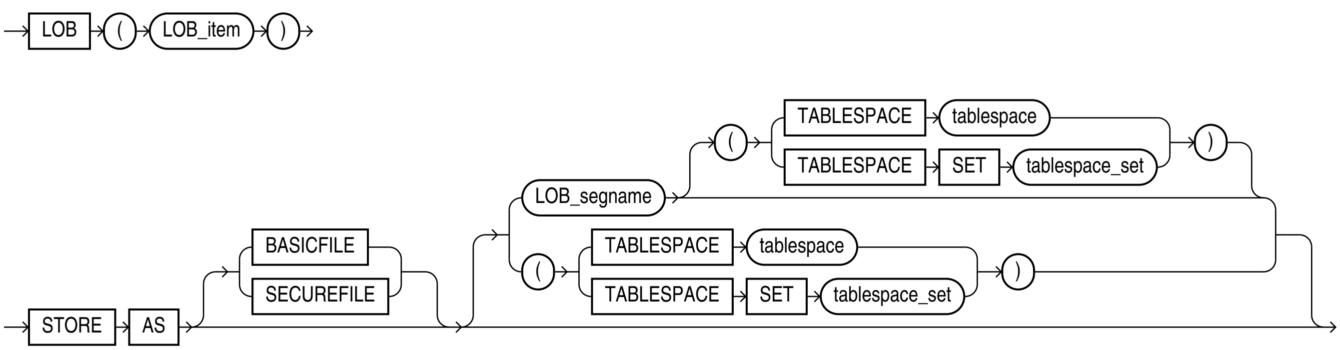


***LOB_partition_storage*::=**

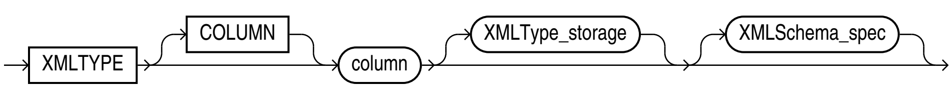


(*LOB_storage_clause*::=, *varray_col_properties*::=, *LOB_partitioning_storage*::=)

***LOB_partitioning_storage*::=**

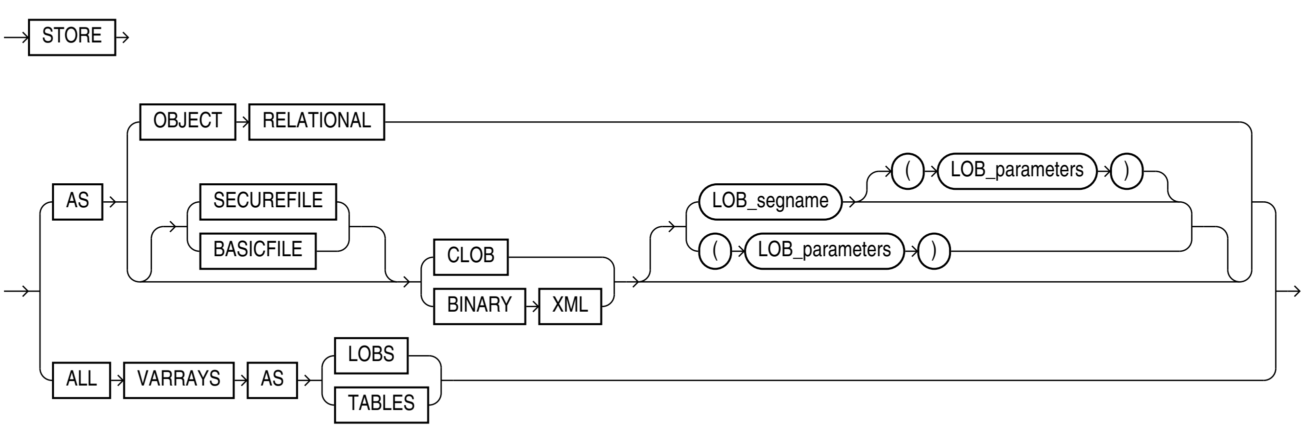


***XMLType_column_properties*::=**



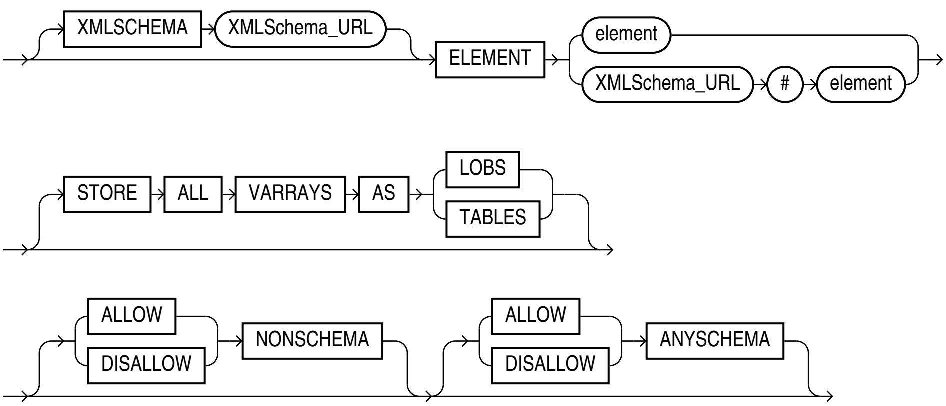
(*XMLType_storage*::=, *XMLSchema_spec*::=)

***XMLType_storage*::=**

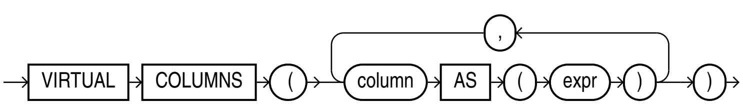


(*LOB_parameters*::=)

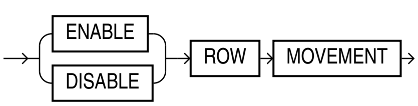
***XMLSchema_spec*::=**



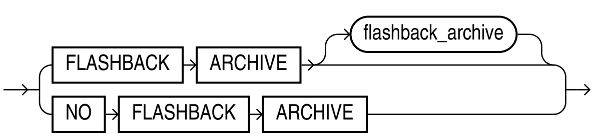
***XMLType_virtual_columns*::=**



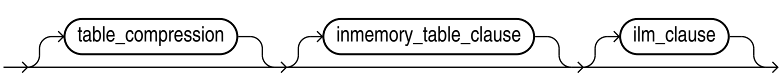
***row_movement_clause*::=**



***flashback_archive_clause*::=**

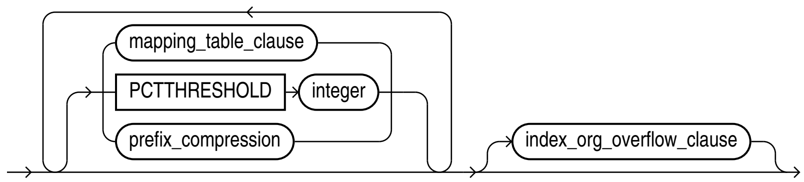


***heap_org_table_clause*::=**



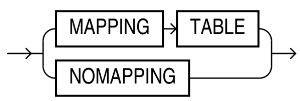
(*table_compression*::=, *inmemory_table_clause*::=, *ilm_clause*::=)

***index_org_table_clause*::=**

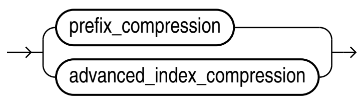


(*mapping_table_clauses*::=, *prefix_compression*::=, *index_org_overflow_clause*::=)

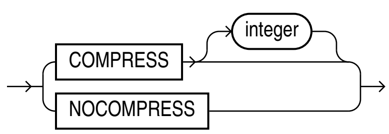
***mapping_table_clauses*::=**



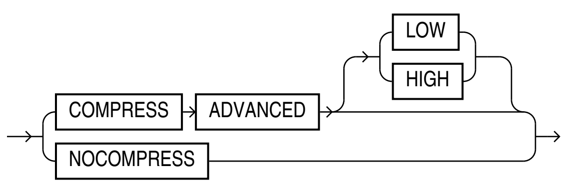
***index_compression*::=**



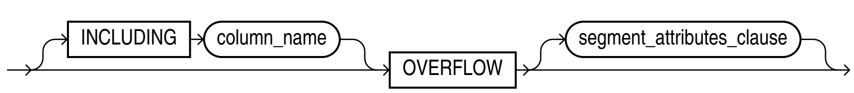
***prefix_compression*::=**



***advanced_index_compression*::=**

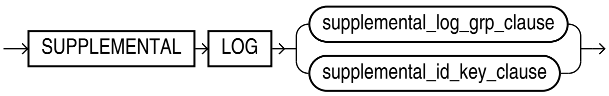


***index_org_overflow_clause*::=**

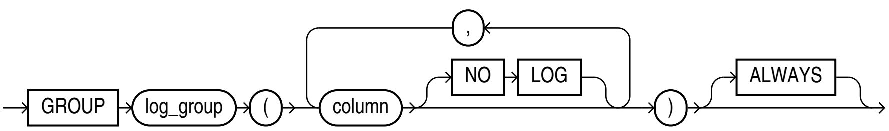


(*segment_attributes_clause::=*)

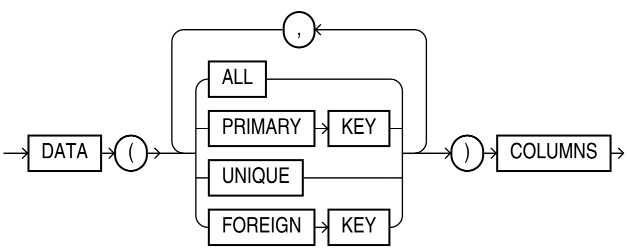
***supplemental_logging_props*::=**



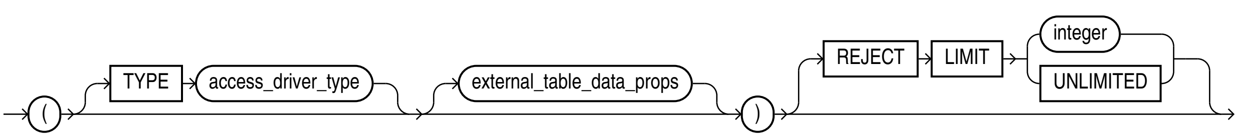
***supplemental_log_grp_clause*::=**



***supplemental_id_key_clause*::=**

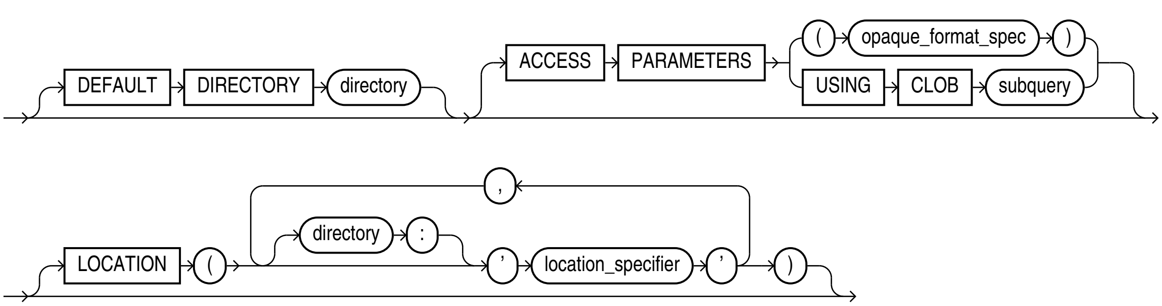


***external_table_clause*::=**



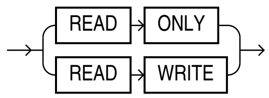
(*external_table_data_props::=*)

***external_table_data_props*::=**

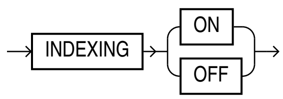


(*opaque_format_spec*: This clause specifies all access parameters for the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers. See *Oracle Database Utilities* for descriptions of these parameters.)

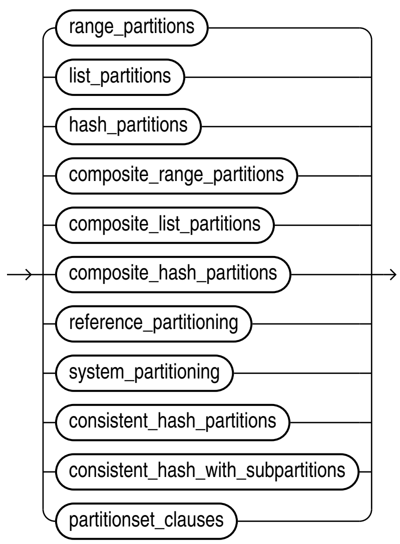
***read_only_clause*::=**



***indexing_clause*::=**

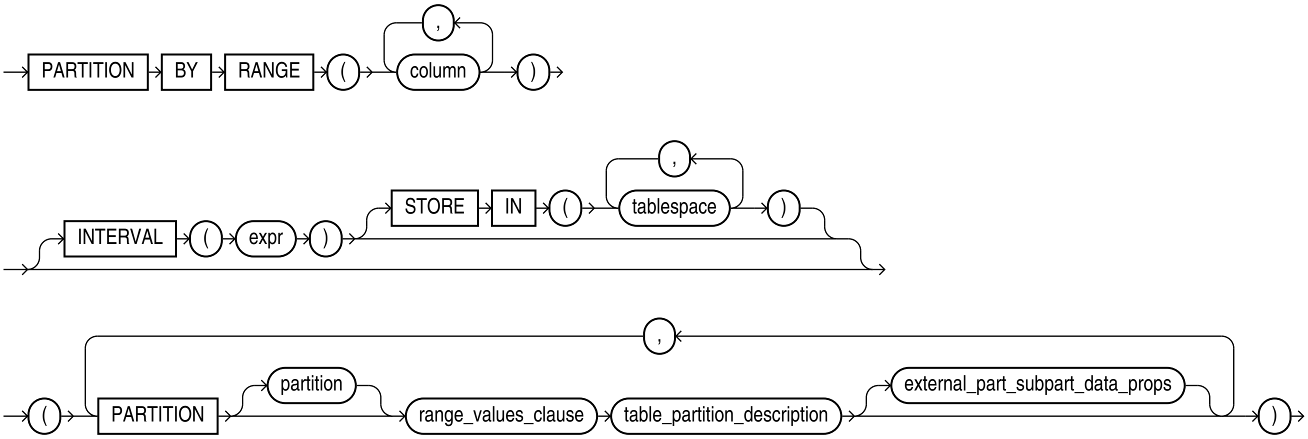


***table_partitioning_clauses*::=**



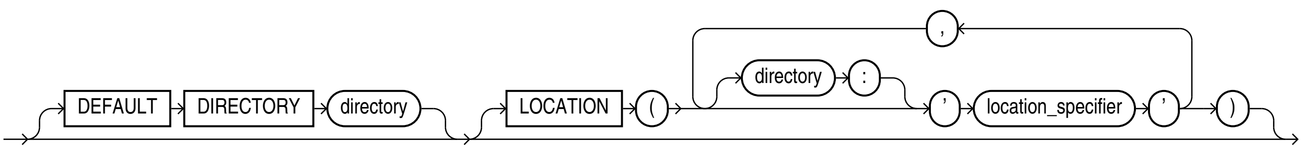
(*range_partitions*::=, *list_partitions*::=, *hash_partitions*::=, *composite_range_partitions*::=, *composite_list_partitions*::= *composite_hash_partitions*::=, *reference_partitioning*::=, *system_partitioning*::=, *consistent_hash_partitions*::=, *consistent_hash_with_subpartitions*::=, *partitionset_clauses*::=)

***range_partitions*::=**

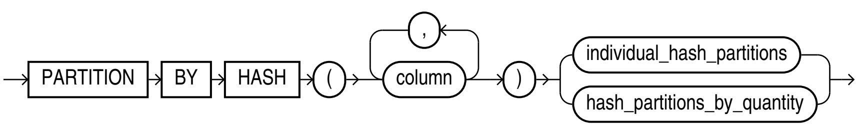


(*range_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=)

***external_part_subpart_data_props*::=**

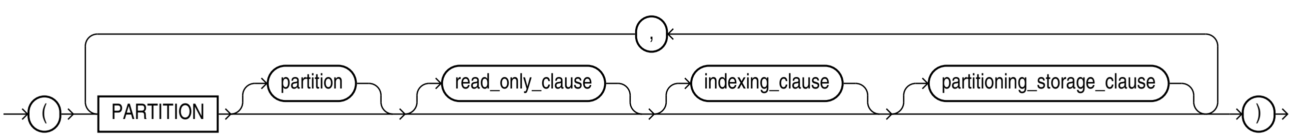


***hash_partitions*::=**



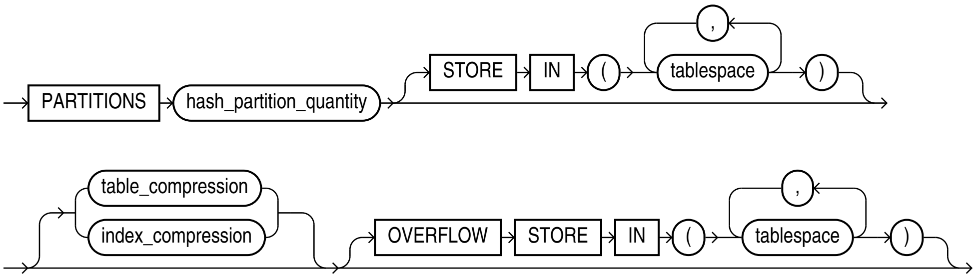
(*individual_hash_partitions*::=, *hash_partitions_by_quantity*::=)

***individual_hash_partitions*::=**



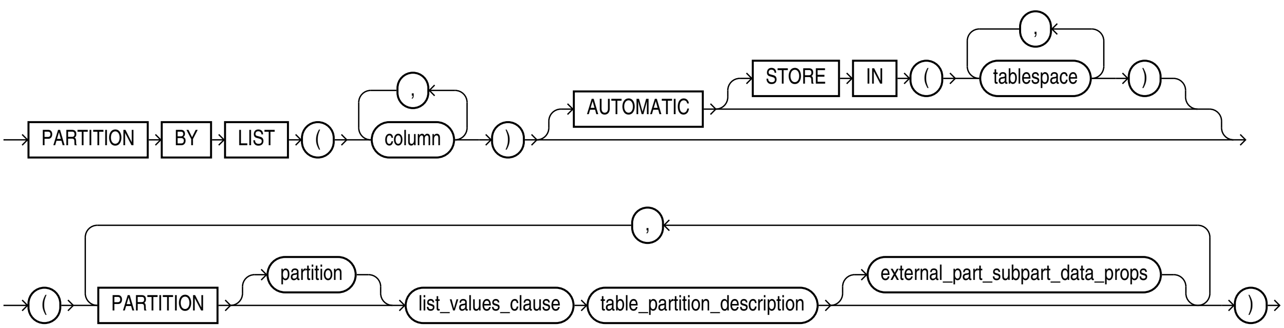
(*read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=)

***hash_partitions_by_quantity*::=**



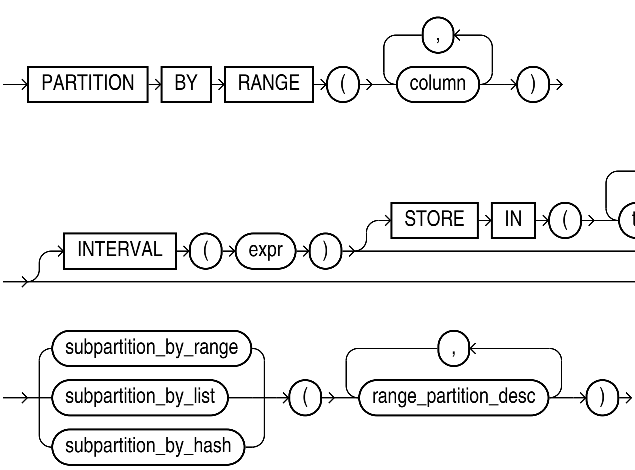
(*table_compression*::=, *index_compression*::=)

***list_partitions*::=**



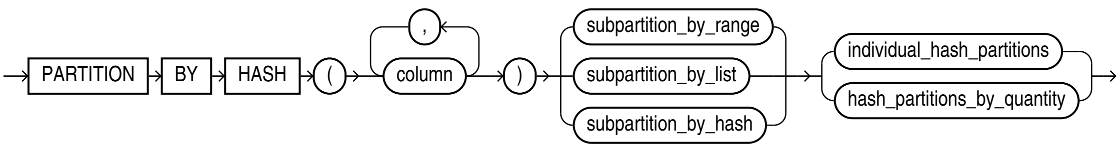
(*list_values_clause*::=, *table_partition_description*::=, *external_part_subpart_data_props*::=)

***composite_range_partitions*::=**



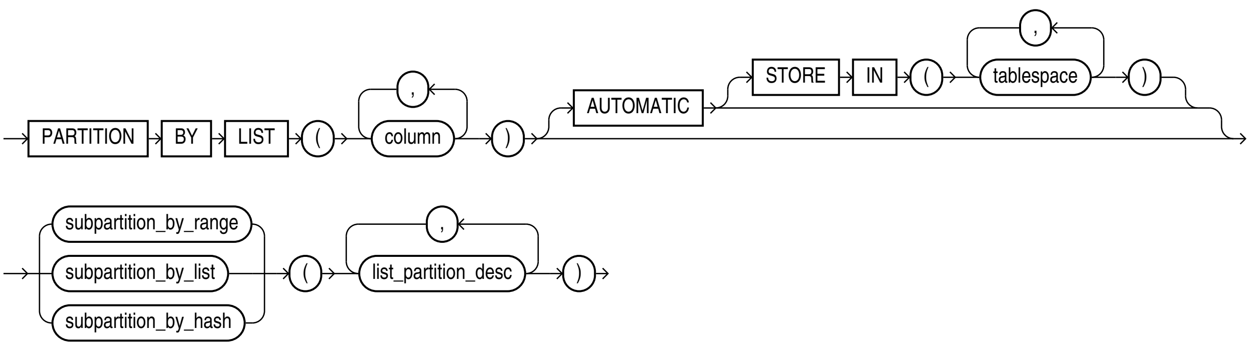
(*subpartition_by_range*::=. *subpartition_by_list*::=, *subpartition_by_hash*::=, *range_partition_desc*::=)

***composite_hash_partitions*::=**



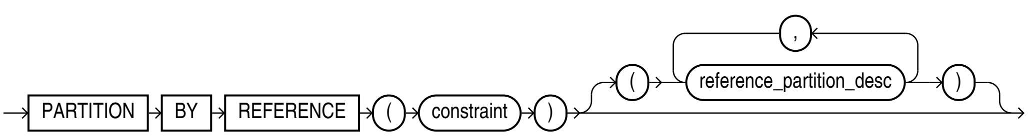
(*subpartition_by_range*::=, *subpartition_by_list*::=, *subpartition_by_hash*::=, *individual_hash_partitions*::=, *hash_partitions_by_quantity*::=)

***composite_list_partitions*::=**



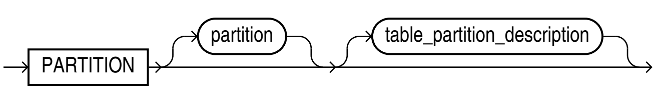
(*subpartition_by_range*::=. *subpartition_by_list*::=, *subpartition_by_hash*::=, *list_partition_desc*::=)

***reference_partitioning*::=**



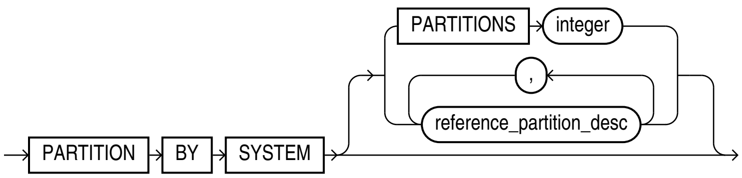
(*constraint*::=, *reference_partition_desc*::=)

***reference_partition_desc*::=**



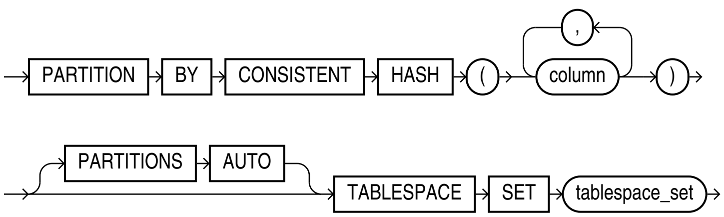
(*table_partition_description*::=)

***system_partitioning*::=**

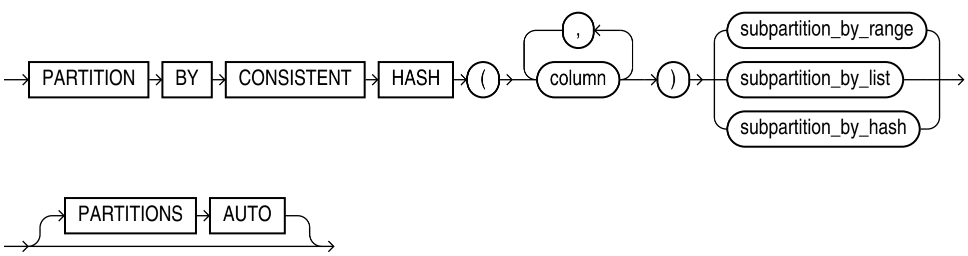


(*reference_partition_desc*::=)

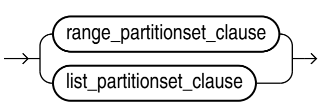
***consistent_hash_partitions*::=**



***consistent_hash_with_subpartitions*::=**

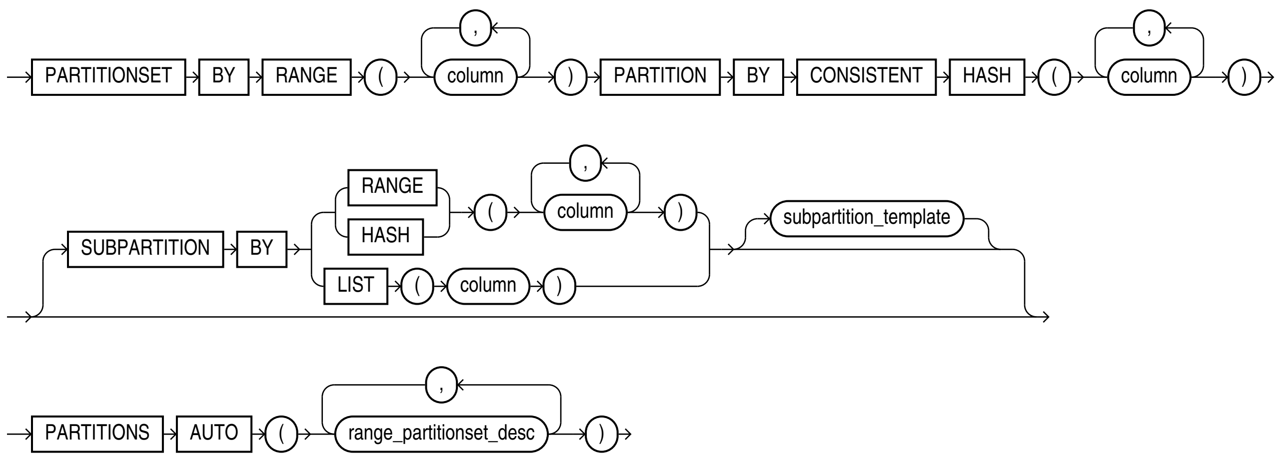


***partitionset_clauses*::=**

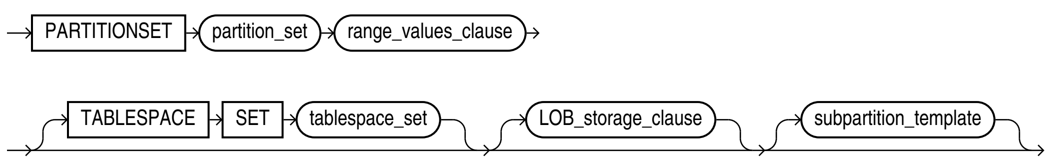


(*range_partitionset_clause*::=, *list_partitionset_clause*::=

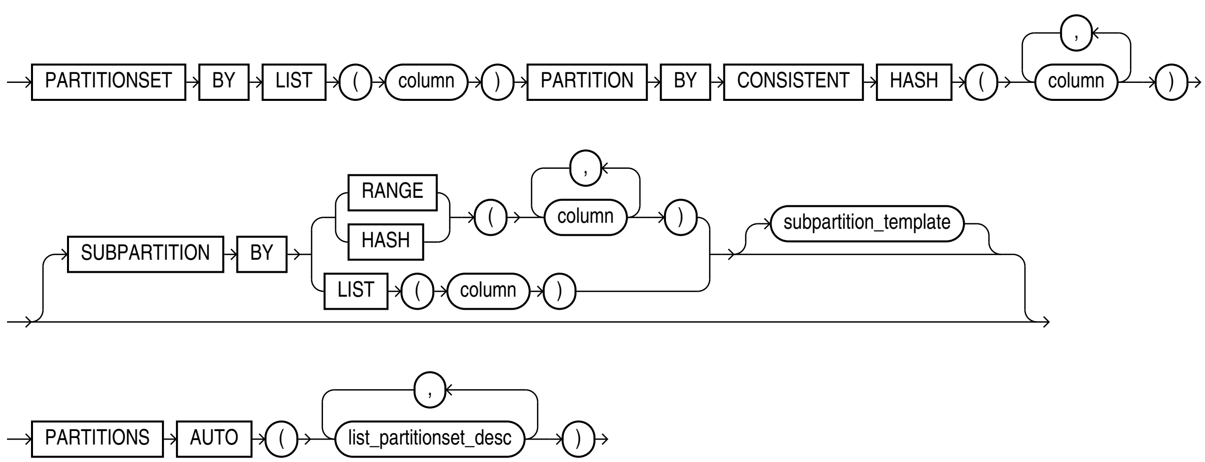
***range_partitionset_clause*::=**



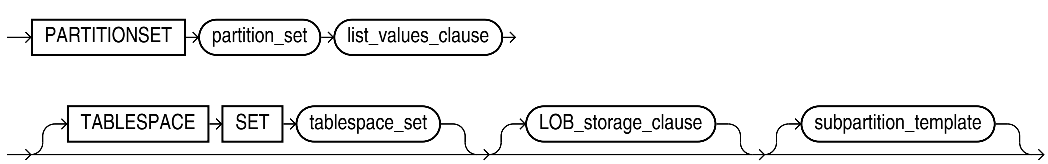
***range_partitionset_desc*::=**



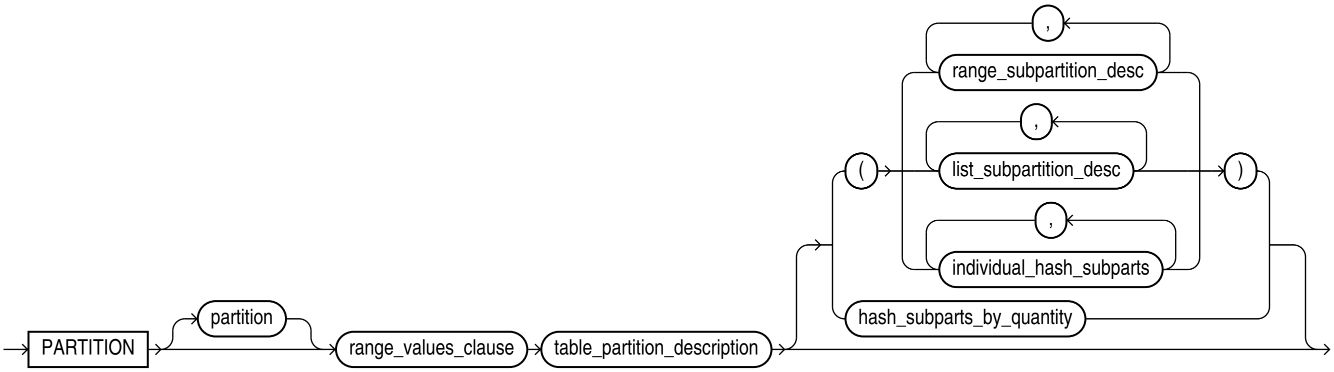
***list_partitionset_clause*::=**



***list_partitionset_desc*::=**

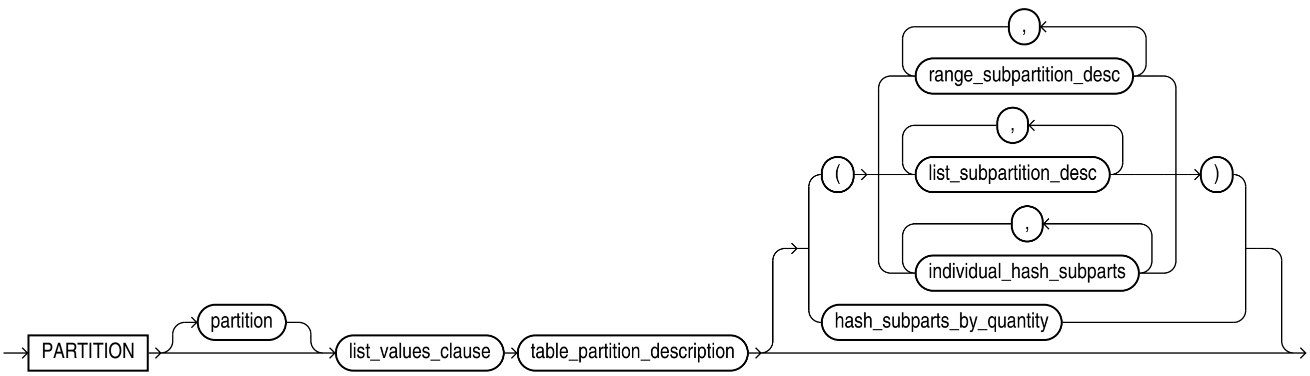


***range_partition_desc*::=**



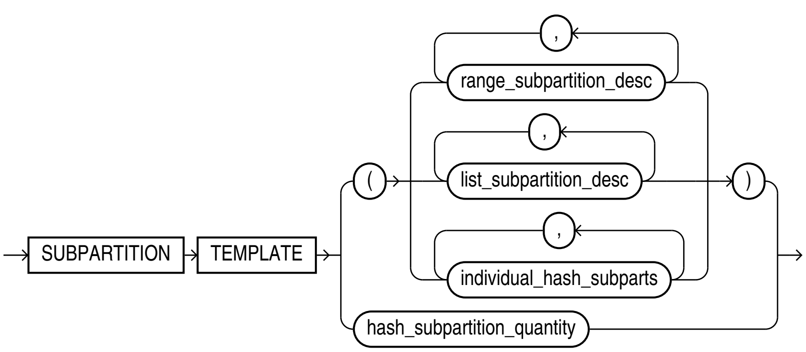
(*range_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=)

***list_partition_desc*::=**



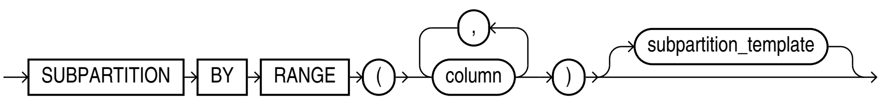
(*list_values_clause*::=, *table_partition_description*::=, *range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=)

***subpartition_template*::=**



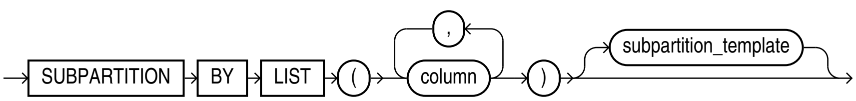
(*range_subpartition_desc*::=, *list_subpartition_desc*::=, *individual_hash_subparts*::=, *hash_subparts_by_quantity*::=)

***subpartition_by_range*::=**



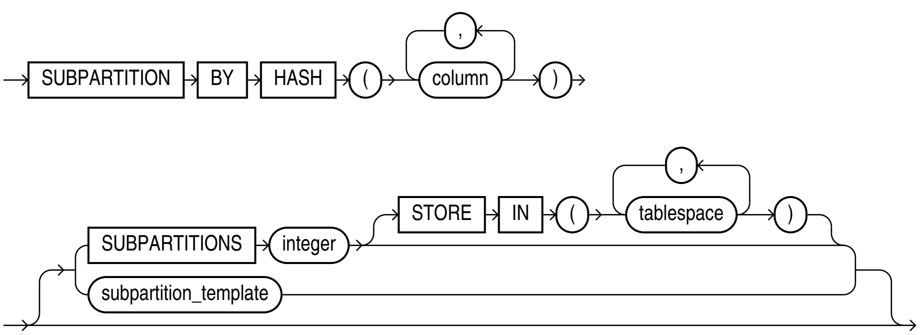
(*subpartition_template*::=)

***subpartition_by_list*::=**



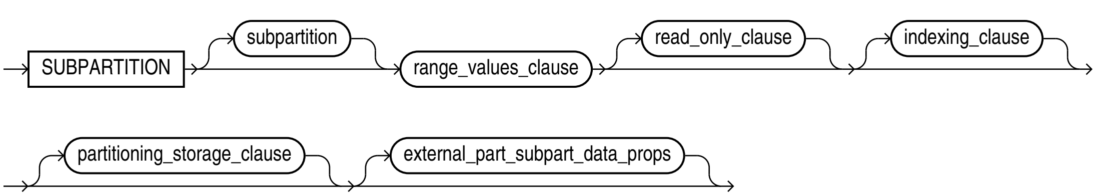
(*subpartition_template*::=)

***subpartition_by_hash*::=**



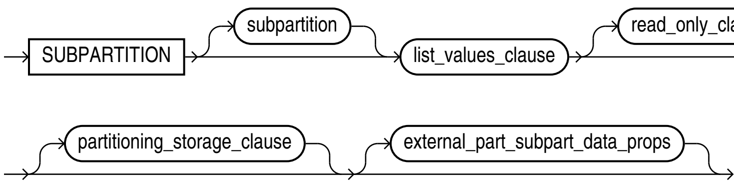
(*subpartition_template*::=)

***range_subpartition_desc*::=**



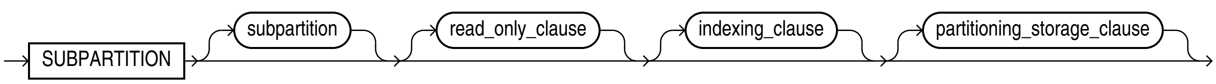
(*range_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***list_subpartition_desc*::=**



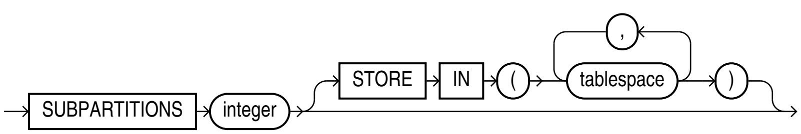
(*list_values_clause*::=, *read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=, *external_part_subpart_data_props*::=)

***individual_hash_subparts*::=**

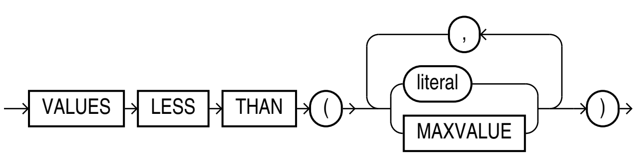


(*read_only_clause*::=, *indexing_clause*::=, *partitioning_storage_clause*::=)

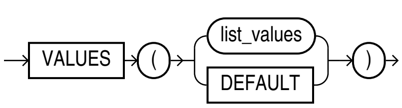
***hash_subparts_by_quantity*::=**



***range_values_clause*::=**

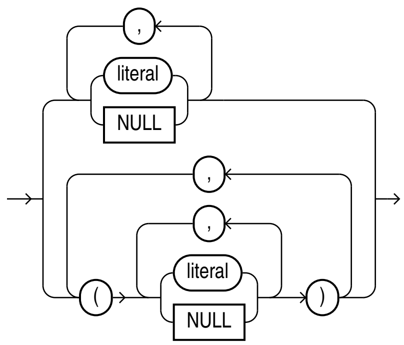


***list_values_clause*::=**

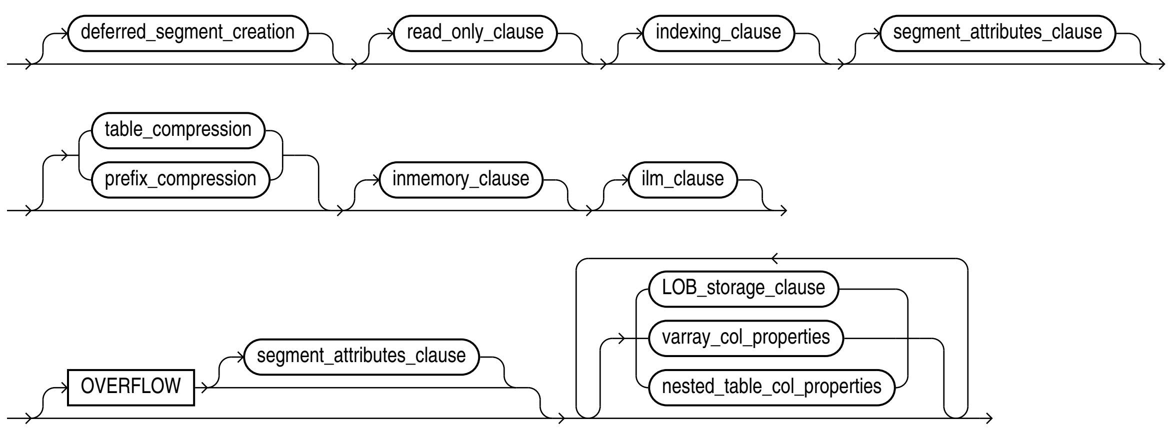


(*list_values*::=)

***list_values*::=**

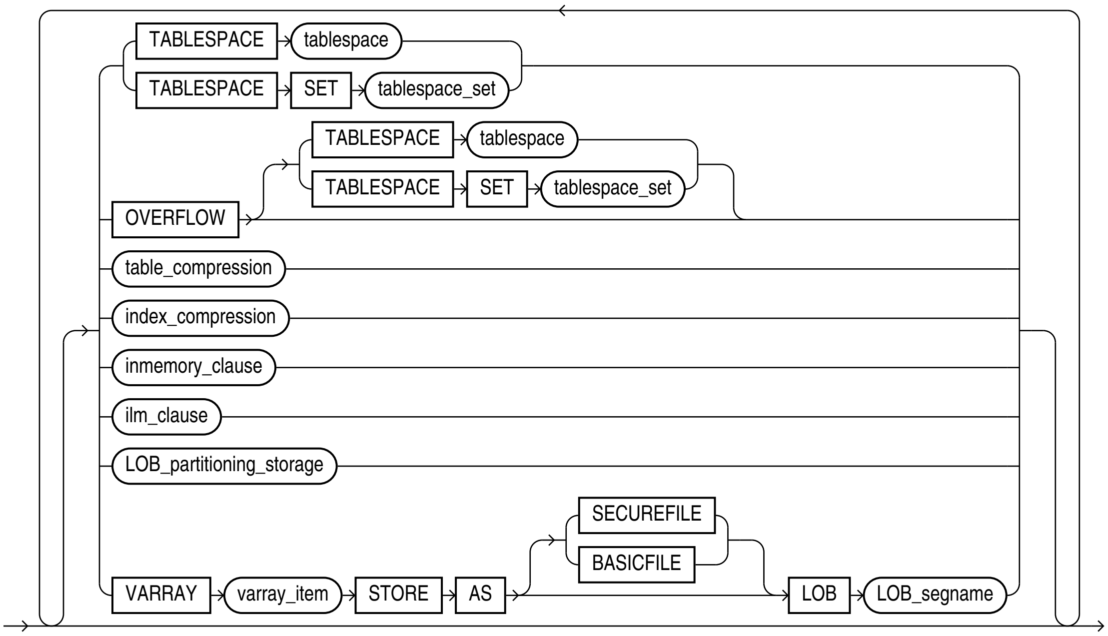


***table_partition_description*::=**



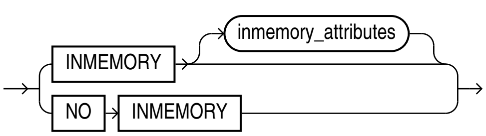
(*deferred_segment_creation*::=, *read_only_clause*::=, *indexing_clause*::=, *segment_attributes_clause*::=, *table_compression*::=, *prefix_compression*::=, *inmemory_clause*::=, *segment_attributes_clause*::=, *LOB_storage_clause*::=, *varray_col_properties*::=, *nested_table_col_properties*::=)

***partitioning_storage_clause*::=**



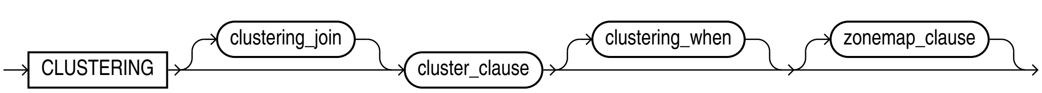
(*table_compression*::=, *index_compression*::=, *inmemory_clause*::=, *LOB_partitioning_storage*::=)

***inmemory_clause*::=**



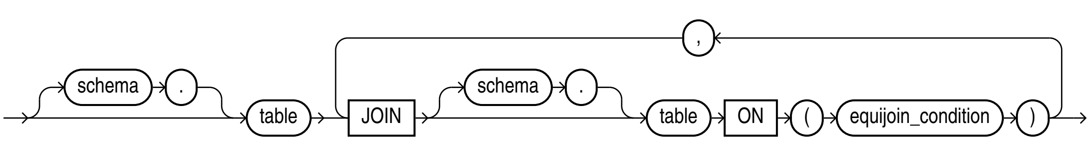
(*inmemory_memcompress*::=, *inmemory_attributes*::=)

***attribute_clustering_clause*::=**

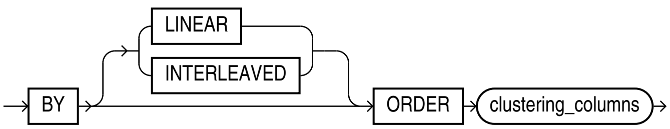


(*clustering_join*::=, *cluster_clause*::=, *clustering_when*::=, *zonemap_clause*::=)

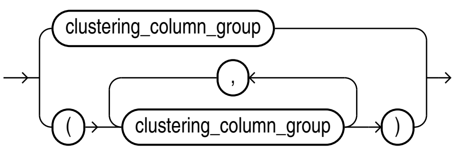
***clustering_join*::=**



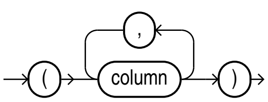
***cluster_clause*::=**



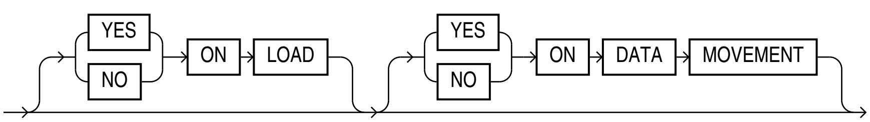
***clustering_columns*::=**



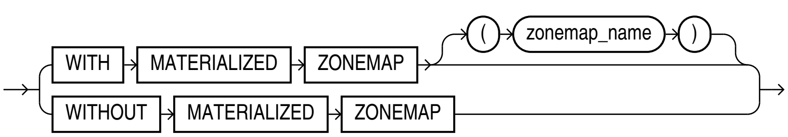
***clustering_column_group*::=**



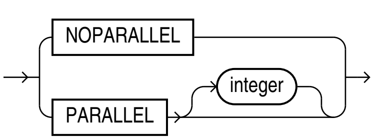
***clustering_when*::=**



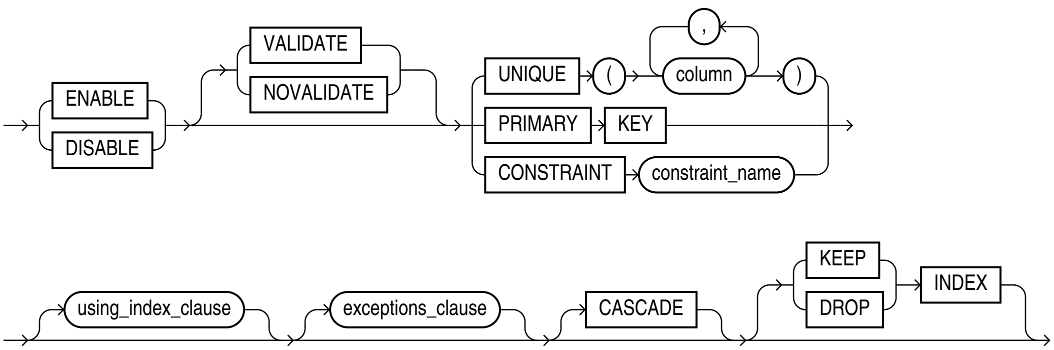
***zonemap_clause*::=**



***parallel_clause*::=**

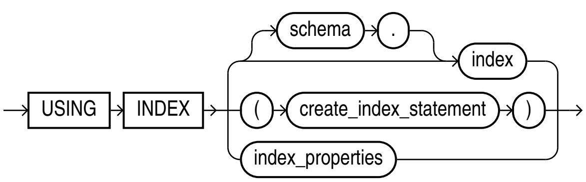


***enable_disable_clause*::=**



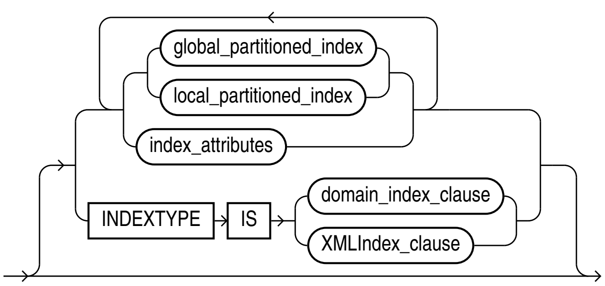
(*using_index_clause*::=, `exceptions_clause` not supported in `CREATE TABLE` statements)

***using_index_clause*::=**



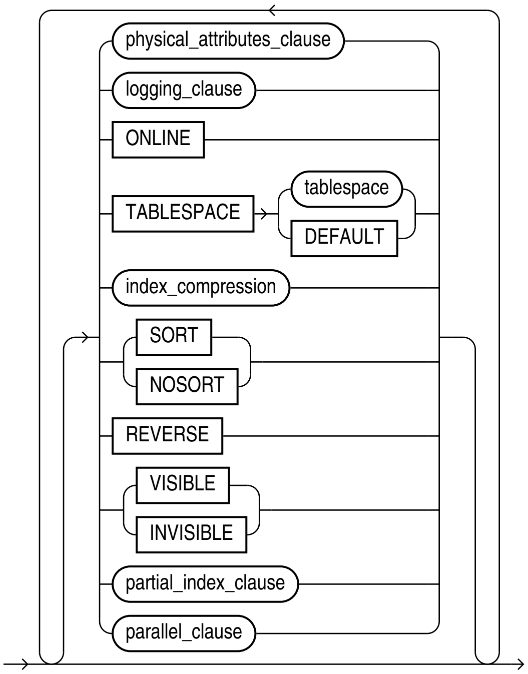
(*create_index*::=, *index_properties*::=)

***index_properties*::=**



(*global_partitioned_index*::=, *local_partitioned_index*::=—part of `CREATE INDEX`, *index_attributes*::=, `domain_index_clause` and `XMLIndex_clause`: not supported in `using_index_clause`)

***index_attributes*::=**



(*physical_attributes_clause*::=, *logging_clause*::=, *index_compression*::=, `partial_index_clause` and `parallel_clause`: not supported in `using_index_clause`)

Semantics

GLOBAL TEMPORARY

Specify `GLOBAL TEMPORARY` to indicate that the table is temporary and that its **definition** is visible to all sessions with appropriate privileges. The **data** in a temporary table is visible only to the session that inserts the data into the table.

When you first create a temporary table, its table metadata is stored in the data dictionary, but no space is allocated for table data. Space is allocated for the table segment at the time of the first DML operation on the table. The temporary table definition persists in the same way as the definitions of regular tables, but the table segment and any data the table contains are either **session-specific** or **transaction-specific** data. You specify whether the table segment and data are session- or transaction-specific with the ON COMMIT keywords.

You can perform DDL operations (such as `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`) on a temporary table only when no session is bound to it. A session becomes bound to a temporary table by performing an `INSERT` operation on the table. A session becomes unbound to the temporary table by issuing a `TRUNCATE` statement or at session termination, or, for a transaction-specific temporary table, by issuing a `COMMIT` or `ROLLBACK` statement.

> **See Also:**
>
> *Oracle Database Concepts* for information on temporary tables and "Creating a Table: Temporary Table Example"

Restrictions on Temporary Tables

Temporary tables are subject to the following restrictions:

- Temporary tables cannot be partitioned, clustered, or index organized.
- You cannot specify any foreign key constraints on temporary tables.
- Temporary tables cannot contain columns of nested table.
- You cannot specify the following clauses of the *LOB_storage_clause*: TABLESPACE, *storage_clause*, or *logging_clause*.
- Parallel UPDATE, DELETE and MERGE are not supported for temporary tables.
- The only part of the *segment_attributes_clause* you can specify for a temporary table is TABLESPACE, which allows you to specify a single temporary tablespace.
- Distributed transactions are not supported for temporary tables.
- A temporary table cannot contain INVISIBLE columns.

SHARDED

Specify SHARDED to create a sharded table.

This clause is valid only if you are using Oracle Sharding, which is a data tier architecture in which data is horizontally partitioned across independent databases. Each database in such configuration is called a shard. All of the shards together make up a single logical database, which is referred to as a sharded database (SDB). Horizontal partitioning involves splitting a table across shards so that each shard contains the table with the same columns but a different subset of rows. A table split up in this manner is called a sharded table.

When you create a sharded table, you must specify a tablespace set in which to create the table. There is no default tablespace set for sharded tables. See CREATE TABLESPACE SET for more information.

Oracle Sharding is based on the Oracle Partitioning feature. Therefore, a sharded table must be a partitioned or composite-partitioned table. When creating a sharded table, you must specify one of the *table_partitioning_clauses*. See *table_partitioning_clauses* for the full semantics of these clauses.

Restrictions on Sharded Tables

The following restrictions apply to sharded tables:

- A sharded table cannot be a temporary table or an index-organized table.
- A sharded table cannot contain a nested table column or an identity column.
- You cannot specify a tablespace for a sharded table with the TABLESPACE clause. You must specify a tablespace set with the TABLESPACE SET clause.

- A primary key constraint defined on a sharded table must contain the sharding column(s). A foreign key constraint on a column of a sharded table referencing a duplicated table column is not supported.
- System partitioning and interval-range partitioning are not supported for sharded tables.
- You cannot specify a virtual column in a sharded table in the `PARTITION BY` or `PARTITIONSET BY` clauses.

DUPLICATED

This clause is valid only if you are using Oracle Sharding. Specify `DUPLICATED` to create a duplicated table, which is duplicated on all shards. It can be a nonpartitioned table or partitioned table.

Restrictions on Duplicated Tables

The following restrictions apply to duplicated tables:

- A duplicated table cannot contain a `LONG` column.
- The maximum number of non-primary key columns in a duplicated table is 999.
- An `XMLType` column in a duplicated table can be used only in an Automatic Segment Space Management (ASSM) tablespace.
- A duplicated table cannot be a temporary table.
- A duplicated table cannot be a reference-partitioned table or a system-partitioned table.
- You cannot specify `NOLOGGING` or `PARALLEL` for a duplicated table.
- You cannot enable a duplicated table for the In-Memory Column Store.

schema

Specify the schema to contain the table. If you omit *`schema`*, then the database creates the table in your own schema.

table

Specify the name of the table or object table to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

> **See Also:**
>
> "Creating Tables: General Examples"

SHARING

This clause applies only when creating a table in an application root. This type of table is called an application common object and its data can be shared with the application PDBs that belong to the application root. To determine how the table data is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the table’s metadata, but its data is unique to each container. This type of table is referred to as a **metadata-linked application common object**.

- DATA - A data link shares the table, and its data is the same for all containers in the application container. Its data is stored only in the application root. This type of table is referred to as a **data-linked application common object**.
- EXTENDED DATA - An extended data link shares the table, and its data in the application root is the same for all containers in the application container. However, each application PDB in the application container can store data that is unique to the application PDB. For this type of table, data is stored in the application root and, optionally, in each application PDB. This type of table is referred to as an **extended data-linked application common object**.
- NONE - The table is not shared.

If you omit this clause, then the database uses the value of the DEFAULT_SHARING initialization parameter to determine the sharing attribute of the table. If the DEFAULT_SHARING initialization parameter does not have a value, then the default is METADATA.

When creating a relational table, you can specify METADATA, DATA, EXTENDED DATA, or NONE.

When creating an object table or an XMLTYPE table, you can specify only METADATA or NONE.

You cannot change the sharing attribute of a table after it is created.

See Also:

- *Oracle Database Reference* for more information on the DEFAULT_SHARING initialization parameter
- *Oracle Database Administrator's Guide* for complete information on creating application common objects

relational_table

This clause lets you create a relational table.

relational_properties

The relational properties describe the components of a relational table.

column_definition

The *column_definition* lets you define the characteristics of the column.

Specifying *column_definition* with AS *subquery*

If you specify the AS *subquery* clause, and each column returned by *subquery* has a column name or is an expression with a specified column alias, then you can omit the *column_definition* clause. In this case, the names of the columns of table are the same as the names of the columns returned by *subquery*. The exception is creating an index-organized table, for which you must specify the *column_definition* clause, because you must designate a primary key column. Regardless of the table type, if you specify the *column_definition* clause and the AS *subquery* clause, then you must omit *datatype* from the *column_definition* clause.

column

Specify the name of a column of the table. The name must satisfy the requirements listed in "Database Object Naming Rules ".

If you also specify `AS` *`subquery`*, then you can omit *`column`* and *`datatype`* unless you are creating an index-organized table. If you specify `AS` *`subquery`* when creating an index-organized table, then you must specify *`column`*, and you must omit *`datatype`*.

The absolute maximum number of columns in a table is 1000. When you create an object table or a relational table with columns of object, nested table, varray, or REF type, Oracle Database maps the columns of the user-defined types to relational columns, in effect creating hidden columns that count toward the 1000-column limit. A relational column that stores a user-defined type attribute inherits the collation property of the attribute. In Oracle Database 12*c* Release 2 (12.2), user-defined types are created using the pseudo-collation property `USING_NLS_COMP` and their corresponding relational columns inherit this property.

datatype

Specify the data type of a column.

In general, you must specify *`datatype`*. However, the following exceptions apply:

- You must omit *`datatype`* if you specify the `AS` *`subquery`* clause.
- You can also omit *`datatype`* if the statement designates the column as part of a foreign key in a referential integrity constraint. Oracle Database automatically assigns to the column the data type of the corresponding column of the referenced key of the referential integrity constraint.

Restrictions on Table Column Data Types

The following restrictions apply to the data types of table columns:

- Do not create a table with `LONG` columns. Use LOB columns (`CLOB`, `NCLOB`, `BLOB`) instead. `LONG` columns are supported only for backward compatibility.
- You can specify a column of type `ROWID`, but Oracle Database does not guarantee that the values in such columns are valid rowids.

> **See Also:**
>
> "Data Types " for information on `LONG` columns and on Oracle-supplied data types

COLLATE

The `COLLATE` clause lets you specify a data-bound collation for the column.

For *`column_collation_name`*, specify a valid named collation or pseudo-collation. For columns of data type `CLOB` or `NCLOB`, the only allowed value for *`column_collation_name`* is the pseudo-collation `USING_NLS_COMP`.

If you omit this clause, then the column is assigned:

- the pseudo-collation `USING_NLS_COMP`, if the column has the data type `CLOB` or `NCLOB`, or

- the collation of the corresponding parent key column, if the column belongs to a foreign key, or
- the default collation for the table as it stands at the time the column is created.

Refer to the DEFAULT COLLATION clause for more information on the default collation for a table.

You can specify the `COLLATE` clause only if the `COMPATIBLE` initialization parameter is set to `12.2` or greater, and the `MAX_STRING_SIZE` initialization parameter is set to `EXTENDED`.

SORT

The `SORT` keyword is valid only if you are creating this table as part of a hash cluster and only for columns that are also cluster columns.

Table rows are hashed into buckets on cluster key columns without `SORT`, and then sorted in each bucket on the columns with this clause. This may improve response time during subsequent operations on the clustered data.

> **See Also:**
>
> - "CLUSTER Clause" for information on creating a cluster table
> - *Managing Hash Clusters*

VISIBLE | INVISIBLE

Use this clause to specify whether *`column`* is `VISIBLE` or `INVISIBLE`. The default is `VISIBLE`.

`INVISIBLE` columns are user-specified hidden columns. To display or assign a value to an `INVISIBLE` column, you must specify its name explicitly. For example:

- The `SELECT *` syntax will not display an `INVISIBLE` column. However, if you include an `INVISIBLE` column in the select list of a `SELECT` statement, then the column will be displayed.
- You cannot implicitly specify a value for an `INVISIBLE` column in the `VALUES` clause of an `INSERT` statement. You must specify the `INVISIBLE` column in the column list.
- You must explicitly specify an `INVISIBLE` column in Oracle Call Interface (OCI) describes.
- You can configure SQL*Plus to allow `INVISIBLE` column information to be viewed with the `DESCRIBE` command. Refer to *SQL*Plus User's Guide and Reference* for more information.

Notes on VISIBLE and INVISIBLE Columns

The following notes apply to `VISIBLE` and `INVISIBLE` columns:

- An `INVISIBLE` column can be used as a partitioning key when specified as part of `CREATE TABLE`.
- You can specify `INVISIBLE` columns in a *`column_expression`*.

- A virtual column can be an INVISIBLE column.
- PL/SQL %ROWTYPE attributes do not show INVISIBLE columns.
- The COLUMN_ID column of the ALL_, DBA_, and USER_TAB_COLUMNS data dictionary views determines the order in which a SELECT * query returns columns for a table, view, or materialized view. The value of COLUMN_ID is NULL for INVISIBLE columns. When you make an invisible column visible, it will be assigned the next highest available COLUMN_ID value. When you make a visible column invisible, its COLUMN_ID value is set to NULL and COLUMN_ID is decremented by 1 for any columns with a higher COLUMN_ID.

Restrictions on VISIBLE and INVISIBLE Columns

The following restrictions apply to VISIBLE and INVISIBLE columns:

- INVISIBLE columns are not supported in external tables, cluster tables, or temporary tables.
- You cannot make a system-generated hidden column visible.

> **Note:**
>
> To determine whether a column is a system-generated hidden column, query the HIDDEN_COLUMN and USER_GENERATED columns of the ALL_, DBA_, and USER_TAB_COLS data dictionary views. Refer to *Oracle Database Reference* for more information.

DEFAULT

The DEFAULT clause lets you specify a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The data type of the expression must match the data type specified for the column. The column must also be large enough to hold this expression.

The DEFAULT expression can include any SQL function as long as the function does not return a literal argument, a column reference, or a nested function invocation.

The DEFAULT expression can include the sequence pseudocolumns CURRVAL and NEXTVAL, as long as the sequence exists and you have the privileges necessary to access it. Users who perform subsequent inserts that use the DEFAULT expression must have the INSERT privilege on the table and the SELECT privilege on the sequence. If the sequence is later dropped, then subsequent INSERT statements where the DEFAULT expression is used will result in an error. If you do not fully qualify the sequence by specifying the sequence owner, for example, SCOTT.SEQ1, then Oracle Database will default the sequence owner to be the user who issues the CREATE TABLE statement. For example, if user MARY creates SCOTT.TABLE and refers to a sequence that is not fully qualified, such as SEQ2, then the column will use sequence MARY.SEQ2. Synonyms on sequences undergo a full name resolution and are stored as the fully qualified sequence in the data dictionary; this is true for public and private synonyms. For example, if user BETH adds a column referring to public or private synonym SYN1 and the synonym refers to PETER.SEQ7, then the column will store PETER.SEQ7 as the default.

Restrictions on Default Column Values

Default column values are subject to the following restrictions:

- A DEFAULT expression cannot contain references to PL/SQL functions or to other columns, the pseudocolumns LEVEL, PRIOR, and ROWNUM, or date constants that are not fully specified.
- The expression can be of any form except a scalar subquery expression.

> **See Also:**
>
> "About SQL Expressions " for the syntax of *expr*

ON NULL

If you specify the ON NULL clause, then Oracle Database assigns the DEFAULT column value when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.

When you specify ON NULL, the NOT NULL constraint and NOT DEFERRABLE constraint state are implicitly specified. If you specify an inline constraint that conflicts with NOT NULL and NOT DEFERRABLE, then an error is raised.

Restriction on the ON NULL Clause

You cannot specify this clause for an object type column or a REF column.

> **See Also:**
>
> "Creating a Table with a DEFAULT ON NULL Column Value: Example"

identity_clause

Use this clause to specify an identity column. The identity column will be assigned an increasing or decreasing integer value from a sequence generator for each subsequent INSERT statement. You can use the *identity_options* clause to configure the sequence generator.

ALWAYS

If you specify ALWAYS, then Oracle Database always uses the sequence generator to assign a value to the column. If you attempt to explicitly assign a value to the column using INSERT or UPDATE, then an error will be returned. This is the default.

BY DEFAULT

If you specify BY DEFAULT, then Oracle Database uses the sequence generator to assign a value to the column by default, but you can also explicitly assign a specified value to the column. If you specify ON NULL, then Oracle Database uses the sequence generator to assign a value to the column when a subsequent INSERT statement attempts to assign a value that evaluates to NULL.

identity_options

Use the *identity_options* clause to configure the sequence generator. The *identity_options* clause has the same parameters as the CREATE SEQUENCE statement. Refer to CREATE SEQUENCE for a full description of these parameters and characteristics. The exception is START WITH LIMIT VALUE, which is specific to *identity_options* and can only be used with ALTER TABLE MODIFY. Refer to identity_options for more information.

> **Note:**
>
> When you create an identity column, Oracle recommends that you specify the CACHE clause with a value higher than the default of 20 to enhance performance.

Restrictions on Identity Columns

Identity columns are subject to the following restrictions:

- You can specify only one identity column per table.
- If you specify *identity_clause*, then you must specify a numeric data type for *datatype* in the *column_definition* clause. You cannot specify a user-defined data type.
- If you specify *identity_clause*, then you cannot specify the DEFAULT clause in the *column_definition* clause.
- When you specify *identity_clause*, the NOT NULL constraint and NOT DEFERRABLE constraint state are implicitly specified. If you specify an inline constraint that conflicts with NOT NULL and NOT DEFERRABLE, then an error is raised.
- If an identity column is encrypted, then the encryption algorithm may be inferred. Oracle recommends that you use a strong encryption algorithm on identity columns.
- CREATE TABLE AS SELECT will not inherit the identity property on a column.

> **See Also:**
>
> "Creating a Table with an Identity Column: Examples"

encryption_spec

The ENCRYPT clause lets you use the Transparent Data Encryption (TDE) feature to encrypt the column you are defining. You can encrypt columns of type CHAR, NCHAR, VARCHAR2, NVARCHAR2, NUMBER, DATE, LOB, and RAW. The data does not appear in its encrypted form to authorized users, such as the user who encrypts the column.

> **Note:**
>
> Column encryption requires that a system administrator with appropriate privileges has initialized the security module, opened a keystore, and set an encryption key. Refer to *Oracle Database Advanced Security Guide* for general information about column encryption and to *security_clauses* for related ALTER SYSTEM statements.

USING '*encrypt_algorithm*'

Use this clause to specify the name of the algorithm to be used. Valid algorithms are AES256, AES192, AES128 and 3DES168. If the COMPATIBLE initialization parameter is set to 12.2 or higher, then the following algorithms are also valid: ARIA128, ARIA192, ARIA256, GOST256, and SEED128. If you omit this clause, then the database uses AES192. If you encrypt more than one column in the same table, and if you specify the USING clause for one of the columns, then you must specify the same encryption algorithm for all the encrypted columns.

IDENTIFIED BY *password*

If you specify this clause, then the database derives the column key from the specified password.

'*integrity_algorithm*'

Use this clause to specify the integrity algorithm to be used. Valid integrity algorithms are SHA-1 and NOMAC.

- If you specify SHA-1, then TDE uses the Secure Hash Algorithm (SHA-1) and adds a 20-byte Message Authentication Code (MAC) to each encrypted value for integrity checking. This is the default.
- If you specify NOMAC, then TDE does not add a MAC and does not perform the integrity check. This saves 20 bytes of disk space per encrypted value. Refer to *Oracle Database Advanced Security Guide* for more information on using NOMAC to save disk space and improve performance.

All encrypted columns in a table must use the same integrity algorithm. If you already have a table column using the SHA-1 algorithm, then you cannot use the NOMAC parameter to encrypt another column in the same table. Refer to the REKEY *encryption_spec* clause of ALTER TABLE to learn how to change the integrity algorithm used by all encrypted columns in a table.

SALT | NO SALT

Specify SALT to instruct the database to append a random string, called "salt," to the clear text of the column before encrypting it. This is the default.

Specify NO SALT to prevent the database from appending salt to the clear text of the column before encrypting it.

The following considerations apply when specifying SALT or NO SALT for encrypted columns:

- If you want to use the column as an index key, then you must specify NO SALT. Refer to *Oracle Database Advanced Security Guide* for a description of "salt" in this context.
- If you specify table compression for the table, then the database does not compress the data in encrypted columns with SALT.

You cannot specify SALT or NO SALT for LOB encryption.

Restrictions on *encryption_spec*

The following restrictions apply to column encryption:

- Transparent Data Encryption is not supported by the traditional import and export utilities or by transportable-tablespace-based export. Use the Data Pump import and export utilities with encrypted columns instead.

- To encrypt a column in an external table, the table must use ORACLE_DATAPUMP as its access type.
- You cannot encrypt a column in tables owned by SYS.
- You cannot encrypt a foreign key column.

> **See Also:**
>
> *Oracle Database Advanced Security Guide* for more information about Transparent Data Encryption

virtual_column_definition

The *virtual_column_definition* clause lets you create a virtual column. A virtual column is not stored on disk. Rather, the database derives the values in a virtual column on demand by computing a set of expressions or functions. Virtual columns can be used in queries, DML, and DDL statements. They can be indexed, and you can collect statistics on them. Thus, they can be treated much as other columns. Exceptions and restrictions are listed below in "Notes on Virtual Columns" and "Restrictions on Virtual Columns".

column

For *column*, specify the name of the virtual column.

datatype

You can optionally specify the data type of the virtual column. If you omit *datatype*, then the database determines the data type of the column based on the data type of the underlying expressions. All Oracle scalar data types and XMLType are supported.

COLLATE

The COLLATE clause lets you specify a data-bound collation for the virtual column. For *column_collation_name*, specify a valid named collation or pseudo-collation. If you omit this clause, then the column is assigned the default collation for the table as it stands at the time the column is created, unless the column belongs to a foreign key, in which case it inherits the collation from the corresponding column of the parent key. Refer to the DEFAULT COLLATION clause for more information on the default collation for a table.

You can specify the COLLATE clause only if the COMPATIBLE initialization parameter is set to 12.2 or greater, and the MAX_STRING_SIZE initialization parameter is set to EXTENDED.

VISIBLE | INVISIBLE

Use this clause to specify whether virtual column is VISIBLE or INVISIBLE. The default is VISIBLE. For complete information, refer to "VISIBLE | INVISIBLE".

GENERATED ALWAYS

The optional keywords GENERATED ALWAYS are provided for semantic clarity. They indicate that the column is not stored on disk, but is evaluated on demand.

column_expression

The AS *column_expression* clause determines the content of the column. Refer to "Column Expressions " for more information on *column_expression*.

VIRTUAL

The optional keyword `VIRTUAL` is provided for semantic clarity.

evaluation_edition_clause

You must specify this clause if *`column_expression`* refers to an editioned PL/SQL function. Use this clause to specify the edition that is searched during name resolution of the editioned PL/SQL function—the evaluation edition.

- Specify `CURRENT EDITION` to search the edition in which this DDL statement is executed.
- Specify `EDITION` *`edition`* to search *`edition`*.
- Specifying `NULL EDITION` is equivalent to omitting the *`evaluation_edition_clause`*.

If you omit the *`evaluation_edition_clause`*, then editioned objects are invisible during name resolution and an error will result. If the evaluation edition is dropped, then a subsequent query on the virtual column will result in an error.

The database does not maintain dependencies on the functions referenced by a virtual column. Therefore, if a virtual column refers to a noneditioned function, and the function becomes editioned, then the following operations may raise an error:

- Querying the virtual column
- Updating a row that includes the virtual column
- Firing a trigger that accesses the virtual column

> **See Also:**
>
> *Oracle Database Development Guide* for more information on specifying the evaluation edition for a virtual column

unusable_editions_clause

This clause lets you specify that the virtual column expression is unusable for evaluating queries in one or more editions. The remaining editions form a range of editions in which it is safe for the optimizer to use the virtual column expression to evaluate queries.

For example, suppose you define a function-based index on the virtual column. The optimizer can use the function-based index to evaluate queries that contain the virtual column expression in their `WHERE` clause. If a query is compiled in an edition that is in the usable range of editions for the virtual column, then the optimizer will consider using the index to evaluate the query. If a query is compiled in an edition outside the usable range of editions for the virtual column, then the optimizer will not consider using the index.

> **See Also:**
>
> *Oracle Database Concepts* for more information on optimization with function-based indexes

UNUSABLE BEFORE Clause

This clause lets you specify that the virtual column expression is unusable for evaluating queries in the ancestors of an edition.

- If you specify `CURRENT EDITION`, then the virtual column expression is unusable in the ancestors of the edition in which this DDL statement is executed.
- If you specify `EDITION` *`edition`*, then the virtual column expression is unusable in the ancestors of the specified *`edition`*.

UNUSABLE BEGINNING WITH Clause

This clause lets you specify that the virtual column expression is unusable for evaluating queries in an edition and its descendants.

- If you specify `CURRENT EDITION`, then the virtual column expression is unusable in the edition in which this DDL statement is executed and its descendants.
- If you specify `EDITION` *`edition`*, then the virtual column expression is unusable in the specified *`edition`* and its descendants.
- Specifying `NULL EDITION` is equivalent to omitting the `UNUSABLE BEGINNING WITH` clause.

If an edition specified in this clause is subsequently dropped, there is no effect on the virtual column.

Notes on Virtual Columns

- If *`column_expression`* refers to a column on which column-level security is implemented, then the virtual column does not inherit the security rules of the base column. In such a case, you must ensure that data in the virtual column is protected, either by duplicating a column-level security policy on the virtual column or by applying a function that implicitly masks the data. For example, it is common for credit card numbers to be protected by a column-level security policy, while still allowing call center employees to view the last four digits of the credit card number for validation purposes. In such a case, you could define the virtual column to take a substring of the last four digits of the credit card number.
- A table index defined on a virtual column is equivalent to a function-based index on the table.
- You cannot directly update a virtual column. Thus, you cannot specify a virtual column in the `SET` clause of an `UPDATE` statement. However, you can specify a virtual column in the `WHERE` clause of an `UPDATE` statement. Likewise, you can specify a virtual column in the `WHERE` clause of a `DELETE` statement to delete rows from a table based on the derived value of the virtual column.
- A query that specifies in its `FROM` clause a table containing a virtual column is eligible for result caching. Refer to "RESULT_CACHE Hint " for more information on result caching.
- The *`column_expression`* can refer to a PL/SQL function if the function is explicitly designated `DETERMINISTIC` during its creation. However, if the function is subsequently replaced, definitions dependent on the virtual column are not invalidated. In such a case,

if the table contains data, queries that reference the virtual column may return incorrect results if the virtual column is used in the definition of constraints, indexes, or materialized views or for result caching. Therefore, in order to replace the deterministic PL/SQL function for a virtual column.

  – Disable and re-enable any constraints on the virtual column.
  – Rebuild any indexes on the virtual column.
  – Fully refresh materialized views accessing the virtual column.
  – Flush the result cache if cached queries have accessed the virtual column.
  – Regather statistics on the table.

- A virtual column can be an `INVISIBLE` column. The *`column_expression`* can contain `INVISIBLE` columns.

Restrictions on Virtual Columns

- You can create virtual columns only in relational heap tables. Virtual columns are not supported for index-organized, external, object, cluster, or temporary tables.
- The *`column_expression`* in the `AS` clause has the following restrictions:
  – It cannot refer to another virtual column by name.
  – Any columns referenced in *`column_expression`* must be defined on the same table.
  – It can refer to a deterministic user-defined function, but if it does, then you cannot use the virtual column as a partitioning key column.
  – The output of *`column_expression`* must be a scalar value.

> **See Also:**
>
> "Column Expressions " for additional information and restrictions on *`column_expression`*

- The virtual column cannot be an Oracle supplied data type, a user-defined type, or LOB or `LONG RAW`.
- You cannot specify a call to a PL/SQL function in the defining expression for a virtual column that you want to use as a partitioning column.

> **See Also:**
>
> "Adding a Virtual Table Column: Example" and *Oracle Database Administrator's Guide* for examples of creating tables with virtual columns

period_definition

Use the *`period_definition`* clause to create a valid time dimension for *`table`*.

This clause implements Temporal Validity support for *`table`*. If you specify this clause, then one column in *`table`*, the start time column, contains a start date or timestamp, and another column in *table*, the end time column, contains an end date or timestamp. These two columns define a valid time dimension for *table*—that is, a period of time for which each row is considered valid. You can use Oracle Flashback Query to retrieve rows from *table* based on whether they are considered valid as of a specified time, before a specified time, or during a specified time period.

You can specify at most one valid time dimension when you create a table. You can subsequently add additional valid time dimensions to a table with the *add_period_clause* of ALTER TABLE.

valid_time_column

Specify the name of the valid time dimension. The name must satisfy the requirements listed in "Database Object Naming Rules ". Oracle Database creates an INVISIBLE virtual column with this name of data type NUMBER in *table*.

start_time_column* and *end_time_column

You can optionally specify these clauses as follows:

- Use *start_time_column* to specify the name of the start time column, which contains the start date or timestamp.
- Use *end_time_column* to specify the name of the end time column, which contains the end date or timestamp.

The names you specify for *start_time_column* and *end_time_column* must satisfy the requirements listed in "Database Object Naming Rules ".

If you specify these clauses, then you must define *start_time_column* and *end_time_column* in the *column_definition* clause of CREATE TABLE. Each column must be of a datetime data type (DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or TIMESTAMP WITH LOCAL TIME ZONE) and can be VISIBLE or INVISIBLE.

If you do not specify these clauses, then Oracle Database creates a start time column named *valid_time_column*_START, and an end time column named *valid_time_column*_END. These columns are of data type TIMESTAMP WITH TIME ZONE and are INVISIBLE.

You can insert and update values in the start time column and end time column as you would any column, with the following considerations:

- If the value of the start time column is NULL, then the row is considered valid for all time values that occur before, but not including, the value of the end time column.
- If the value of the end time column is NULL, then the row is considered valid for all time values that occur on or after the value of the start time column.
- If the value of neither column is NULL, then the value of the start time column must be earlier than the value of the end time column. The row is considered valid for all time values that occur on or after the value of the start time column, and up to, but not including, the value of the end time column.
- If the value of both columns is NULL, then the row is considered valid for all time values.

Restrictions on Valid Time Dimension Columns

The following restrictions apply to valid time dimension columns:

- The *valid_time_column* is for internal use only. You cannot perform DDL or DML operations on it with one exception: You can drop the column by using the *drop_period_clause* of ALTER TABLE.

- You can drop the start time column and end time column only by using the *drop_period_clause* of ALTER TABLE.
- If the start time column and end time column are automatically created by Oracle Database, then they are INVISIBLE and you cannot subsequently make them VISIBLE.

> **See Also:**
>
> - *Oracle Database Development Guide* for more information on Temporal Validity
> - SELECT *flashback_query_clause* for more information on Oracle Flashback Query
> - ALTER TABLE *add_period_clause* and *drop_period_clause* for information how to add and drop a valid time dimension

Constraint Clauses

Use these clauses to create constraints on the table columns. You must specify a PRIMARY KEY constraint for an index-organized table, and it cannot be DEFERRABLE. Refer to *constraint* for syntax and description of these constraints as well as examples.

inline_ref_constraint* and *out_of_line_ref_constraint

These clauses let you describe a column of type REF. The only difference between these clauses is that you specify *out_of_line_ref_constraint* from the table level, so you must identify the REF column or attribute you are defining. Specify *inline_ref_constraint* as part of the definition of the REF column or attribute.

> **See Also:**
>
> "REF Constraint Examples"

inline_constraint

Use the *inline_constraint* to define an integrity constraint as part of the column definition.

You can create UNIQUE, PRIMARY KEY, and REFERENCES constraints on scalar attributes of object type columns. You can also create NOT NULL constraints on object type columns and CHECK constraints that reference object type columns or any attribute of an object type column.

out_of_line_constraint

Use the *out_of_line_constraint* syntax to define an integrity constraint as part of the table definition.

supplemental_logging_props

The *supplemental_logging_props* clause lets you instruct the database to put additional data into the log stream to support log-based tools.

supplemental_log_grp_clause

Use this clause to create a named log group.

- The NO LOG clause lets you omit from the redo log one or more columns that would otherwise be included in the redo for the named log group. You must specify at least one fixed-length column without NO LOG in the named log group.
- If you specify ALWAYS, then during an update, the database includes in the redo all columns in the log group. This is called an **unconditional log group** (sometimes called an "always log group"), because Oracle Database supplementally logs all the columns in the log group when the associated row is modified. If you omit ALWAYS, then the database supplementally logs all the columns in the log group only if any column in the log group is modified. This is called a **conditional log group**.

You can query the appropriate USER_, ALL_, or DBA_LOG_GROUP_COLUMNS data dictionary view to determine whether any supplemental logging has already been specified.

supplemental_id_key_clause

Use this clause to specify that all or a combination of the primary key, unique key, and foreign key columns should be supplementally logged. Oracle Database will generate either an **unconditional log group** or a **conditional log group**. With an unconditional log group, the database supplementally logs all the columns in the log group when the associated row is modified. With a conditional log group, the database supplementally logs all the columns in the log group only if any column in the log group is modified.

- If you specify ALL COLUMNS, then the database includes in the redo log all the fixed-length maximum size columns of that row. Such a redo log is a system-generated unconditional log group.
- If you specify PRIMARY KEY COLUMNS, then for all tables with a primary key, the database places into the redo log all columns of the primary key whenever an update is performed. Oracle Database evaluates which columns to supplementally log as follows:
  - First the database chooses columns of the primary key constraint, if the constraint is validated or marked RELY and is not marked as DISABLED or INITIALLY DEFERRED.
  - If no primary key columns exist, then the database looks for the smallest UNIQUE index with at least one NOT NULL column and uses the columns in that index.
  - If no such index exists, then the database supplementally logs all scalar columns of the table.
- If you specify UNIQUE COLUMNS, then for all tables with a unique key or a bitmap index, if any of the unique key or bitmap index columns are modified, the database places into the redo log all other columns belonging to the unique key or bitmap index. Such a log group is a system-generated conditional log group.
- If you specify FOREIGN KEY COLUMNS, then for all tables with a foreign key, if any foreign key columns are modified, the database places into the redo log all other columns belonging to the foreign key. Such a redo log is a system-generated conditional log group.

If you specify this clause multiple times, then the database creates a separate log group for each specification. You can query the appropriate USER_, ALL_, or DBA_LOG_GROUPS data dictionary view to determine whether any supplemental logging data has already been specified.

DEFAULT COLLATION

This clause lets you specify the default collation for the table. The default collation is assigned to columns of the table that are of a character data type and are created with this statement or subsequently added to the table with an ALTER TABLE statement. For *collation_name*, specify a valid named collation or pseudo-collation.

If you omit this clause, then the default collation for the table is set to the effective schema default collation of the schema containing the table. Refer to the DEFAULT_COLLATION clause of ALTER SESSION for more information on the effective schema default collation.

You can override the table's default collation and assign a data-bound collation to a particular column by specifying the COLLATE clause in the *column_definition* or *virtual_column_definition* clause of CREATE TABLE or ALTER TABLE, or the *modify_col_properties* or *modify_virtcol_properties* clause of ALTER TABLE.

You can specify the DEFAULT COLLATION clause only if the COMPATIBLE initialization parameter is set to 12.2 or greater, and the MAX_STRING_SIZE initialization parameter is set to EXTENDED.

Restriction on Collation for CLOB and NCLOB Columns

If a column has the data type of CLOB or NCLOB, then its specified collation must be USING_NLS_COMP. The collation of CLOB and NCLOB columns is always USING_NLS_COMP and is not affected by the default collation for the table.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for full information on default collations and data-bound collations

ON COMMIT

The ON COMMIT clause is relevant only if you are creating a temporary table. This clause specifies whether the data in the temporary table persists for the duration of a transaction or a session.

DELETE ROWS

Specify DELETE ROWS for a transaction-specific temporary table. This is the default. Oracle Database will truncate the table (delete all its rows) after each commit.

PRESERVE ROWS

Specify PRESERVE ROWS for a session-specific temporary table. Oracle Database will truncate the table (delete all its rows) when you terminate the session.

physical_properties

The physical properties relate to the treatment of extents and segments and to the storage characteristics of the table.

deferred_segment_creation

Use this clause to determine when the database should create the segment(s) for this table:

- `SEGMENT CREATION DEFERRED`: This clause defers creation of the table segment — as well as segments for any LOB columns of the table, any indexes created implicitly as part of table creation, and any indexes subsequently explicitly created on the table — until the first row of data is inserted into the table. At that time, the segments for the table, LOB columns and indexes, and explicitly created indexes are all materialized and inherit any storage properties specified in this `CREATE TABLE` statement or, in the case of explicitly created indexes, the `CREATE INDEX` statement. These segments are created regardless whether the initial insert operation is uncommitted or rolled back. This is the default value.

  > **Caution:**
  >
  > When creating many tables with deferred segment creation, ensure that you allocate enough space for your database so that when the first rows are inserted, there is enough space for all the new segments.

- `SEGMENT CREATION IMMEDIATE`: The table segment is created as part of this `CREATE TABLE` statement.

Immediate segment creation is useful, for example, if your application depends upon the object appearing in the `DBA_`, `USER_`, and `ALL_SEGMENTS` data dictionary views, because the object will not appear in those views until the segment is created. This clause overrides the setting of the `DEFERRED_SEGMENT_CREATION` initialization parameter.

To determine whether a segment has been created for an existing table or its LOB columns or indexes, query the `SEGMENT_CREATED` column of `USER_TABLES`, `USER_INDEXES`, or `USER_LOBS`.

Notes on Tables Without Segments

The following rules apply to a table whose segment has not yet been materialized:

- If you create this table with `CREATE TABLE` ... `AS` *`subquery`*, then if the source table has no rows, segment creation of the new table is deferred. If the source table has rows, then segment creation of the new table is not deferred.
- If you specify `ALTER TABLE` ... `ALLOCATE EXTENT` before the segment is materialized, then the segment is materialized and then an extent is allocated. However the `ALLOCATE EXTENT` clause in a DDL statement on any indexes of the table will return an error.
- In a DDL statement on the table or its LOB columns or indexes, any specification of `DEALLOCATE UNUSED` is silently ignored.
- `ONLINE` operations on indexes of a table or table partition without a segment will silently be disabled; that is, they will proceed `OFFLINE`.
- If any of the following DDL statements are executed on a table with one or more LOB columns, then the resulting partition(s) or subpartition(s) will be materialized:
  - `ALTER TABLE SPLIT [SUB]PARTITION`

- `ALTER TABLE MERGE [SUB]PARTITIONS`
- `ALTER TABLE ADD [SUB]PARTITION` (hash partitions only)
- `ALTER TABLE COALESCE [SUB]PARTITION` (hash partitions only)

Restrictions on Deferred Segment Creation

This clause is subject to the following restrictions:

- You cannot defer segment creation for the following types of tables: clustered tables, global temporary tables, session-specific temporary tables, internal tables, external tables, and tables owned by `SYS`, `SYSTEM`, `PUBLIC`, `OUTLN`, or `XDB`.
- Deferred segment creation is not supported in dictionary-managed tablespaces.
- Deferred segment creation is not supported in the `SYSTEM` tablespace.
- Serializable transactions do not work with deferred segment creation. Trying to insert data into an empty table with no segment created causes an error.

> **See Also:**
>
> *Oracle Database Concepts* for general information on segment allocation and *Oracle Database Reference* for more information about the `DEFERRED_SEGMENT_CREATION` initialization parameter

segment_attributes_clause

The *`segment_attributes_clause`* lets you specify physical attributes and tablespace storage for the table.

physical_attributes_clause

The *`physical_attributes_clause`* lets you specify the value of the `PCTFREE`, `PCTUSED`, and `INITRANS` parameters and the storage characteristics of the table.

- For a nonpartitioned table, each parameter and storage characteristic you specify determines the actual physical attribute of the segment associated with the table.
- For partitioned tables, the value you specify for the parameter or storage characteristic is the default physical attribute of the segments associated with all partitions specified in this `CREATE` statement (and in subsequent `ALTER TABLE` ... `ADD PARTITION` statements), unless you explicitly override that value in the `PARTITION` clause of the statement that creates the partition.

If you omit this clause, then Oracle Database sets `PCTFREE` to 10, `PCTUSED` to 40, and `INITRANS` to 1.

> **See Also:**
>
> - *physical_attributes_clause* and storage_clause for a description of these clauses
> - "Creating a Table: Storage Example"

TABLESPACE

Specify the tablespace in which Oracle Database creates the table, object table `OIDINDEX`, partition, LOB data segment, LOB index segment, or index-organized table overflow data segment. If you omit `TABLESPACE`, then the database creates that item in the default tablespace of the owner of the schema containing the table.

For a heap-organized table with one or more LOB columns, if you omit the `TABLESPACE` clause for LOB storage, then the database creates the LOB data and index segments in the tablespace where the table is created.

For an index-organized table with one or more LOB columns, if you omit `TABLESPACE`, then the LOB data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.

For nonpartitioned tables, the value specified for `TABLESPACE` is the actual physical attribute of the segment associated with the table. For partitioned tables, the value specified for `TABLESPACE` is the default physical attribute of the segments associated with all partitions specified in the `CREATE` statement and on subsequent `ALTER TABLE` ... `ADD PARTITION` statements, unless you specify `TABLESPACE` in the `PARTITION` description.

> **See Also:**
>
> CREATE TABLESPACE for more information on tablespaces

TABLESPACE SET

This clause is valid only when creating a sharded table by specifying the `SHARDED` keyword of `CREATE TABLE`. Use this clause to specify the tablespace set in which Oracle Database creates the table.

logging_clause

Specify whether the creation of the table and of any indexes required because of constraints, partition, or LOB storage characteristics will be logged in the redo log file (`LOGGING`) or not (`NOLOGGING`).The logging attribute of the table is independent of that of its indexes.

This attribute also specifies whether subsequent direct loader (SQL*Loader) and direct-path `INSERT` operations against the table, partition, or LOB storage are logged (`LOGGING`) or not logged (`NOLOGGING`).

Refer to *logging_clause* for a full description of this clause.

table_compression

The *`table_compression`* clause is valid only for heap-organized tables. Use this clause to instruct the database whether to compress data segments to reduce disk use. The `COMPRESS` clauses enable table compression. The `NOCOMPRESS` clause disables table compression. The default is `NOCOMPRESS`.

COMPRESS

Specifying only the keyword `COMPRESS` is equivalent to specifying `ROW STORE COMPRESS BASIC` and enables basic table compression.

ROW STORE COMPRESS BASIC

When you enable table compression by specifying either ROW STORE COMPRESS or ROW STORE COMPRESS BASIC, you enable **basic table compression**. Oracle Database attempts to compress data during direct-path INSERT operations when it is productive to do so. The original import utility (imp) does not support direct-path INSERT, and therefore cannot import data in a compressed format.

Tables with basic table compression use a PCTFREE value of 0 to maximize compression, unless you explicitly set a value for PCTFREE in the *physical_attributes_clause*.

In earlier releases, basic table compression was enabled using COMPRESS BASIC. This syntax is still supported for backward compatibility.

> **See Also:**
>
> "Conventional and Direct-Path INSERT" for information on direct-path INSERT operations, including restrictions

ROW STORE COMPRESS ADVANCED

When you enable table compression by specifying ROW STORE COMPRESS ADVANCED, you enable **Advanced Row Compression**. Oracle Database compresses data during all DML operations on the table. This form of compression is recommended for OLTP environments.

Tables with ROW STORE COMPRESS ADVANCED or NOCOMPRESS use the PCTFREE default value of 10, to maximize compress while still allowing for some future DML changes to the data, unless you override this default explicitly.

In earlier releases, Advanced Row Compression was called OLTP table compression and was enabled using COMPRESS FOR OLTP. This syntax is still supported for backward compatibility.

COLUMN STORE COMPRESS FOR { QUERY | ARCHIVE }

When you specify COLUMN STORE COMPRESS FOR QUERY or COLUMN STORE COMPRESS FOR ARCHIVE, you enable **Hybrid Columnar Compression**. With Hybrid Columnar Compression, data can be compressed during direct-path inserts, conventional inserts, and array inserts. During the load process, data is transformed into a column-oriented format and then compressed. Oracle Database uses a compression algorithm appropriate for the level you specify. In general, the higher the level, the greater the compression ratio. Hybrid Columnar Compression can result in higher compression ratios, at a greater CPU cost. Therefore, this form of compression is recommended for data that is not frequently updated.

COLUMN STORE COMPRESS FOR QUERY is useful in data warehousing environments. Valid values are LOW and HIGH, with HIGH providing a higher compression ratio. The default is HIGH.

COLUMN STORE COMPRESS FOR ARCHIVE uses higher compression ratios than COLUMN STORE COMPRESS FOR QUERY, and is useful for compressing data that will be stored for long periods of time. Valid values are LOW and HIGH, with HIGH providing the highest possible compression ratio. The default is LOW.

Specifying `COLUMN STORE COMPRESS` is equivalent to specifying `COLUMN STORE COMPRESS FOR QUERY HIGH`.

Tables with `COLUMN STORE COMPRESS FOR QUERY` or `COLUMN STORE COMPRESS FOR ARCHIVE` use a `PCTFREE` value of 0 to maximize compression, unless you explicitly set a value for `PCTFREE` in the *`physical_attributes_clause`*. For these tables, `PCTFREE` has no effect for blocks loaded using direct-path `INSERT`. `PCTFREE` is honored for blocks loaded using conventional `INSERT`, and for blocks created as a result of DML operations on blocks originally loaded using direct-path `INSERT`.

[NO] ROW LEVEL LOCKING

If you specify `ROW LEVEL LOCKING`, then Oracle Database uses row-level locking during DML operations. This improves the performance of these operations when accessing Hybrid Columnar Compressed data. If you specify `NO ROW LEVEL LOCKING`, then row-level locking is not used. The default is `NO ROW LEVEL LOCKING`.

In earlier releases, Hybrid Columnar Compression was enabled using `COMPRESS FOR QUERY` and `COMPRESS FOR ARCHIVE`. This syntax is still supported for backward compatibility.

> **See Also:**
>
> *Oracle Database Concepts* for more information on Hybrid Columnar Compression, which is a feature of certain Oracle storage systems

Notes on Table Compression

You can specify table compression for the following portions of a heap-organized table:

- For an entire table, in the *`physical_properties`* clause of *`relational_table`* or *`object_table`*
- For a range partition, in the *`table_partition_description`* of the *`range_partitions`* clause
- For a composite range partition, in the *`table_partition_description`* of the *`range_partition_desc`* clause
- For a composite list partition, in the *`table_partition_description`* of the *`list_partition_desc`* clause
- For a list partition, in the *`table_partition_description`* of the *`list_partitions`* clause
- For a system or reference partition, in the *`table_partition_description`* of the *`reference_partition_desc`* clause
- For the storage table of a nested table, in the *`nested_table_col_properties`* clause

  > **See Also:**
  >
  > *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_COMPRESSION` package, which helps you choose the correct compression level for an application, and *Oracle Database Administrator's Guide* for more information about table compression, including examples

Restrictions on Table Compression

Table compression is subject to the following restrictions:

- Data segments of BasicFiles LOBs are not compressed. For information on compression of SecureFiles LOBs, see LOB_compression_clause.
- You cannot drop a column from a table that uses `COMPRESS BASIC`, although you can set such a column as unused. All of the operations of the `ALTER TABLE` ... *`drop_column_clause`* are valid for tables that use `ROW STORE COMPRESS ADVANCED`, `COLUMN STORE COMPRESS FOR QUERY`, and `COLUMN STORE COMPRESS FOR ARCHIVE`.
- You cannot specify any type of table compression for an index-organized table, any overflow segment or partition of an overflow segment, or any mapping table segment of an index-organized table.
- You cannot specify any type of table compression for external tables or for tables that are part of a cluster.
- You cannot specify any type of table compression for tables with `LONG` or `LONG RAW` columns, tables that are owned by the `SYS` schema and reside in the `SYSTEM` tablespace, or tables with `ROWDEPENDENCIES` enabled.
- You cannot specify Hybrid Columnar Compression on tables that are enabled for flashback archiving.
- You cannot specify Hybrid Columnar Compression on the following object-relational features: object tables, `XMLType` tables, columns with abstract data types, collections stored as tables, or OPAQUE types, including `XMLType` columns stored as objects.
- When you update a row in a table compressed with Hybrid Columnar Compression, the `ROWID` of the row may change.
- In tables compressed with Hybrid Columnar Compression, updates to a single row may result in locks on multiple rows. Concurrency for write transactions may therefore be affected.
- If a table compressed with Hybrid Columnar Compression has a foreign key constraint, and you insert data using `INSERT` with the `APPEND` hint, then the data will be compressed to a lesser level than is typical with Hybrid Columnar Compression. To compress the data with Hybrid Columnar Compression, disable the foreign key constraint, insert the data using `INSERT` with the `APPEND` hint, and then reenable the foreign key constraint.

inmemory_table_clause

Use this clause to enable or disable the table for the In-Memory Column Store (IM column store). The IM column store is an optional, static SGA pool that stores copies of tables and partitions in a special columnar format optimized for rapid scans. The IM column store does not replace the buffer cache, but acts as a supplement so that both memory areas can store the same data in different formats.

- Specify `INMEMORY` to enable the table for the IM column store.

  You can optionally use the *`inmemory_attributes`* clause to specify how table data is stored in the IM column store. This clause enables you to specify the data compression method and the data population priority. In an Oracle RAC environment, it also enables you to specify how the data is distributed and duplicated across Oracle RAC instances. Refer to the *inmemory_attributes* clause for more information.

- Specify `NO INMEMORY` to disable the table for the IM column store.
- Specify the *`inmemory_column_clause`* to enable or disable specific table columns for the IM column store, and to specify the data compression method for specific columns. Refer to the *inmemory_column_clause* for more information.

If you omit this clause, then the table is assigned the default IM column store settings for the tablespace in which it is created. Refer to the *inmemory_clause* of `CREATE TABLESPACE` for more information on specifying the default IM column store settings for a tablespace.

In an Oracle Active Data Guard environment, if you specify this clause for a table on the primary database, then the table is enabled or disabled for the IM column store in the Oracle Active Data Guard instance.

> **Note:**
>
> The `INMEMORY_CLAUSE_DEFAULT` initialization parameter enables you to specify a default IM column store clause for new tables and materialized views. Refer to *Oracle Database Reference* for more information on the `INMEMORY_CLAUSE_DEFAULT` initialization parameter.

Restrictions on the In-Memory Column Store

The following restrictions apply to the In-Memory Column Store:

- You cannot specify the `INMEMORY` clause for index-organized tables or external tables.
- You cannot specify the `INMEMORY` clause for tables that are owned by the `SYS` schema and reside in the `SYSTEM` or `SYSAUX` tablespace.
- The IM column store does not support `LONG` or `LONG RAW` columns, out-of-line columns (LOBs, varrays, nested table columns), or extended data type columns. If you enable a table for the IM column store and it contains any of these types of columns, then the columns will not be populated in the IM column store.
- If you enable a table for the IM column store and it contains a virtual column, then the column will be populated in the IM column store only if the value of the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter is `ENABLED` and the SQL expression for the virtual column refers only to columns that are enabled for the IM column store.

> **See Also:**
>
> *Oracle Database In-Memory Guide* for an overview of the IM column store

inmemory_attributes

Use the *`inmemory_memcompress`*, *`inmemory_priority`*, *`inmemory_distribute`*, and *`inmemory_duplicate`* clauses to specify how table data is stored in the IM column store.

inmemory_memcompress

Use this clause to specify the compression method for table data stored in the IM column store. This data is called In-Memory data. To instruct the database to not compress In-Memory data, specify `NO MEMCOMPRESS`. To instruct the database to compress In-Memory data, specify `MEMCOMPRESS FOR` followed by one of the following methods:

- `DML` - This method is optimized for DML operations and performs little or no data compression.
- `QUERY` - Specifying `QUERY` is equivalent to specifying `QUERY LOW`.
- `QUERY LOW` - This method compresses In-Memory data the least (except for `DML`) and results in the best query performance. This is the default.
- `QUERY HIGH` -This method compress In-Memory data more than `QUERY LOW`, but less than `CAPACITY LOW`.
- `CAPACITY` - Specifying `CAPACITY` is equivalent to specifying `CAPACITY LOW`.
- `CAPACITY LOW` - This method compresses In-Memory data more than `QUERY HIGH`, but less than `CAPACITY HIGH`, and results in excellent query performance.
- `CAPACITY HIGH` - This method compresses In-Memory data the most and results in good query performance.

inmemory_priority

Use the `PRIORITY` clause to specify the data population priority for table data in the IM column store. This clause controls the priority of population, but not the speed of population.

- Specify `NONE` for **on-demand population**. In this case, the database populates table data in the IM column store when the table it is accessed through a full table scan. If the table is never accessed, or if it is accessed only through an index scan or fetch by rowid, then population never occurs. This is the default.
- Specify one of the following priority levels for **priority-based population**: `LOW`, `MEDIUM`, `HIGH`, or `CRITICAL`. In this case, the database automatically populates table data in the IM column store using an internally managed priority queue; a full scan is not a necessary condition for population. The database queues population of the table data based on the specified priority level. For example, a table with the setting `INMEMORY PRIORITY CRITICAL` takes precedence over a table with the setting `INMEMORY PRIORITY HIGH`, which in turn takes precedence over a table with the setting `INMEMORY PRIORITY LOW`, and so on. If the IM column store has insufficient space, then the database does not populate additional table data until space is available.

inmemory_distribute

The `DISTRIBUTE` clause is applicable only if you are using Oracle Real Application Clusters (Oracle RAC) or Oracle Active Data Guard. It lets you specify how table data in the IM column store is distributed across Oracle RAC instances, and lets you specify the database instances in which the data is eligible to be populated.

AUTO and BY

Use the AUTO and BY clauses to specify how table data in the IM column store is distributed across Oracle RAC instances. You can specify the following options:

- AUTO - Oracle Database controls how data is distributed across Oracle RAC instances. Large tables are distributed across Oracle RAC instances depending on their access patterns. Smaller tables may be distributed between instances. This is the default.
- BY ROWID RANGE - Data in certain ranges of rowids is distributed to different Oracle RAC instances.
- BY PARTITION - Data in partitions is distributed to different Oracle RAC instances.
- BY SUBPARTITION - Data in subpartitions is distributed to different Oracle RAC instances.

You can only use AUTO and BY to distribute the In-Memory Compression Units (IMCUs) for an object between instances in a single Oracle RAC database, not between a primary instance and standby instance in Active Data Guard.

FOR SERVICE

Use the FOR SERVICE clause to specify the Oracle RAC or Oracle Active Data Guard instances in which the object is eligible to be populated. You can specify the following options:

- DEFAULT - The object is eligible for population on all instances specified with the PARALLEL_INSTANCE_GROUP initialization parameter. If this parameter is not set, then the object is populated on all instances. This is the default.
- ALL - The object is eligible for population on all instances, regardless of the value of the PARALLEL_INSTANCE_GROUP initialization parameter.
- *service_name* - The object is eligible for population only on instances belonging to the specified service and only when the service is active and not blocked on an instance.
- NONE - The object is not eligible for population on any instances. This option lets you disable IM column store population while preserving the other In-Memory attributes for the table. These attributes take effect if you subsequently enable IM column store population for the table by specifying FOR SERVICE DEFAULT, FOR SERVICE ALL, or FOR SERVICE *service_name* in the *inmemory_distribute* clause of an ALTER TABLE statement.

In Oracle RAC, the FOR SERVICE clause specifies the instances within the Oracle RAC database. In Active Data Guard, the primary and standby databases may use a single-instance or Oracle RAC configuration. In Active Data Guard, the FOR SERVICE clause specifies instances in the primary database, instances in the standby database, or a mixture of primary and standby instances.

inmemory_duplicate

The DUPLICATE clause is applicable only if you are using Oracle Real Application Clusters (Oracle RAC) on an engineered system. It controls how table data in the IM column store is duplicated across Oracle RAC instances. You can specify the following options:

- DUPLICATE - Data is duplicated on one Oracle RAC instance, resulting in the data existing on a total of two Oracle RAC instances.
- DUPLICATE ALL - Data is duplicated across all Oracle RAC instances. If you specify DUPLICATE ALL, then the database uses the DISTRIBUTE AUTO setting, regardless of whether or how you specify the *inmemory_distribute* clause.
- NO DUPLICATE - Data is not duplicated across Oracle RAC instances. This is the default.

inmemory_column_clause

Use this clause to enable or disable specific table columns for the IM column store, and to specify the data compression method for specific columns. If you specify this clause when creating a `NO INMEMORY` table, then the column settings will take effect when the table or partition is subsequently enabled for the IM column store.

- Specify `INMEMORY` to enable the specified table columns for the IM column store.

  You can optionally use the *`inmemory_memcompress`* clause to specify the data compression method for specific columns. See *inmemory_memcompress*. If you omit the *`inmemory_memcompress`* clause, then the table column uses the data compression method for the table. You cannot specify the `PRIORITY`, `DISTRIBUTE`, or `DUPLICATE` settings for a specific table column. These settings are the same for all table columns as they are for the table.
- Specify `NO INMEMORY` to disable the specified table columns for the IM column store.

If you omit the *`inmemory_column_clause`*, then all table columns use the IM column store settings for the table.

Restrictions on *inmemory_column_clause*

- You cannot specify this clause for a `LONG` or `LONG RAW` column, an out-of-line column (LOB, varray, nested table column), or an extended data type column.
- To selectively enable a virtual column for the IM column store, the value of the `INMEMORY_VIRTUAL_COLUMNS` initialization parameter must be `ENABLED` or `MANUAL`, and the SQL expression for the virtual column must refer only to columns that are enabled for the IM column store.

inmemory_clause

Use this clause to enable or disable a table partition for the IM column store. In order to specify this clause, the table must be enabled for the IM column store. If you omit this clause, then the table partition uses the IM column store settings for the table.

The *`inmemory_attributes`* clause has the same semantics for table partitions as for tables. Refer to the *inmemory_attributes* clause for full information.

ilm_clause

Use this clause to add an Automatic Data Optimization policy to *`table`*.

This clause has the same semantics in `CREATE TABLE` and `ALTER TABLE`, with the following additional restriction: You can specify only the `ADD POLICY` clause for `CREATE TABLE`. Refer to the *ilm_clause* for the full semantics of this clause.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on managing policies for Automatic Data Optimization

Restrictions on Automatic Data Optimization

Automatic Data Optimization is subject to the following restrictions:

- Automatic Data Optimization is not supported for tables that contain object types, index-organized tables, clustered tables, or materialized views.
- Row-level policies are not supported for tables that support Temporal Validity or tables that are enabled for row archiving for In-Database Archiving.

ilm_policy_clause

Use this clause to describe the Automatic Data Optimization policy.

This clause has the same semantics in `CREATE TABLE` and `ALTER TABLE`. Refer to *ilm_policy_clause* for the full semantics of this clause.

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with `LOGGING` and `NOLOGGING`, respectively. Although `RECOVERABLE` and `UNRECOVERABLE` are supported for backward compatibility, Oracle strongly recommends that you use the `LOGGING` and `NOLOGGING` keywords.

Restrictions on [UN]RECOVERABLE

This clause is subject to the following restrictions:

- You cannot specify `RECOVERABLE` for partitioned tables or LOB storage characteristics.
- You cannot specify `UNRECOVERABLE` for partitioned or index-organized tables.
- You can specify `UNRECOVERABLE` only with `AS` *`subquery`*.

ORGANIZATION

The `ORGANIZATION` clause lets you specify the order in which the data rows of the table are stored.

HEAP

`HEAP` indicates that the data rows of *`table`* are stored in no particular order. This is the default.

INDEX

`INDEX` indicates that *`table`* is created as an index-organized table. In an index-organized table, the data rows are held in an index defined on the primary key for the table.

EXTERNAL

`EXTERNAL` indicates that table is a read-only table located outside the database.

> **See Also:**
>
> "External Table Example"

index_org_table_clause

Use the *`index_org_table_clause`* to create an index-organized table. Oracle Database maintains the table rows, both primary key column values and nonkey column values, in an index built on the primary key. Index-organized tables are therefore best suited for primary key-based access and manipulation. An index-organized table is an alternative to:

- A noncluster table indexed on the primary key by using the `CREATE INDEX` statement
- A cluster table stored in an indexed cluster that has been created using the `CREATE CLUSTER` statement that maps the primary key for the table to the cluster key

You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row. The primary key cannot be `DEFERRABLE`. Use the primary key instead of the rowid for directly accessing index-organized rows.

If an index-organized table is partitioned and contains LOB columns, then you should specify the *`index_org_table_clause`* first, then the *`LOB_storage_clause`*, and then the appropriate *`table_partitioning_clauses`*.

You cannot use the `TO_LOB` function to convert a `LONG` column to a LOB column in the subquery of a `CREATE TABLE` ... `AS SELECT` statement if you are creating an index-organized table. Instead, create the index-organized table without the `LONG` column, and then use the `TO_LOB` function in an `INSERT` ... `AS SELECT` statement.

The `ROWID` pseudocolumn of an index-organized table returns logical rowids instead of physical rowids. A column that you create with the data type `ROWID` cannot store the logical rowids of the IOT. The only data you can store in a column of type `ROWID` is rowids from heap-organized tables. If you want to store the logical rowids of an IOT, then create a column of type `UROWID` instead. A column of type `UROWID` can store both physical and logical rowids.

> **See Also:**
>
> "Index-Organized Table Example"

Restrictions on Index-Organized Tables

Index-organized tables are subject to the following restrictions:

- You cannot define a virtual column for an index-organized table.
- You cannot specify the *`composite_range_partitions`*, *`composite_list_partitions`*, or *`composite_hash_partitions`* clauses for an index-organized table.
- If the index-organized table is a nested table or varray, then you cannot specify *`table_partitioning_clauses`*.
- The collations of character data type columns belonging to the primary key of an index-organized table must be `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.

PCTTHRESHOLD *integer*

Specify the percentage of space reserved in the index block for an index-organized table row. `PCTTHRESHOLD` must be large enough to hold the primary key. All trailing columns of a row, starting with the column that causes the specified threshold to be exceeded, are stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. If you do not specify PCTTHRESHOLD, then the default is 50.

Restriction on PCTTHRESHOLD

You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

mapping_table_clauses

Specify MAPPING TABLE to instruct the database to create a mapping of local to physical ROWIDs and store them in a heap-organized table. This mapping is needed in order to create a bitmap index on the index-organized table. If the index-organized table is partitioned, then the mapping table is also partitioned and its partitions have the same name and physical attributes as the base table partitions.

Oracle Database creates the mapping table or mapping table partition in the same tablespace as its parent index-organized table or partition. You cannot query, perform DML operations on, or modify the storage characteristics of the mapping table or its partitions.

prefix_compression

The *prefix_compression* clauses let you enable or disable prefix compression for index-organized tables.

- Specify COMPRESS to enable **prefix compression**, also known as key compression, for an index-organized table, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length, which is the number of prefix columns to compress.

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.
- Specify NOCOMPRESS to disable prefix compression in index-organized tables. This is the default.

Restriction on Prefix Compression of Index-organized Tables

At the partition level, you can specify COMPRESS, but you cannot specify the prefix length with *integer*.

index_org_overflow_clause

The *index_org_overflow_clause* lets you instruct the database that index-organized table data rows exceeding the specified threshold are placed in the data segment specified in this clause.

- When you create an index-organized table, Oracle Database evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, then the database raises an error and does not execute the CREATE TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.
- All physical attributes and storage characteristics you specify in this clause after the OVERFLOW keyword apply only to the overflow segment of the table. Physical attributes and storage characteristics for the index-organized table itself, default values for all its partitions, and values for individual partitions must be specified before this keyword.

- If the index-organized table contains one or more LOB columns, then the LOBs will be stored out-of-line unless you specify OVERFLOW, even if they would otherwise be small enough be to stored inline.
- If *table* is partitioned, then the database equipartitions the overflow data segments with the primary key index segments.

INCLUDING *column_name*

Specify a column at which to divide an index-organized table row into index and overflow portions. The primary key columns are always stored in the index. *column_name* can be either the last primary key column or any non primary key column. All non primary key columns that follow *column_name* are stored in the overflow data segment.

If an attempt to divide a row at *column_name* causes the size of the index portion of the row to exceed the specified or default PCTTHRESHOLD value, then the database breaks up the row based on the PCTTHRESHOLD value.

Restriction on the INCLUDING Clause

You cannot specify this clause for individual partitions of an index-organized table.

external_table_clause

Use the *external_table_clause* to create an external table, which is a read-only table whose metadata is stored in the database but whose data in stored outside the database. Among other capabilities, external tables let you query data without first loading it into the database.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide*, *Oracle Database Administrator's Guide*, and *Oracle Database Utilities* for information on the uses for external tables

Because external tables have no data in the database, you define them with a small subset of the clauses normally available when creating tables.

- Within the *relational_properties* clause, you can specify only *column*, *datatype*, ENCRYPT, *inline_constraint*, and *out_of_line_constraint*. You can specify the ENCRYPT clause only when you specify the ORACLE_DATAPUMP access driver and the AS *subquery* clause to load data into the external table. Within the *inline_constraint* and *out_of_line_constraint* clauses, you can specify all subclauses except CHECK.
- Within the *physical_properties_clause*, you can specify only the organization of the table (ORGANIZATION EXTERNAL *external_table_clause*).
- Within the *table_properties* clause, you can specify the *parallel_clause*. The *parallel_clause* lets you parallelize subsequent queries on the external data and subsequent operations that populate the external table.

  Starting with Oracle Database 12*c* Release 2 (12.2), you can create a **partitioned external table**. To do this, within the *table_properties* clause, you can specify the following subclauses of the *table_partitioning_clauses* :

- *range_partitions* - specify this clause to create a range-partitioned or interval-partitioned external table
- *list_partitions* - specify this clause to create a list-partitioned external table. Within this clause, you cannot specify the AUTOMATIC clause; an automatic list-partitioned table cannot be an external table.
- *composite_range_partitions* - specify this clause to create a range-range, range-list, interval-range, or interval-list composite-partitioned external table
- *composite_list_partitions* - specify this clause to create a list-range or list-list composite-partitioned external table. Within this clause, you cannot specify the AUTOMATIC clause; an automatic composite-partitioned table cannot be an external table.

- You can populate the external table at create time by using the AS *subquery* clause.

No other clauses are permitted in the same CREATE TABLE statement.

See Also:

- "External Table Example"
- ALTER TABLE ... "PROJECT COLUMN Clause" for information on the effect of changing the default property of the column projection

Restrictions on External Tables

External tables are subject to the following restrictions:

- An external table cannot be a temporary table.
- You can specify only the following types of constraints on an external table: NOT NULL constraints, unique constraints, primary key constraints, and foreign key constraints. When you specify unique constraints, primary key constraints, or foreign key constraints, you must also specify RELY DISABLE. These constraints are declarative and are not enforced. They can increase query performance and reduce resource consumption because more optimizer transformations can be taken into account. In order for the optimizer to utilize these RELY DISABLE constraints, the QUERY_REWRITE_INTEGRITY initialization parameter must be set to either trusted or stale_tolerated.
- You cannot create an index on an external table.
- An external table cannot contain INVISIBLE columns.
- An external table cannot have object type, varray, or LONG columns. However, you can populate LOB columns of an external table with varray or LONG data from an internal database table.

TYPE

TYPE *access_driver_type* indicates the **access driver** of the external table. The access driver is the API that interprets the external data for the database. Oracle Database provides two access drivers: ORACLE_LOADER, ORACLE_DATAPUMP, ORACLE_HDFS, and ORACLE_HIVE. If you do not specify TYPE, then the database uses ORACLE_LOADER as the default access driver. You must specify the ORACLE_DATAPUMP access driver if you specify the AS *subquery* clause to unload data from one Oracle Database and reload it into the same or a different Oracle Database.

> **See Also:**
>
> *Oracle Database Utilities* for information about the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers

DEFAULT DIRECTORY

`DEFAULT DIRECTORY` lets you specify a default directory object corresponding to a directory on the file system where the external data sources may reside. The default directory can also be used by the access driver to store auxiliary files such as error logs.

ACCESS PARAMETERS

The optional `ACCESS PARAMETERS` clause lets you assign values to the parameters of the specific access driver for this external table.

- The *`opaque_format_spec`* specifies all access parameters for the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access drivers. See *Oracle Database Utilities* for descriptions of the `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, and `ORACLE_HIVE` access parameters.

  Field names specified in the *`opaque_format_spec`* must match columns in the table definition. Oracle Database ignores any field in the *`opaque_format_spec`* that is not matched by a column in the table definition.
- `USING CLOB` *`subquery`* lets you derive the parameters and their values through a subquery. The subquery cannot contain any set operators or an `ORDER BY` clause. It must return one row containing a single item of data type `CLOB`.

Whether you specify the parameters in an *`opaque_format_spec`* or derive them using a subquery, the database does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

LOCATION

The `LOCATION` clause lets you specify one or more external data sources. Usually the *`location_specifier`* is a file, but it need not be. Oracle Database does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.

You must specify the `LOCATION` clause as follows:

- When creating a nonpartitioned external table, you must specify the `LOCATION` clause at the table level in the *`external_table_data_props`* clause.
- When creating a partitioned external table, you must specify the `LOCATION` clause at the partition level in the *`external_part_subpart_data_props`* clause.
- When creating a composite-partitioned external table, you must specify the `LOCATION` clause at the subpartition level in the *`external_part_subpart_data_props`* clause.

REJECT LIMIT

The `REJECT LIMIT` clause lets you specify how many conversion errors can occur during a query of the external data before an Oracle Database error is returned and the query is aborted. The default value is 0.

CLUSTER Clause

The `CLUSTER` clause indicates that the table is to be part of *`cluster`*. The columns listed in this clause are the table columns that correspond to the cluster columns. Generally, the cluster columns of a table are the column or columns that make up its primary key or a portion of its primary key. Refer to CREATE CLUSTER for more information.

Specify one column from the table for each column in the cluster key. The columns are matched by position, not by name.

A cluster table uses the space allocation of the cluster. Therefore, do not use the `PCTFREE`, `PCTUSED`, or `INITRANS` parameters, the `TABLESPACE` clause, or the *`storage_clause`* with the `CLUSTER` clause.

Restrictions on Cluster Tables

Cluster tables are subject to the following restrictions:

- Object tables and tables containing LOB columns or columns of the `Any*` Oracle-supplied types cannot be part of a cluster.
- You cannot specify the *`parallel_clause`* or `CACHE` or `NOCACHE` for a table that is part of a cluster.
- You cannot specify `CLUSTER` with either `ROWDEPENDENCIES` or `NOROWDEPENDENCIES` unless the cluster has been created with the same `ROWDEPENDENCIES` or `NOROWDEPENDENCIES` setting.
- A cluster table cannot contain `INVISIBLE` columns.

table_properties

The *`table_properties`* further define the characteristics of the table.

column_properties

Use the *`column_properties`* clauses to specify the storage attributes of a column.

object_type_col_properties

The *`object_type_col_properties`* determine storage characteristics of an object column or attribute or of an element of a collection column or attribute.

column

For *`column`*, specify an object column or attribute.

substitutable_column_clause

The *`substitutable_column_clause`* indicates whether object columns or attributes in the same hierarchy are substitutable for each other. You can specify that a column is of a particular type, or whether it can contain instances of its subtypes, or both.

- If you specify `ELEMENT`, then you constrain the element type of a collection column or attribute to a subtype of its declared type.

- The IS OF [TYPE] (ONLY *type*) clause constrains the type of the object column to a subtype of its declared type.
- NOT SUBSTITUTABLE AT ALL LEVELS indicates that the object column cannot hold instances corresponding to any of its subtypes. Also, substitution is disabled for any embedded object attributes and elements of embedded nested tables and varrays. The default is SUBSTITUTABLE AT ALL LEVELS.

Restrictions on the *substitutable_column_clause*

This clause is subject to the following restrictions:

- You cannot specify this clause for an attribute of an object column. However, you can specify this clause for a object type column of a relational table and for an object column of an object table if the substitutability of the object table itself has not been set.
- For a collection type column, the only part of this clause you can specify is [NOT] SUBSTITUTABLE AT ALL LEVELS.

LOB_storage_clause

The *LOB_storage_clause* lets you specify the storage attributes of LOB data segments. You must specify at least one clause after the STORE AS keywords. If you specify more than one clause, then you must specify them in the order shown in the syntax diagram, from top to bottom.

For a nonpartitioned table, this clause specifies the storage attributes of LOB data segments of the table.

For a partitioned table, Oracle Database implements this clause depending on where it is specified:

- For a partitioned table specified at the table level—when specified in the *physical_properties* clause along with one of the partitioning clauses—this clause specifies the default storage attributes for LOB data segments associated with each partition or subpartition. These storage attributes apply to all partitions or subpartitions unless overridden by a *LOB_storage_clause* at the partition or subpartition level.
- For an individual partition of a partitioned table—when specified as part of a *table_partition_description*—this clause specifies the storage attributes of the data segments of the partition or the default storage attributes of any subpartitions of the partition. A partition-level *LOB_storage_clause* overrides a table-level *LOB_storage_clause*.
- For an individual subpartition of a partitioned table—when specified as part of *subpartition_by_hash* or *subpartition_by_list*—this clause specifies the storage attributes of the data segments of the subpartition. A subpartition-level *LOB_storage_clause* overrides both partition-level and table-level *LOB_storage_clauses*.

Restriction on the *LOB_storage_clause*:

Only the TABLESPACE clause is allowed when specifying the *LOB_storage_clause* in a subpartition.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for detailed information about LOBs, including guidelines for creating gigabyte LOBs
> - "Creating a Table: LOB Column Example"

LOB_item

Specify the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table. Oracle Database automatically creates a system-managed index for each `LOB_item` you create.

SECUREFILE | BASICFILE

Use this clause to specify the type of LOB storage, either high-performance LOB (SecureFiles), or the traditional LOB (BasicFiles).

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about SecureFiles LOBs

> **Note:**
>
> You cannot convert a LOB from one type of storage to the other. Instead you must migrate to SecureFiles or BasicFiles by using online redefinition or partition exchange.

LOB_segname

Specify the name of the LOB data segment. You cannot use `LOB_segname` if you specify more than one `LOB_item`.

LOB_storage_parameters

The `LOB_storage_parameters` clause lets you specify various elements of LOB storage.

TABLESPACE Clause

Use this clause to specify the tablespace in which LOB data is to be stored.

TABLESPACE SET Clause

This clause is valid only when creating a sharded table by specifying the `SHARDED` keyword of `CREATE TABLE`. Use this clause to specify the tablespace set in which LOB data is to be stored.

storage_clause

Use the *storage_clause* to specify various aspects of LOB segment storage. Of particular interest in the context of LOB storage is the MAXSIZE clause of the *storage_clause*, which can be used in combination with the *LOB_retention_clause* of *LOB_parameters*. Refer to *storage_clause* for more information.

LOB_parameters

Several of the *LOB_parameters* are no longer needed if you are using SecureFiles for LOB storage. The PCTVERSION and FREEPOOLS parameters are valid and useful only if you are using BasicFiles LOB storage.

ENABLE STORAGE IN ROW

If you enable storage in row, then the LOB value is stored in the row (inline) if its length is less than approximately 4000 bytes minus system control information. This is the default.

Restriction on Enabling Storage in Row

For an index-organized table, you cannot specify this parameter unless you have specified an OVERFLOW segment in the *index_org_table_clause*.

DISABLE STORAGE IN ROW

If you disable storage in row, then the LOB value is stored outside of the row out of line regardless of the length of the LOB value.

The LOB locator is always stored inline regardless of where the LOB value is stored. You cannot change the value of STORAGE IN ROW once it is set except by moving the table. See the *move_table_clause* in the ALTER TABLE documentation for more information.

CHUNK *integer*

Specify the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, then the database rounds up in bytes to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, then the database allocates 4096 bytes (2 blocks). The maximum value is 32768 (32K), which is the largest Oracle Database block size allowed. The default CHUNK size is one Oracle Database block.

The value of CHUNK must be less than or equal to the value of NEXT, either the default value or that specified in the *storage_clause*. If CHUNK exceeds the value of NEXT, then the database returns an error. You cannot change the value of CHUNK once it is set.

PCTVERSION *integer*

Specify the maximum percentage of overall LOB storage space used for maintaining old versions of the LOB. If the database is running in manual undo mode, then the default value is 10, meaning that older versions of the LOB data are not overwritten until they consume 10% of the overall LOB storage space.

You can specify the PCTVERSION parameter whether the database is running in manual or automatic undo mode. PCTVERSION is the default in manual undo mode. RETENTION is the default in automatic undo mode. You cannot specify both PCTVERSION and RETENTION.

This clause is not valid if you have specified SECUREFILE. If you specify both SECUREFILE and PCTVERSION, then the database silently ignores the PCTVERSION parameter.

LOB_retention_clause

Use this clause to specify whether you want the LOB segment retained for flashback purposes, consistent-read purposes, both, or neither.

You can specify the RETENTION parameter only if the database is running in automatic undo mode. Oracle Database uses the value of the UNDO_RETENTION initialization parameter to determine the amount of committed undo data to retain in the database. In automatic undo mode, RETENTION is the default value unless you specify PCTVERSION. You cannot specify both PCTVERSION and RETENTION.

You can specify the optional settings after RETENTION only if you are using SecureFiles. The SECUREFILE parameter of the *LOB_storage_clause* indicates that the database will use SecureFiles to manage storage dynamically, taking into account factors such as the undo mode of the database.

- Specify MAX to signify that the undo should be retained until the LOB segment has reached MAXSIZE. If you specify MAX, then you must also specify the MAXSIZE clause in the *storage_clause*.
- Specify MIN if the database is in flashback mode to limit the undo retention **duration** for the specific LOB segment to *n* seconds.
- Specify AUTO if you want to retain undo sufficient for consistent read purposes only.
- Specify NONE if no undo is required for either consistent read or flashback purposes.

If you do not specify the RETENTION parameter, or you specify RETENTION with no optional settings, then RETENTION is set to DEFAULT, which is functionally equivalent to AUTO.

See Also:

- CREATE TABLE clause *LOB_storage_parameters* for more information on simplified LOB storage using SecureFiles
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on using SecureFiles
- *flashback_mode_clause* of ALTER DATABASE for information on putting a database in flashback mode
- "Creating an Undo Tablespace: Example"

FREEPOOLS *integer*

Specify the number of groups of free lists for the LOB segment. Normally *integer* will be the number of instances in an Oracle Real Application Clusters environment or 1 for a single-instance database.

You can specify this parameter only if the database is running in automatic undo mode. In this mode, FREEPOOLS is the default unless you specify the FREELIST GROUPS parameter of the *storage_clause*. If you specify neither FREEPOOLS nor FREELIST GROUPS, then the database uses a default of FREEPOOLS 1 if the database is in automatic undo management mode and a default of FREELIST GROUPS 1 if the database is in manual undo management mode.

This clause is not valid if you have specified SECUREFILE. If you specify both SECUREFILE and FREEPOOLS, then the database silently ignores the FREEPOOLS parameter.

Restriction on FREEPOOLS

You cannot specify both FREEPOOLS and the FREELIST GROUPS parameter of the *storage_clause*.

LOB_deduplicate_clause

This clause is valid only for SecureFiles LOBs. Use the *LOB_deduplicate_clause* to enable or disable LOB deduplication, which is the elimination of duplicate LOB data.

The DEDUPLICATE keyword instructs the database to eliminate duplicate copies of LOBs. Using a secure hash index to detect duplication, the database coalesces LOBs with identical content into a single copy, reducing storage consumption and simplifying storage management.

If you omit this clause, then LOB deduplication is disabled by default.

This clause implements LOB deduplication for the entire LOB segment. To enable or disable deduplication for an individual LOB, use the DBMS_LOB.SETOPTIONS procedure.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information about LOB deduplication and *Oracle Database PL/SQL Packages and Types Reference* for information about about the DBMS_LOB package

LOB_compression_clause

This clause is valid only for SecureFiles LOBs, not for BasicFiles LOBs. Use the *LOB_compression_clause* to instruct the database to enable or disable server-side LOB compression. Random read/write access is possible on server-side compressed LOB segments. LOB compression is independent from table compression or index compression. If you omit this clause, then the default is NOCOMPRESS.

You can specify HIGH, MEDIUM, or LOW to vary the degree of compression. The HIGH degree of compression incurs higher latency than MEDIUM but provides better compression. The LOW degree results in significantly higher decompression and compression speeds, at the cost of slightly lower compression ratio than either HIGH or MEDIUM. If you omit this optional parameter, then the default is MEDIUM.

This clause implements server-side LOB compression for the entire LOB segment. To enable or disable compression on an individual LOB, use the DBMS_LOB.SETOPTIONS procedure.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on server-side LOB storage and *Oracle Database PL/SQL Packages and Types Reference* for information about client-side LOB compression using the `UTL_COMPRESS` supplied package and for information about the `DBMS_LOB` package

ENCRYPT | DECRYPT

These clauses are valid only for LOBs that are using SecureFiles for LOB storage. Specify `ENCRYPT` to encrypt all LOBs in the column. Specify `DECRYPT` to keep the LOB in cleartext. If you omit this clause, then `DECRYPT` is the default.

Refer to *encryption_spec* for general information on that clause. When applied to a LOB column, *`encryption_spec`* is specific to the individual LOB column, so the encryption algorithm can differ from that of other LOB columns and other non-LOB columns. Use the *`encryption_spec`* as part of the *`column_definition`* to encrypt the entire LOB column. Use the *`encryption_spec`* as part of the *`LOB_storage_clause`* in the *`table_partition_description`* to encrypt a LOB partition.

Restriction on *encryption_spec* for LOBs

You cannot specify the `SALT` or `NO SALT` clauses of *`encryption_spec`* for LOB encryption.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on LOB encryption and *Oracle Database PL/SQL Packages and Types Reference* for information the `DBMS_LOB` package

CACHE | NOCACHE | CACHE READS

Refer to CACHE | NOCACHE | CACHE READS for information on these clauses.

LOB_partition_storage

The *`LOB_partition_storage`* clause lets you specify a separate *`LOB_storage_clause`* or *`varray_col_properties`* clause for each partition. You must specify the partitions in the order of partition position. You can find the order of the partitions by querying the `PARTITION_NAME` and `PARTITION_POSITION` columns of the `USER_IND_PARTITIONS` view.

If you do not specify a *`LOB_storage_clause`* or *`varray_col_properties`* clause for a particular partition, then the storage characteristics are those specified for the LOB item at the table level. If you also did not specify any storage characteristics for the LOB item at the table level, then Oracle Database stores the LOB data partition in the same tablespace as the table partition to which it corresponds.

Restrictions on *LOB_partition_storage*

`LOB_partition_storage` is subject to the following restrictions:

- In the *`LOB_parameters`* of the *`LOB_storage_clause`*, you cannot specify *`encryption_spec`*, because it is invalid to specify an encryption algorithm for partitions and subpartitions.

- You can only specify the `TABLESPACE` clause for hash partitions and all types of subpartitions.

varray_col_properties

The *`varray_col_properties`* let you specify separate storage characteristics for the LOB in which a varray will be stored. If *`varray_item`* is a multilevel collection, then the database stores all collection items nested within *`varray_item`* in the same LOB in which *`varray_item`* is stored.

- For a nonpartitioned table—when specified in the *`physical_properties`* clause without any of the partitioning clauses—this clause specifies the storage attributes of the LOB data segments of the varray.
- For a partitioned table specified at the table level—when specified in the *`physical_properties`* clause along with one of the partitioning clauses—this clause specifies the default storage attributes for the varray LOB data segments associated with each partition (or its subpartitions, if any).
- For an individual partition of a partitioned table—when specified as part of a *`table_partition_description`*—this clause specifies the storage attributes of the varray LOB data segments of that partition or the default storage attributes of the varray LOB data segments of any subpartitions of this partition. A partition-level *`varray_col_properties`* overrides a table-level *`varray_col_properties`*.
- For an individual subpartition of a partitioned table—when specified as part of *`subpartition_by_hash`* or *`subpartition_by_list`*—this clause specifies the storage attributes of the varray data segments of this subpartition. A subpartition-level *`varray_col_properties`* overrides both partition-level and table-level *`varray_col_properties`*.

STORE AS [SECUREFILE | BASICFILE] LOB Clause

If you specify `STORE AS LOB`, then:

- If the maximum varray size is less than approximately 4000 bytes, then the database stores the varray as an inline LOB unless you have disabled storage in row.
- If the maximum varray size is greater than approximately 4000 bytes or if you have disabled storage in row, then the database stores in the varray as an out-of-line LOB.

If you do not specify `STORE AS LOB`, then storage is based on the maximum possible size of the varray rather than on the actual size of a varray column. The maximum size of the varray is the number of elements times the element size, plus a small amount for system control information. If you omit this clause, then:

- If the maximum size of the varray is less than approximately 4000 bytes, then the database does not store the varray as a LOB, but as inline data.
- If the maximum size is greater than approximately 4000 bytes, then the database always stores the varray as a LOB.
  - If the actual size is less than approximately 4000 bytes, then it is stored as an inline LOB
  - If the actual size is greater than approximately 4000 bytes, then it is stored as an out-of-line LOB, as is true for other LOB columns.

substitutable_column_clause

The `substitutable_column_clause` has the same behavior as described for object_type_col_properties.

> **See Also:**
>
> "Substitutable Table and Column Examples"

Restriction on Varray Column Properties

You cannot specify this clause on an interval partitioned table.

nested_table_col_properties

The `nested_table_col_properties` let you specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. Unless you explicitly specify otherwise in this clause:

- For a nonpartitioned table, the storage table is created in the same schema and the same tablespace as the parent table.
- For a partitioned table, the storage table is created in the default tablespace of the schema. By default, nested tables are equipartitioned with the partitioned base table.
- In either case, the storage table uses default storage characteristics, and stores the nested table values of the column for which it was created.

You must include this clause when creating a table with columns or column attributes whose type is a nested table. Clauses within `nested_table_col_properties` that function the same way they function for the parent table are not repeated here.

nested_item

Specify the name of a column, or of a top-level attribute of the object type of the tables, whose type is a nested table.

COLUMN_VALUE

If the nested table is a multilevel collection, then the inner nested table or varray may not have a name. In this case, specify `COLUMN_VALUE` in place of the `nested_item` name.

> **See Also:**
>
> "Creating a Table: Multilevel Collection Example" for examples using `nested_item` and `COLUMN_VALUE`

LOCAL | GLOBAL

Specify `LOCAL` to equipartition the nested table with the base table. This is the default. Oracle Database automatically creates a local partitioned index for the partitioned nested table.

Specify `GLOBAL` to indicate that the nested table is a nonpartitioned nested table of a partitioned base table.

storage_table

Specify the name of the table where the rows of *nested_item* reside.

You cannot query or perform DML statements on *storage_table* directly, but you can modify its storage characteristics by specifying its name in an ALTER TABLE statement.

> **See Also:**
>
> ALTER TABLE for information about modifying nested table column storage characteristics

RETURN [AS]

Specify what Oracle Database returns as the result of a query.

- VALUE returns a copy of the nested table itself.
- LOCATOR returns a collection locator to the copy of the nested table.

  The locator is scoped to the session and cannot be used across sessions. Unlike a LOB locator, the collection locator cannot be used to modify the collection instance.

If you do not specify the *segment_attributes_clause* or the *LOB_storage_clause*, then the nested table is heap organized and is created with default storage characteristics.

Restrictions on Nested Table Column Properties

Nested table column properties are subject to the following restrictions:

- You cannot specify this clause for a temporary table.
- You cannot specify this clause on an interval partitioned table.
- You cannot specify the *oid_clause*.
- At create time, you cannot use *object_properties* to specify an *out_of_line_ref_constraint*, *inline_ref_constraint*, or foreign key constraint for the attributes of a nested table. However, you can modify a nested table to add such constraints using ALTER TABLE.

> **See Also:**
>
> - ALTER TABLE for information about modifying nested table column storage characteristics
> - "Nested Table Example" and "Creating a Table: Multilevel Collection Example"

XMLType_column_properties

The *XMLType_column_properties* let you specify storage attributes for an XMLTYPE column.

XMLType_storage

XMLType data can be stored in binary XML, CLOB, or object-relational columns.

- Specify BINARY XML to store the XML data in compact binary XML format.

  Any LOB parameters you specify are applied to the underlying BLOB column created for storing the binary XML encoded value.

  In earlier releases, binary XML data is stored by default in a BasicFiles LOB. Beginning with Oracle Database 11*g* Release 2 (11.2.0.2), if the COMPATIBLE initialization parameter is 11.2 or higher and you do not specify BASICFILE or SECUREFILE, then binary XML data is stored in a SecureFiles LOB whenever possible. If SecureFiles LOB storage is not possible then the binary XML data is stored in a BasicFiles LOB. This can occur if either of the following is true:

  – The tablespace for the XMLType table does not use automatic segment space management.

  – A setting in file init.ora prevents SecureFiles LOB storage. For example, see parameter DB_SECUREFILE in *Oracle Database Reference*.

- Specify CLOB if you want the database to store the XMLType data in a CLOB column. Storing data in a CLOB column preserves the original content and enhances retrieval time.

  If you specify LOB storage, then you can specify either LOB parameters or the *XMLSchema_spec* clause, but not both. Specify the *XMLSchema_spec* clause if you want to restrict the table or column to particular schema-based XML instances.

  If you do not specify BASICFILE or SECUREFILE with this clause, then the CLOB column is stored in a BasicFiles LOB.

  > **Note:**
  >
  > Oracle recommends against storing XMLType data in a CLOB column. CLOB storage of XMLType is deprecated. Use binary XML storage of XMLType instead.

- Specify OBJECT RELATIONAL if you want the database to store the XMLType data in object-relational columns. Storing data objects relationally lets you define indexes on the relational columns and enhances query performance.

  If you specify object-relational storage, then you must also specify the *XMLSchema_spec* clause.

Use the ALL VARRAYS AS clause if you want the database to store all varrays in an XMLType column.

In earlier releases, XMLType data is stored in a CLOB column in a BasicFiles LOB by default. Beginning with Oracle Database 11*g* Release 2 (11.2.0.2), if the COMPATIBLE initialization parameter is 11.2 or higher and you do not specify the *XMLType_storage* clause, then XMLType data is stored in a binary XML column in a SecureFiles LOB. If SecureFiles LOB storage is not possible, then it is stored in a binary XML column in a BasicFiles LOB.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information on SecureFiles LOBs

XMLSchema_spec

Refer to the XMLSchema_spec for the full semantics of this clause.

> **See Also:**
>
> - *LOB_storage_clause* for information on the *LOB_segname* and *LOB_parameters* clauses
> - "XMLType Column Examples" for examples of XMLType columns in object-relational tables and "Using XML in SQL Statements " for an example of creating an XMLSchema
> - *Oracle XML DB Developer's Guide* for more information on XMLType columns and tables and on creating XMLSchemas
> - *Oracle Database PL/SQL Packages and Types Reference* for information on the DBMS_XMLSCHEMA package

XMLType_virtual_columns

This clause is valid only for XMLType tables with binary XML storage, which you designate in the *XMLType_storage* clause. Specify the VIRTUAL COLUMNS clause to define virtual columns, which can be used as in a function-based index or in the definition of a constraint. You cannot define a constraint on such a virtual column during creation of the table, but you can use a subsequent ALTER TABLE statement to add a constraint to the column.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for examples of how to use this clause in an XML environment

read_only_clause

This clause lets you specify whether to create a table, partition, or subpartition in read-only or read/write mode.

- Use READ ONLY to specify read-only mode. When an object is in read-only mode, you cannot issue any DML statements that affect the object or any SELECT ... FOR UPDATE ... statements on the object. You can issue DDL statements as long as they do not modify any table data. See *Oracle Database Administrator's Guide* for the complete list of operations that are allowed and disallowed on read-only objects.

- Use `READ WRITE` to specify read/write mode. This is the default.

When you specify this clause for a partitioned table, you specify the default read-only or read/write mode for the table. This mode is assigned to all partitions in the table at creation time, as well as any partitions that are subsequently added to the table, unless you override this behavior by specifying the mode at the partition level.

When you specify this clause for a composite-partitioned table, you specify the default read-only or read/write mode for all partitions in the table. You can override this behavior by specifying this clause for a particular partition. The default mode of a partition is assigned to all subpartitions in the partition at creation time, as well as any subpartitions that are subsequently added to the partition, unless you override this behavior by specifying the mode at the subpartition level.

indexing_clause

The *`indexing_clause`* is valid only for partitioned tables. Use this clause to set the **indexing property** for a table, table partition, or table subpartition.

- Specify `INDEXING ON` to set the indexing property to `ON`. This is the default.
- Specify `INDEXING OFF` to set the indexing property to `OFF`.

The indexing property determines whether table partitions and subpartitions are included in partial indexes on the table.

- For simple partitioned tables, partitions with an indexing property of `ON` are included in partial indexes on the table. Partitions with an indexing property of `OFF` are excluded.
- For composite-partitioned tables, subpartitions with an indexing property of `ON` are included in partial indexes on the table. Subpartitions with an indexing property of `OFF` are excluded.

You can specify the *`indexing_clause`* at the table, partition, or subpartition level. When you specify the *`indexing_clause`* at the table level, in the *`table_properties`* clause, you set the default indexing property for the table. Interval partitions, which are automatically created by the database, always inherit the default indexing property for the table. Other types of partitions and subpartitions inherit the default indexing property as follows:

- For simple partitioned tables, partitions inherit the default indexing property for the table. You can override this behavior by specifying the *`indexing_clause`* for an individual partition:
  - For a range partition, in the *`table_partition_description`* of the *`range_partitions`* clause
  - For a hash partition, in the *`individual_hash_partitions`* clause of the *`hash_partitions`* clause
  - For a list partition, in the *`table_partition_description`* of the *`list_partitions`* clause
  - For a reference partition, in the *`table_partition_description`* of the *`reference_partition_desc`* clause of the *`reference_partitioning`* clause
  - For a system partition, in the *`table_partition_description`* of the *`reference_partition_desc`* clause of the *`system_partitioning`* clause
- For composite-partitioned tables, subpartitions inherit the default indexing property for the table. You can override this behavior by specifying the *`indexing_clause`* for an individual partition or subpartition.

If you specify the *indexing_clause* for a partition, then its subpartitions inherit the indexing property of the partition:

– For composite range partitions, in the *table_partition_description* of the *composite_range_partitions* clause

– For composite list partitions, in the *table_partition_description* of the *composite_list_partitions* clause

– For composite hash partitions, in the *individual_hash_partitions* clause of the *composite_hash_partitions* clause

You can set the indexing property of a subpartition by specifying the *indexing_clause* for the subpartition:

– For range subpartitions, in the *range_subpartition_desc* clause of the *composite_range_partitions* clause

– For list subpartitions, in the *list_subpartition_desc* clause of the *composite_list_partitions* clause

– For hash subpartitions, in the *individual_hash_subparts* clause of the *composite_hash_partitions* clause

See Also:

Oracle Database Reference for information on viewing the indexing property of a table, table partition, or table subpartition.

- To view the default indexing property of a table, query the `DEF_INDEXING` column of the `*_PART_TABLES` views.
- To view the indexing property of a table partition, query the `INDEXING` column of the `*_TAB_PARTITIONS` views.
- To view the indexing property of a table subpartition, query the `INDEXING` column of the `*_TAB_SUBPARTITIONS` views.

Restrictions on the *indexing_clause*

The *indexing_clause* is subject to the following restrictions:

- You cannot specify the *indexing_clause* for nonpartitioned tables.
- You cannot specify the *indexing_clause* for index-organized tables.

See Also:

The *partial_index_clause* of `CREATE INDEX` for more information on partial indexes

table_partitioning_clauses

Use the *table_partitioning_clauses* to create a partitioned table.

Notes on Partitioning in General

The following notes pertain to all types of partitioning:

- You can specify up to a total of 1024K-1 partitions and subpartitions.
- You can create a partitioned table with just one partition. A table with one partition is different from a nonpartitioned table. For example, you cannot add a partition to a nonpartitioned table.
- You can specify a name for every table and LOB partition and for every table and LOB subpartition, but you need not do so. If you specify a name, then it must conform to the rules for naming schema objects and their parts as described in Database Object Naming Rules . If you omit the name, then the database generates names as follows:
  - If you omit a partition name, then the database generates a name of the form `SYS_P`*n*. System-generated names for LOB data and LOB index partitions take the form `SYS_LOB_P`*n* and `SYS_IL_P`*n*, respectively.
  - If you specify a subpartition name in *subpartition_template*, then for each subpartition created with that template, the database generates a name by concatenating the partition name with the template subpartition name. For LOB subpartitions, the generated LOB subpartition name is a concatenation of the partition name and the template LOB segment name. If the `COMPATIBLE` initialization parameter is set to `12.2` or higher, then the maximum length of the concatenation is 128 bytes; otherwise, the maximum length is 30 bytes. If the concatenation exceeds the maximum length, then the database returns an error and the statement fails.
  - If you omit a subpartition name when specifying an individual subpartition, and you have not specified *subpartition_template*, then the database generates a name of the form `SYS_SUBP`*n*. The corresponding system-generated names for LOB data and index subpartitions are `SYS_LOB_SUBP`*n* and `SYS_IL_SUBP`*n*, respectively.
- Tablespace storage can be specified at various levels in the `CREATE TABLE` statement for both table segments and LOB segments. The number of tablespaces does not have to equal the number of partitions or subpartitions. If the number of partitions or subpartitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

  The database evaluates tablespace storage in the following order of descending priority:
  - Tablespace storage specified at the individual table subpartition or LOB subpartition level has the highest priority, followed by storage specified for the partition or LOB in the *subpartition_template*.
  - Tablespace storage specified at the individual table partition or LOB partition level. Storage parameters specified here take precedence over the *subpartition_template*.
  - Tablespace storage specified for the table
  - Default tablespace storage specified for the user
- By default, nested tables are equipartitioned with the partitioned base table.

Restrictions on Partitioning in General

All partitioning is subject to the following restrictions:

- You cannot partition a table that is part of a cluster.
- You cannot partition a nested table or varray that is defined as an index-organized table.

- You cannot partition a table containing any `LONG` or `LONG RAW` columns.

The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Refer to *Oracle Database VLDB and Partitioning Guide* for a discussion of these restrictions.

> **See Also:**
>
> "Partitioning Examples"

range_partitions

Use the *`range_partitions`* clause to partition the table on ranges of values from the column list. For an index-organized table, the column list must be a subset of the primary key columns of the table.

Restrictions on Range Partitioning

Range partitioning is subject to the restrictions listed in "Restrictions on Partitioning in General". The following additional restrictions apply:

- You cannot specify more than 16 partitioning key columns.
- Partitioning key columns must be of type `CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `VARCHAR`, `NUMBER`, `FLOAT`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH LOCAL TIMEZONE`, or `RAW`.
- Each partitioning key column with a character data type must have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`. For all these collations, partition bounds are checked using the collation `BINARY`.
- You cannot specify NULL in the `VALUES` clause.

column

Specify an ordered list of columns used to determine into which partition a row belongs. These columns are the **partitioning key**. You can specify virtual columns and `INVISIBLE` columns as partitioning key columns.

INTERVAL Clause

Use this clause to establish **interval partitioning** for the table. Interval partitions are partitions based on a numeric range or datetime interval. They extend range partitioning by instructing the database to create partitions of the specified range or interval automatically when data inserted into the table exceeds all of the range partitions. For each automatically created partition, the database generates a name of the form `SYS_P`*`n`*. The database guarantees that automatically generated partition names are unique and do not violate namespace rules.

- For *`expr`*, specify a valid number or interval expression.
- The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store interval partition data.
- You must also specify at least one range partition using the `PARTITION` clause of *`range_partitions`*. The range partition key value determines the high value of the range partitions, which is called the **transition point**, and the database creates interval partitions for data beyond that transition point.

Restrictions on Interval Partitioning

The `INTERVAL` clause is subject to the restrictions listed in "Restrictions on Partitioning in General" and "Restrictions on Range Partitioning". The following additional restrictions apply:

- You can specify only one partitioning key column, and it must be of type `NUMBER`, `DATE`, `FLOAT`, or `TIMESTAMP`.
- This clause is not supported for index-organized tables.
- This clause is not supported for tables containing varray columns.
- You cannot create an interval-partitioned table with equipartitioned nested tables. If you create an interval-partitioned table using nested tables or XML object-relational data types, then the nested tables will be created as nonpartitioned tables.
- This clause is supported for tables containing `XMLType` columns only if the XML data is stored as binary XML.
- Interval partitioning is not supported at the subpartition level.
- Serializable transactions do not work with interval partitioning. Trying to insert data into a partition of an interval partitioned table that does not yet have a segment causes an error.
- In the `VALUES` clause:
  - You cannot specify `MAXVALUE` (an infinite upper bound), because doing so would defeat the purpose of the automatic addition of partitions as needed.
  - You cannot specify `NULL` values for the partitioning key column.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on interval partitioning

PARTITION *partition*

If you specify a partition name, then the name *`partition`* must conform to the rules for naming schema objects and their part as described in "Database Object Naming Rules ". If you omit *`partition`*, then the database generates a name as described in "Notes on Partitioning in General".

range_values_clause

Specify the noninclusive upper bound for the current partition. The value list is an ordered list of literal values corresponding to the column list in the *`range_partitions`* clause. You can substitute the keyword `MAXVALUE` for any literal in the value list. `MAXVALUE` specifies a maximum value that will always sort higher than any other value, including null.

Specifying a value other than `MAXVALUE` for the highest partition bound imposes an implicit integrity constraint on the table.

> **Note:**
>
> If *table* is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, then you must use the TO_DATE function with the YYYY 4-character format mask for the year. The RRRR format mask is not supported in this clause. The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT. Refer to *Oracle Database Globalization Support Guide* for more information on these initialization parameters.

> **See Also:**
>
> *Oracle Database Concepts* for more information about partition bounds and "Range Partitioning Example"

table_partition_description

Use the *table_partition_description* to define the physical and storage characteristics of the table.

The clauses *deferred_segment_creation*, *segment_attributes_clause*, *table_compression*, *inmemory_clause*, and *ilm_clause* have the same function as described for the *physical_properties* of the table as a whole.

Use the *indexing_clause* to set the indexing property for a range, list, system, or reference table partition. Refer to the *indexing_clause* for more information.

The *prefix_compression* clause and OVERFLOW clause, have the same function as described for the *index_org_table_clause*.

LOB_storage_clause

The *LOB_storage_clause* lets you specify LOB storage characteristics for one or more LOB items in this partition or in any range or list subpartitions of this partition. If you do not specify the *LOB_storage_clause* for a LOB item, then the database generates a name for each LOB data partition as described in "Notes on Partitioning in General".

varray_col_properties

The *varray_col_properties* let you specify storage characteristics for one or more varray items in this partition or in any range or list subpartitions of this partition.

nested_table_col_properties

The *nested_table_col_properties* let you specify storage characteristics for one or more nested table storage table items in this partition or in any range or list subpartitions of this partition. Storage characteristics specified in this clause override any storage attributes specified at the table level.

partitioning_storage_clause

Use the *partitioning_storage_clause* to specify storage characteristics for hash partitions and for range, hash, and list subpartitions.

Restrictions on *partitioning_storage_clause*

This clause is subject to the following restrictions:

- The `TABLESPACE SET` clause is valid only when creating a sharded table by specifying the `SHARDED` keyword of `CREATE TABLE`. Use this clause to specify the tablespace set in which table partition data is to be stored.
- The `OVERFLOW` clause is relevant only for index-organized partitioned tables and is valid only within the *`individual_hash_partitions`* clause. It is not valid for range or hash partitions or for subpartitions of any type.
- You cannot specify the *`advanced_index_compression`* clause of the *`index_compression`* clause.
- You can specify the *`prefix_compression`* clause of the *`indexing_clause`* only for partitions of index-organized tables and you can specify `COMPRESS` or `NOCOMPRESS`, but you cannot specify the prefix length with *`integer`*.

list_partitions

Use the *`list_partitions`* clause to partition the table on a list of literal values for each column in the *`column`* list. List partitioning is useful for controlling how individual rows map to specific partitions.

Restrictions on List Partitioning

List partitioning is subject to the restrictions listed in "Restrictions on Partitioning in General". The following additional restrictions apply:

- You cannot specify more than 16 partitioning key columns.
- You cannot specify more than one partitioning key column when partitioning an index-organized table.
- The partitioning key columns must be of type `CHAR`, `NCHAR`, `VARCHAR2`, `NVARCHAR2`, `VARCHAR`, `NUMBER`, `FLOAT`, `DATE`, `TIMESTAMP`, `TIMESTAMP WITH LOCAL TIMEZONE`, or `RAW`.
- Each partitioning key column with a character data type must have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`. For all these collations, partitions are selected using the collation `BINARY`.

AUTOMATIC

Specify `AUTOMATIC` to create an automatic list-partitioned table. This type of table enables the database to create additional partitions on demand.

When you create an automatic list-partitioned table, you specify partitions and partitioning key values just as you would when creating a regular list-partitioned table. However, you do not specify a `DEFAULT` partition. As data is loaded into the table, the database automatically creates a new partition when the loaded partitioning key values do not correspond to any of the existing partitions. If list partitioning is defined with a single partitioning key value, then the database creates a new partition for each new partitioning key value. If list partitioning is defined with multiple partitioning key columns, then the database creates a new partition for each new and unique set of partitioning key values. For each automatically created partition, the database generates a name of the form `SYS_P`*`n`*. The database guarantees that automatically generated partition names are unique and do not violate namespace rules.

You can specify the `AUTOMATIC` keyword for list-partitioned tables, and list-range, list-list, list-hash, and list-interval composite-partitioned tables. For composite-partitioned tables, each automatically created list partition will have one subpartition, unless a subpartition template is defined for the table.

If a local partitioned index is defined on an automatic list-partitioned table, then local index partitions will be created when the corresponding table partitions are created.

Restrictions on Automatic List Partitioning

Automatic list partitioning is subject to the restrictions listed in "Restrictions on List Partitioning". The following additional restrictions apply:

- An automatic list-partitioned table must have at least one partition when created. Because new partitions are automatically created for new, and unknown, partitioning key values, an automatic list-partitioned table cannot have a `DEFAULT` partition.
- Automatic list partitioning is not supported for index-organized tables or external tables.
- Automatic list partitioning is not supported for tables containing varray columns.
- You cannot create a local domain index on an automatic list-partitioned table. You can create a global domain index on an automatic list-partitioned table.
- An automatic list-partitioned table cannot be a child table or a parent table for reference partitioning.
- Automatic list partitioning is not supported at the subpartition level.

STORE IN

The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store data for the automatically created list partitions.

> **Note:**
>
> You can change an automatic list-partitioned table to a regular list-partitioned table, and vice versa. You can also change the tablespaces into which the database will store data for automatically created list partitions. See the clause *alter_automatic_partitioning* of `ALTER TABLE` for more information.

list_values_clause

The *`list_values_clause`* of each partition must have at least one value. If the table is partitioned on one key column, then use the upper branch of the *`list_values`* syntax to specify a list of values for that column. In this case, no value, including NULL, can appear in more than one partition. If the table is partitioned on multiple key columns, then use the lower branch of the *`list_values`* syntax to specify a list of value lists. Each value list is enclosed in parentheses and represents a list of values for the key columns. In this case, individual key column values can appear in more than one partition; however, no complete value list can appear in more than one partition. List partitions are not ordered.

If you specify the literal `NULL` for a partition value in the `VALUES` clause, then to access data in that partition in subsequent queries, you must use an `IS NULL` condition in the `WHERE` clause, rather than a comparison condition.

The DEFAULT keyword creates a partition into which the database will insert any row that does not map to another partition. Therefore, you can specify DEFAULT for only one partition, and you cannot specify any other values for that partition. Further, the default partition must be the last partition you define. The use of DEFAULT is similar to the use of MAXVALUE for range partitions.

The string comprising the list of values for each partition can be up to 4K bytes. The total number of values for all partitions cannot exceed 64K-1.

The partitioning key column for a list partition can be an extended data type column, which has a maximum size of 32767 bytes. In this case, the list of values that you want to specify for a partition may exceed the 4K byte limit. You can work around this limitation by using one of the following methods:

- Use the DEFAULT partition for values that exceed the 4K byte limit.
- Use a hash function, such as STANDARD_HASH, in the partition key column to create unique identifiers of lengths less than 4K bytes. See STANDARD_HASH for more information.

Restriction on the *list_values_clause*

You cannot specify a DEFAULT partition for an automatic list-partitioned table.

> **See Also:**
>
> "Extended Data Types" for more information on extended data types

table_partition_description

The subclauses of the *table_partition_description* have the same behavior as described for range partitions in *table_partition_description*.

hash_partitions

Use the *hash_partitions* clause to specify that the table is to be partitioned using the hash method. Oracle Database assigns rows to partitions using a hash function on values found in columns designated as the partitioning key. You can specify individual hash partitions, or you can specify how many hash partitions the database should create.

Restrictions on Hash Partitioning

Hash partitioning is subject to the restrictions listed in "Restrictions on Partitioning in General". The following additional restrictions apply:

- You cannot specify more than 16 partitioning key columns.
- Partitioning key columns must be of type CHAR, NCHAR, VARCHAR2, NVARCHAR2, VARCHAR, NUMBER, FLOAT, DATE, TIMESTAMP, TIMESTAMP WITH LOCAL TIMEZONE, or RAW.

column

Specify an ordered list of columns used to determine into which partition a row belongs (the partitioning key).

individual_hash_partitions

Use this clause to specify individual partitions by name.

Use the `indexing_clause` to set the indexing property for a hash partition. Refer to the *indexing_clause* for more information.

Restriction on Specifying Individual Hash Partitions

The only clauses you can specify in the `partitioning_storage_clause` are the `TABLESPACE` clause and table compression.

> **Note:**
>
> If your enterprise has or will have databases using different character sets, then use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets. Refer to *Oracle Database Globalization Support Guide* for more information on character set support.

hash_partitions_by_quantity

An alternative to defining individual partitions is to specify the number of hash partitions. In this case, the database assigns partition names of the form `SYS_Pn`. The `STORE IN` clause lets you specify one or more tablespaces where the hash partition data is to be stored. The number of tablespaces need not equal the number of partitions. If the number of partitions is greater than the number of tablespaces, then the database cycles through the names of the tablespaces.

For both methods of hash partitioning, for optimal load balancing you should specify a number of partitions that is a power of 2. When you specify individual hash partitions, you can specify both `TABLESPACE` and table compression in the `partitioning_storage_clause`. When you specify hash partitions by quantity, you can specify only `TABLESPACE`. Hash partitions inherit all other attributes from table-level defaults.

The `table_compression` clause has the same function as described for the *table_properties* of the table as a whole.

The `prefix_compression` clause and the `OVERFLOW` clause have the same function as described for the *index_org_table_clause*.

Tablespace storage specified at the table level is overridden by tablespace storage specified at the partition level, which in turn is overridden by tablespace storage specified at the subpartition level.

In the `individual_hash_partitions` clause, the `TABLESPACE` clause of the `partitioning_storage_clause` determines tablespace storage only for the individual partition being created. In the `hash_partitions_by_quantity` clause, the `STORE IN` clause determines placement of partitions as the table is being created and the default storage location for subsequently added partitions.

Restriction on Specifying Hash Partitions by Quantity

You cannot specify the `advanced_index_compression` clause of the `index_compression` clause.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for more information on hash partitioning

composite_range_partitions

Use the `composite_range_partitions` clause to first partition `table` by range, and then partition the partitions further into range, hash, or list subpartitions.

The `INTERVAL` clause has the same semantics for composite range partitioning that it has for range partitioning. Refer to "INTERVAL Clause" for more information.

Specify *subpartition_by_range*, *subpartition_by_hash* or *subpartition_by_list* to indicate the type of subpartitioning you want for each composite range partition. Within these clauses you can specify a subpartition template, which establishes default subpartition characteristics for subpartitions created as part of this statement or subsequently created subpartitions.

After establishing the type of subpartitioning you want for the table, and optionally a subpartition template, you must define at least one range partition.

- You must specify the *range_values_clause* , which has the same requirements as for noncomposite range partitions.
- Use the *table_partition_description* to define the physical and storage characteristics of the each partition.
- In the `range_partition_desc`, use `range_subpartition_desc`, `list_subpartition_desc`, `individual_hash_subparts`, or `hash_subparts_by_quantity` to specify characteristics for the individual subpartitions of the partition. The values you specify in these clauses supersede for these subpartitions any values you have specified in the `subpartition_template`.
- The only characteristics you can specify for a hash or list subpartition or any LOB subpartition are `TABLESPACE` and `table_compression`.

Restrictions on Composite Range Partitioning

Regardless of the type of subpartitioning, composite range partitioning is subject to the following restrictions:

- The only physical attributes you can specify at the subpartition level are `TABLESPACE` and table compression.
- You cannot specify composite partitioning for an index-organized table. Therefore, the `OVERFLOW` clause of the `table_partition_description` is not valid for composite-partitioned tables.
- You cannot specify composite partitioning for tables containing `XMLType` columns.

> **See Also:**
>
> "Composite-Partitioned Table Examples" for examples of composite range partitioning and *Oracle Database VLDB and Partitioning Guide* for examples of composite list partitioning

composite_list_partitions

Use the *composite_list_partitions* clause to first partition *table* by list, and then partition the partitions further into range, hash, or list subpartitions.

Specify *subpartition_by_range*, *subpartition_by_hash* or *subpartition_by_list* to indicate the type of subpartitioning you want for each composite list partition. Within these clauses you can specify a subpartition template, which establishes default subpartition characteristics for subpartitions created as part of this statement and for subsequently created subpartitions.

After establishing the type of subpartitioning you want for each composite partition, and optionally defining a subpartition template, you must define at least one list partition.

- In the *list_partition_desc*, you must specify the *list_values_clause*, which has the same requirements as for noncomposite list partitions.
- Use the *table_partition_description* to define the physical and storage characteristics of the each partition.
- In the *list_partition_desc*, use *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* to specify characteristics for the individual subpartitions of the partition. The values you specify in these clauses supersede the for these subpartitions any values you have specified in the *subpartition_template*.

Specify `AUTOMATIC` to create an automatic list-range, list-list, list-hash, or list-interval composite-partitioned table. This type of table enables the database to create additional partitions on demand. The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store data for the automatically created partitions. The `AUTOMATIC` and `STORE IN` clauses have the same semantics here as they have for noncomposite list partitions. Refer to AUTOMATIC and STORE IN in the documentation on *list_partitions* for the full semantics of these clauses. Automatic composite-partitioned tables are subject to the restrictions listed in Restrictions on Composite List Partitioning and Restrictions on Automatic List Partitioning.

Restrictions on Composite List Partitioning

Composite list partitioning is subject to the same restrictions as described in "Restrictions on Composite Range Partitioning".

composite_hash_partitions

Use the *composite_hash_partitions* clause to first partition *table* using the hash method, and then partition the partitions further into range, hash, or list subpartitions.

Specify *subpartition_by_range*, *subpartition_by_hash* or *subpartition_by_list* to indicate the type of subpartitioning you want for each composite range partition. Within these clauses you can specify a subpartition template, which establishes default subpartition characteristics for subpartitions created as part of this statement or subsequently created subpartitions.

After establishing the type of subpartitioning you want for the table, you must specify *individual_hash_partitions* or *hash_partitions_by_quantity*.

Restrictions on Composite Hash Partitioning

Composite hash partitioning is subject to the same restrictions as described in "Restrictions on Composite Range Partitioning".

subpartition_template

The *subpartition_template* is an optional element of range, list, and hash subpartitioning. The template lets you define default subpartitions for each table partition. Oracle Database will create these default subpartition characteristics in any partition for which you do not explicitly define subpartitions. This clause is useful for creating symmetric partitions. You can override this clause by explicitly defining subpartitions at the partition level, in the *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* clause.

When defining subpartitions with a template, you can explicitly define range, list, or hash subpartitions, or you can define a quantity of hash subpartitions.

- To explicitly define subpartitions, use *range_subpartition_desc*, *list_subpartition_desc*, or *individual_hash_subparts*. You must specify a name for each subpartition. If you specify the *LOB_partitioning_clause* of the *partitioning_storage_clause*, then you must specify *LOB_segname*.
- To define a quantity of hash subpartitions, specify a positive integer for *hash_subpartition_quantity*. The database creates that number of subpartitions in each partition and assigns subpartition names of the form `SYS_SUBP`*n*.

> **Note:**
>
> When you specify tablespace storage for the subpartition template, it does not override any tablespace storage you have specified explicitly for the partitions of *table*. To specify tablespace storage for subpartitions, do one of these things:
>
> - Omit tablespace storage at the partition level and specify tablespace storage in the subpartition template.
> - Define individual subpartitions with specific tablespace storage.

Restrictions on Subpartition Templates

Subpartition templates are subject to the following restrictions:

- If you specify `TABLESPACE` for one LOB subpartition, then you must specify `TABLESPACE` for all of the LOB subpartitions of that LOB column. You can specify the same tablespace for more than one LOB subpartition.
- If you specify separate LOB storage for list subpartitions using the *partitioning_storage_clause*, either in the *subpartition_template* or when defining individual subpartitions, then you must specify *LOB_segname* for both LOB and varray columns.

subpartition_by_range

Use the *subpartition_by_range* clause to indicate that the database should subpartition by range each partition in *table*. The subpartitioning column list is unrelated to the partitioning key but is subject to the same restrictions (see column).

You can use the *subpartition_template* to specify default subpartition characteristic values. See *subpartition_template*. The database uses these values for any subpartition in this partition for which you do not explicitly specify the characteristic.

You can also define range subpartitions individually for each partition using the *range_subpartition_desc* of *range_partition_desc* or *list_partition_desc*. If you omit both *subpartition_template* and the *range_subpartition_desc*, then the database creates a single MAXVALUE subpartition.

subpartition_by_list

Use the *subpartition_by_list* clause to indicate that the database should subpartition each partition in the table on lists of literal values from the *column* list. You can specify a maximum of 16 list subpartitioning key columns.

You can use the *subpartition_template* to specify default subpartition characteristic values. See *subpartition_template*. The database uses these values for any subpartition in this partition for which you do not explicitly specify the characteristic.

You can also define list subpartitions individually for each partition using the *list_subpartition_desc* of *range_partition_desc* or *list_partition_desc*. If you omit both *subpartition_template* and the *list_subpartition_desc*, then the database creates a single DEFAULT subpartition.

Restrictions on List Subpartitioning

List subpartitioning is subject to the same restrictions as described in Restrictions on Composite Range Partitioning.

subpartition_by_hash

Use the *subpartition_by_hash* clause to indicate that the database should subpartition by hash each partition in *table*. The subpartitioning column list is unrelated to the partitioning key but is subject to the same restrictions (see column).

You can define the subpartitions using the *subpartition_template* or the SUBPARTITIONS *integer* clause. See *subpartition_template*. In either case, for optimal load balancing you should specify a number of partitions that is a power of 2.

If you specify SUBPARTITIONS *integer*, then you determine the default number of subpartitions in each partition of *table*, and optionally one or more tablespaces in which they are to be stored. The default value is 1. If you omit both this clause and *subpartition_template*, then the database will create each partition with one hash subpartition.

Notes on Composite Partitions

The following notes apply to composite partitions:

- For all subpartitions, you can use the *range_subpartition_desc*, *list_subpartition_desc*, *individual_hash_subparts*, or *hash_subparts_by_quantity* to specify individual subpartitions by name, and optionally some other characteristics.
- Alternatively, for hash and list subpartitions:

   - You can specify the number of subpartitions and optionally one or more tablespaces where they are to be stored. In this case, Oracle Database assigns subpartition names of the form `SYS_SUBPn`.
   - If you omit the subpartition description and if you have created a subpartition template, then the database uses the template to create subpartitions. If you have not created a subpartition template, then the database creates one hash subpartition or one `DEFAULT` list subpartition.
- For all types of subpartitions, if you omit the subpartitions description entirely, then the database assigns subpartition names as follows:
   - If you have specified a subpartition template *and* you have specified partition names, then the database generates subpartition names of the form *`partition_name`* underscore (_) *`subpartition_name`* (for example, `P1_SUB1`).
   - If you have not specified a subpartition template *or* if you have specified a subpartition template but did not specify partition names, then the database generates subpartition names of the form `SYS_SUBP`*`n`*.

reference_partitioning

Use this clause to partition the table by reference. Partitioning by reference is a method of equipartitioning the table being created (the **child table**) by a referential constraint to an existing partitioned table (the **parent table**). When you partition a table by reference, partition maintenance operations subsequently performed on the parent table automatically cascade to the child table. Therefore, you cannot perform partition maintenance operations on a reference-partitioned table directly.

If the parent table is an interval-partitioned table, then partitions in the reference-partitioned child table that correspond to interval partitions in the parent table will be created during inserts into the child table. When an interval partition in a child table is created, the partition name is inherited from the associated parent table partition. If the child table has a table-level default tablespace, then it will be used as the tablespace for the new interval partition. Otherwise, the tablespace will be inherited from the parent table partition. Refer to *Oracle Database VLDB and Partitioning Guide* for more information on referencing an interval-partitioned table.

constraint

The partitioning referential constraint must meet the following conditions:

- You must specify a referential integrity constraint defined on the table being created, which must refer to a primary key or unique constraint on the parent table. The constraint must be in `ENABLE VALIDATE NOT DEFERRABLE` state, which is the default when you specify a referential integrity constraint during table creation.
- All foreign key columns referenced in the constraint must be `NOT NULL`.
- When you specify the constraint, you cannot specify the `ON DELETE SET NULL` clause of the *`references_clause`*.
- The parent table referenced in the constraint must be an existing partitioned table. It can be partitioned by any method.
- The foreign and parent keys cannot contain any virtual columns that reference PL/SQL functions or LOB columns.

reference_partition_desc

Use this optional clause to specify partition names and to define the physical and storage characteristics of the partition. The subclauses of the `table_partition_description` have the same behavior as described for range partitions in *table_partition_description*.

If you specify this clause when creating a reference-partitioned child table whose parent is an interval-partitioned table, then the partition descriptors are used for the child table's non-interval partitions. Partition descriptors cannot be specified for interval partitions.

Restrictions on Reference Partitioning

Reference partitioning is subject to the restrictions listed in Restrictions on Partitioning in General. The following additional restrictions apply:

- Restrictions for reference partitioning are derived from the partitioning strategy of the parent table.
- Neither the parent table nor the child table can be an automatic list-partitioned table.
- You cannot specify this clause for an index-organized table, an external table, or a domain index storage table.
- The parent table can be partitioned by reference, but `constraint` cannot be self-referential. The table being created cannot be partitioned based on a reference to itself.
- If `ROW MOVEMENT` is enabled for the parent table, it must also be enabled for the child table.
- In both the child table and parent table, the character columns involved in the referential integrity constraint must each have one of the following declared collations: `BINARY`, `USING_NLS_COMP`, `USING_NLS_SORT`, or `USING_NLS_SORT_CS`.

See Also:

Oracle Database VLDB and Partitioning Guide for more information on partitioning by reference and "Reference Partitioning Example"

system_partitioning

Use this clause to create system partitions. System partitioning does not entail any partitioning key columns, nor do system partitions have any range or list bounds or hash algorithms. Rather, they provide a way to equipartition dependent tables such as nested table or domain index storage tables with partitioned base tables.

- If you specify only `PARTITION BY SYSTEM`, then the database creates one partition with a system-generated name of the form `SYS_Pn`.
- If you specify `PARTITION BY SYSTEM PARTITIONS integer`, then the database creates as many partitions as you specify in `integer`, which can range from 1 to 1024K-1.
- The description of the partition takes the same syntax as reference partitions, so they share the `reference_partition_desc`. You can specify additional partition

attributes with the *reference_partition_desc* syntax. However, within the *table_partition_description*, you cannot specify the OVERFLOW clause.

Restrictions on System Partitioning

System partitioning is subject to the following restrictions:

- You cannot system partition an index-organized table or a table that is part of a cluster.
- Composite partitioning is not supported with system partitioning.
- You cannot split a system partition.
- You cannot specify system partitioning in a CREATE TABLE ... AS SELECT statement.
- To insert data into a system-partitioned table using an INSERT INTO ... AS *subquery* statement, you must use partition-extended syntax to specify the partition into which the values returned by the subquery will be inserted.

> **See Also:**
>
> Refer to *Oracle Database Data Cartridge Developer's Guide* for information on the uses for system partitioning and "References to Partitioned Tables and Indexes "

consistent_hash_partitions

This clause is valid only for sharded tables. Use this clause to create consistent hash partitions.

consistent_hash_with_subpartitions

This clause is valid only for sharded tables. Use this clause to create consistent hash with subpartitions.

range_partitionset_clause

Use this clause to create a range partition set.

In the SUBPARTITION BY clause, within the *subpartition_template* clause, you cannot specify a tablespace for a subpartition. That is, for range, list, and individual hash subpartitions, you cannot specify the TABLESPACE clause of the *partitioning_storage_clause*, and in the *hash_subpartitions_by_quantity* clause, you cannot specify the STORE IN (*tablespace*) clause.

In the PARTITIONS AUTO clause, within the *subpartition_template* clause of the *range_partitionset_desc* clause, you *can* specify a tablespace for a subpartition.

list_partitionset_clause

Use this clause to create a list partition set.

In the SUBPARTITION BY clause, within the *subpartition_template* clause, you cannot specify a tablespace for a subpartition. That is, for range, list, and individual hash subpartitions, you cannot specify the TABLESPACE clause of the *partitioning_storage_clause*, and in the *hash_subpartitions_by_quantity* clause, you cannot specify the STORE IN (*tablespace*) clause.

In the PARTITIONS AUTO clause, within the *subpartition_template* clause of the *list_partitionset_desc* clause, you *can* specify a tablespace for a subpartition.

attribute_clustering_clause

Use this clause to enable the table for attribute clustering. Attribute clustering lets you cluster data in close physical proximity based on the content of specified columns.

Attribute clustering can be based only on columns in *table* or on joined values from other tables. The latter is called join attribute clustering.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on attribute clustering

clustering_join

Use this clause to specify join attribute clustering. Use the JOIN clause to specify the joined values from other tables on which to base the attribute clustering. You can specify a maximum of four JOIN clauses.

cluster_clause

Use this clause to specify the type of ordering to use for the table: linear ordering or interleaved ordering. If you do not specify the LINEAR or INTERLEAVED keyword, then the default is LINEAR.

BY LINEAR ORDER

Use this clause to specify linear ordering. This type of ordering stores data according to the order of the specified columns. If you specify this clause, then you can specify only one clustering column group, which can contain at most 10 columns.

BY INTERLEAVED ORDER

Use this clause to specify interleaved ordering. This type of ordering uses a special multidimensional clustering technique, similar to z-ordering, that permits multicolumn clustering. If you specify this clause, then you can specify at most four clustering column groups, with a maximum of 40 columns across all groups.

clustering_columns

Use this clause to specify one or more clustering column groups.

clustering_column_group

Use this clause to specify one or more columns to be included in the clustering column group.

Restriction on Attribute Clustering Columns

Each character column in the clustering column group must have one of the following declared collations: BINARY or USING_NLS_COMP.

clustering_when

Use these clauses to allow or disallow attribute clustering during direct-path insert operations and data movement operations.

ON LOAD

Specify `YES ON LOAD` to allow, or `NO ON LOAD` to disallow, attribute clustering during direct-path inserts (serial or parallel) resulting either from an `INSERT` or a `MERGE` operation. The default is `YES ON LOAD`.

ON DATA MOVEMENT

Specify `YES ON DATA MOVEMENT` to allow, or `NO ON DATA MOVEMENT` to disallow, attribute clustering for data movement that occurs during the following operations:

- Data redefinition using the `DBMS_REDEFINITION` package
- Table partition maintenance operations that are specified by the following clauses of `ALTER TABLE`: *`coalesce_table`*, *`merge_table_partitions`*, *`move_table_partition`*, and *`split_table_partition`*

The default is `YES ON DATA MOVEMENT`.

zonemap_clause

Use this clause to create a zone map on the table. The zone map tracks the columns specified in the *`clustering_columns`* clause.

- Specify `WITH MATERIALIZED ZONEMAP` to create a zone map. For *`zonemap_name`*, specify the name of the zone map to be created. If you omit *`zonemap_name`*, then the name of the zone map is `ZMAP$_`*`table`*.
- Specify `WITHOUT MATERIALIZED ZONEMAP` to not create a zone map. This is the default.

If you subsequently drop the table or use the `ALTER TABLE` statement to `DROP CLUSTERING` or `MODIFY CLUSTERING` ... `WITHOUT MATERIALIZED ZONEMAP`, then the zone map will be dropped.

> **See Also:**
>
> CREATE MATERIALIZED ZONEMAP for more information on zone maps

Restrictions on Attribute Clustering

The following restrictions apply to attribute clustering:

- Attribute clustering is not supported for temporary tables or external tables.
- The table being created must be a heap-organized table. However, tables specified in the *`clustering_join`* clause can be heap-organized or index-organized tables.
- Clustering columns must be of a scalar data type and cannot be encrypted.
- If you specify `BY LINEAR ORDER`, then you can specify only one clustering column group, which can contain at most 10 columns.
- If you specify `BY INTERLEAVED ORDER`, then you can specify at most four clustering column groups, with a maximum of 40 columns across all groups.
- For join attribute clustering:
  – The number of dimension tables cannot exceed four.

  - The join to the table or tables providing the attribute clustering columns must be on a unique key or primary key column to avoid row duplication.
- Attribute clustering will not order rows that are inserted using `MERGE` statements or multitable insert operations.

CACHE | NOCACHE | CACHE READS

Use these clauses to indicate how Oracle Database should store blocks in the buffer cache. For LOB storage, you can specify `CACHE`, `NOCACHE`, or `CACHE READS`. For other types of storage, you can specify only `CACHE` or `NOCACHE`.

If you omit these clauses, then:

- In a `CREATE TABLE` statement, `NOCACHE` is the default.
- In an `ALTER TABLE` statement, the existing value is not changed.

The behavior of `CACHE` and `NOCACHE` described in this section does not apply when Oracle Database chooses to use direct reads or to perform table scans using parallel query.

CACHE

For data that is accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the most recently used end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

> **See Also:**
>
> *Oracle Database Concepts* for more information on how the database maintains the least recently used (LRU) list

As a parameter in the *`LOB_storage_clause`*, `CACHE` specifies that the database places LOB data values in the buffer cache for faster access. The database evaluates this parameter in conjunction with the *`logging_clause`*. If you omit this clause, then the default value for both BasicFiles and SecureFiles LOBs is `NOCACHE LOGGING`.

Restriction on CACHE

You cannot specify `CACHE` for an index-organized table. However, index-organized tables implicitly provide `CACHE` behavior.

NOCACHE

For data that is not accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. `NOCACHE` is the default for LOB storage.

As a parameter in the *`LOB_storage_clause`*, `NOCACHE` specifies that the LOB values are not brought into the buffer cache. `NOCACHE` is the default for LOB storage.

Restriction on NOCACHE

You cannot specify `NOCACHE` for an index-organized table.

CACHE READS

`CACHE READS` applies only to LOB storage. It specifies that LOB values are brought into the buffer cache only during read operations but not during write operations.

logging_clause

Use this clause to indicate whether the storage of data blocks should be logged or not.

> **See Also:**
>
> *logging_clause* for a description of the *`logging_clause`* when specified as part of *`LOB_parameters`*

RESULT_CACHE Clause

Use this clause to determine whether the results of statements or query blocks that name this table are considered for storage in the result cache. Two modes of result caching are available:

- `DEFAULT`: Result caching is not determined at the table level. The query is considered for result caching if the `RESULT_CACHE_MODE` initialization parameter is set to `FORCE`, or if that parameter is set to `MANUAL` and the `RESULT_CACHE` hint is specified in the query. This is the default if you omit this clause.
- `FORCE`: If all tables names in the query have this setting, then the query is always considered for caching unless the `NO_RESULT_CACHE` hint is specified for the query. If one or more tables named in the query are set to `DEFAULT`, then the effective table annotation for that query is considered to be `DEFAULT`, with the semantics described above.

You can query the `RESULT_CACHE` column of the `DBA_`, `ALL_`, and `USER_TABLES` data dictionary views to learn the result cache mode of the table.

The `RESULT_CACHE` and `NO_RESULT_CACHE` SQL hints take precedence over these result cache table annotations and the `RESULT_CACHE_MODE` initialization parameter. The `RESULT_CACHE_MODE` setting of `FORCE` in turn takes precedence over this table annotation clause.

> **Note:**
>
> The `RESULT_CACHE_MODE` setting of `FORCE` is not recommended, as it can cause significant performance and latching overhead, as database and clients will try to cache all queries.

See Also:

- *Oracle Call Interface Programmer's Guide* and *Oracle Database Concepts* for general information about result caching
- *Oracle Database Performance Tuning Guide* for information about using this clause
- *Oracle Database Reference* for information about the `RESULT_CACHE_MODE` initialization parameter and the `*_TABLES` data dictionary views
- "RESULT_CACHE Hint " and "NO_RESULT_CACHE Hint " for information about the hints

parallel_clause

The *`parallel_clause`* lets you parallelize creation of the table and set the default degree of parallelism for queries and the DML `INSERT`, `UPDATE`, `DELETE`, and `MERGE` after table creation.

Note:

The syntax of the *`parallel_clause`* supersedes syntax appearing in earlier releases of Oracle. The superseded syntax is still supported for backward compatibility, but may result in slightly different behavior from that documented.

NOPARALLEL

Specify `NOPARALLEL` for serial execution. This is the default.

PARALLEL

Specify `PARALLEL` if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.

PARALLEL *integer*

Specification of *`integer`* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *`integer`*.

See Also:

parallel_clause for more information on this clause

NOROWDEPENDENCIES | ROWDEPENDENCIES

This clause lets you specify whether *table* will use **row-level dependency tracking**. With this feature, each row in the table has a system change number (SCN) that represents a time greater than or equal to the commit time of the last transaction that modified the row. You cannot change this setting after *table* is created.

ROWDEPENDENCIES

Specify `ROWDEPENDENCIES` if you want to enable row-level dependency tracking. This setting is useful primarily to allow for parallel propagation in replication environments. It increases the size of each row by 6 bytes.

Restriction on the ROWDEPENDENCIES Clause

Oracle does not support table compression for tables that use row-level dependency tracking. If you specify both the `ROWDEPENDENCIES` clause and the *table_compression* clause, then the *table_compression* clause is ignored. To remove the `ROWDEPENDENCIES` attribute, you must redefine the table using the `DBMS_REDEFINITION` package or recreate the table.

NOROWDEPENDENCIES

Specify `NOROWDEPENDENCIES` if you do not want *table* to use the row-level dependency tracking feature. This is the default.

enable_disable_clause

The *enable_disable_clause* lets you specify whether Oracle Database should apply a constraint. By default, constraints are created in `ENABLE VALIDATE` state.

Restrictions on Enabling and Disabling Constraints

Enabling and disabling constraints are subject to the following restrictions:

- To enable or disable any integrity constraint, you must have defined the constraint in this or a previous statement.
- You cannot enable a foreign key constraint unless the referenced unique or primary key constraint is already enabled.
- In the *index_properties* clause of the *using_index_clause*, the `INDEXTYPE IS` ... clause is not valid in the definition of a constraint.

ENABLE Clause

Use this clause if you want the constraint to be applied to the data in the table.

DISABLE Clause

Use this clause if you want to disable the integrity constraint.

> **See Also:**
>
> *constraint* for full descriptions of **ENABLE Clause** and **DISABLE Clause** and "Creating a Table: ENABLE/DISABLE Examples"

Use this clause if you want to disable the integrity constraint. This clause is described fully in in the documentation on constraints.

UNIQUE

The `UNIQUE` clause lets you enable or disable the unique constraint defined on the specified column or combination of columns.

PRIMARY KEY

The `PRIMARY KEY` clause lets you enable or disable the primary key constraint defined on the table.

CONSTRAINT

The `CONSTRAINT` clause lets you enable or disable the integrity constraint named *`constraint_name`*.

KEEP | DROP INDEX

This clause lets you either preserve or drop the index Oracle Database has been using to enforce a unique or primary key constraint.

Restriction on Preserving and Dropping Indexes

You can specify this clause only when disabling a unique or primary key constraint.

using_index_clause

The *`using_index_clause`* lets you specify an index for Oracle Database to use to enforce a unique or primary key constraint, or lets you instruct the database to create the index used to enforce the constraint.

> **See Also:**
>
> - CREATE INDEX for a description of *index_attributes*, the *global_partitioned_index* and *local_partitioned_index* clauses, `NOSORT`, and the *`logging_clause`* in relation to indexes
> - *constraint* for information on the *`using_index_clause`* and on `PRIMARY KEY` and `UNIQUE` constraints
> - "Explicit Index Control Example" for an example of using an index to enforce a constraint

CASCADE

Specify `CASCADE` to disable any integrity constraints that depend on the specified integrity constraint. To disable a primary or unique key that is part of a referential integrity constraint, you must specify this clause.

Restriction on CASCADE

You can specify `CASCADE` only if you have specified `DISABLE`.

row_movement_clause

The *row_movement_clause* lets you specify whether the database can move a table row. It is possible for a row to move, for example, during table compression or an update operation on partitioned data.

> **Note:**
>
> If you need static rowids for data access, then do not enable row movement. For a normal (heap-organized) table, moving a row changes the rowid of the row. For a moved row in an index-organized table, the logical rowid remains valid, although the physical guess component of the logical rowid becomes inaccurate.

- Specify ENABLE to allow the database to move a row, thus changing the rowid.
- Specify DISABLE if you want to prevent the database from moving a row, thus preventing a change of rowid.

If you omit this clause, then the database disables row movement.

Restriction on Row Movement

You cannot specify this clause for a nonpartitioned index-organized table.

flashback_archive_clause

You must have the FLASHBACK ARCHIVE object privilege on the specified flashback data archive to specify this clause. Use this clause to enable or disable historical tracking for the table.

- Specify FLASHBACK ARCHIVE to enable tracking for the table. You can specify *flashback_archive* to designate a particular flashback data archive for this table. The flashback data archive you specify much already exist.

  If you omit *flashback_archive*, then the database uses the default flashback data archive designated for the system. If no default flashback data archive has been designated for the system, then you must specify *flashback_archive*.
- Specify NO FLASHBACK ARCHIVE to disable tracking for the table. This is the default.

Restrictions on *flashback_archive_clause*

Flashback data archives are subject to the following restrictions:

- You cannot specify this clause for a nested table, clustered table, temporary table, remote table, or external table.
- You cannot specify this clause for a table compressed with Hybrid Columnar Compression.
- The table for which you are specifying this clause cannot contain any LONG or nested table columns.
- If you specify this clause and subsequently copy the table to a different database—using the export and import utilities or the transportable tablespace feature—then the copied table will not be enabled for tracking and the archived data for the original table will not be available for the copied table.

> **See Also:**
>
> - *Oracle Database Development Guide* for general information on using flashback data archives
> - ALTER FLASHBACK ARCHIVE for information on changing the quota and retention attributes of the flashback data archive, as well as adding or changing tablespace storage for the flashback data archive

ROW ARCHIVAL

Specify this clause to enable `table` for row archival. This clause lets you implement In-Database Archiving, which allows you to designate table rows as active or archived. You can then perform queries on only the active rows within the table.

When you specify this clause, a hidden column `ORA_ARCHIVE_STATE` is created in the table. The column is of data type `VARCHAR2`. You can specify a value of `0` or `1` for this column to indicate whether a row is active (`0`) or archived (`1`). If you do not specify a value for `ORA_ARCHIVE_STATE` when inserting data into the table, then the value is set to `0`.

- If `ROW ARCHIVE VISIBILITY = ACTIVE` for the session, then the database will consider only active rows when performing queries on the table.
- If `ROW ARCHIVE VISIBILITY = ALL` for the session, then the database will consider all rows when performing queries on the table.

> **See Also:**
>
> - The `ALTER SESSION` Semantics clause to learn how to configure row archival visibility for a session
> - The `ALTER TABLE` [NO] ROW ARCHIVAL clause to learn how to enable or disable an existing table for row archival
> - *Oracle Database VLDB and Partitioning Guide* for more information on In-Database Archiving

FOR EXCHANGE WITH TABLE

This clause lets you create a table that matches the structure of an existing partitioned table. The two tables are then eligible for exchanging partitions and subpartitions. For *`table`*, specify an existing partitioned table. For *`schema`*, specify the schema that contains the existing partitioned table. If you omit *`schema`*, then the database assumes the table is in your own schema.

This operation creates a metadata clone, without data, of the partitioned table. The clone has the same column ordering and column properties of the original table. Column properties copied to the clone during this operation include unusable columns, invisible columns, virtual expression columns, functional index expression columns, and other internal settings and attributes. Indexes on the existing partitioned table are not created on the clone table.

You can subsequently use the *exchange_partition_subpart* clause of ALTER TABLE to exchange partitions or subpartitions between the two tables. Refer to *exchange_partition_subpart* in the documentation on ALTER TABLE for more information.

Restrictions on FOR EXCHANGE WITH TABLE

The following restrictions apply to the FOR EXCHANGE WITH TABLE clause:

- If you specify this clause, then you cannot specify the *relational_properties* clause.
- If you specify this clause, then within the *table_properties* clause, you can specify only the *table_partitioning_clause*.
- When you create a clone for a partition of a composite-partitioned table, you must explicitly specifying the appropriate *table_partitioning_clause* that matches exactly the subpartitioning of the partition you want to exchange.
- You cannot create a clone of a partitioned index-organized table.
- Oracle does not clone the statistics setup of the partitioned table. For example, if you plan to perform an exchange with a partitioned table for which incremental statistics are enabled, you must manually enable the creation of a table synopsis on the clone table. See *Oracle Database SQL Tuning Guide* for more information on maintaining incremental statistics on partitioned tables.

AS *subquery*

Specify a subquery to determine the contents of the table. The rows returned by the subquery are inserted into the table upon its creation.

For object tables, *subquery* can contain either one expression corresponding to the table type, or the number of top-level attributes of the table type. Refer to SELECT for more information.

If *subquery* returns the equivalent of part or all of an existing materialized view, then the database may rewrite the query to use the materialized view in place of one or more tables specified in *subquery*.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on materialized views and query rewrite

Oracle Database derives data types and lengths from the subquery. Oracle Database follows the following rules for integrity constraints and other column and table attributes:

- Oracle Database automatically defines on columns in the new table any NOT NULL constraints that have a state of NOT DEFERRABLE and VALIDATE, and were explicitly created on the corresponding columns of the selected table if the subquery selects the column rather than an expression containing the column. If any rows violate the constraint, then the database does not create the table and returns an error.
- NOT NULL constraints that were implicitly created by Oracle Database on columns of the selected table (for example, for primary keys) are not carried over to the new table.
- In addition, primary keys, unique keys, foreign keys, check constraints, partitioning criteria, indexes, and column default values are not carried over to the new table.

- If the selected table is partitioned, then you can choose whether the new table will be partitioned the same way, partitioned differently, or not partitioned. Partitioning is not carried over to the new table. Specify any desired partitioning as part of the `CREATE TABLE` statement before the `AS` *`subquery`* clause.
- A column that is encrypted using Transparent Data Encryption in the selected table will not be encrypted in the new table unless you define the column in the new table as encrypted at create time.

> **Note:**
>
> Oracle recommends that you encrypt sensitive columns before populating them with data. This will avoid creating clear text copies of sensitive data.

If each column returned by *`subquery`* has a column name or is an expression with a specified column alias, then you can omit the columns from the table definition entirely. In this case, the names of the columns of *`table`* are the same as the columns in *`subquery`*. The exception is creating an index-organized table, for which you must specify the columns in the table definition because you must specify a primary key column.

You can use *`subquery`* in combination with the `TO_LOB` function to convert the values in a `LONG` column in another table to LOB values in a column of the table you are creating.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for a discussion of why and when to copy `LONG` data to a LOB
> - "Conversion Functions " for a description of how to use the `TO_LOB` function
> - SELECT for more information on the *`order_by_clause`*
> - *Oracle Database SQL Tuning Guide* for information on statistics gathering when using the `AS` *`subquery`* clause

parallel_clause

If you specify the *`parallel_clause`* in this statement, then the database will ignore any value you specify for the `INITIAL` storage parameter and will instead use the value of the `NEXT` parameter.

> **See Also:**
>
> storage_clause for information on these parameters

ORDER BY

The `ORDER BY` clause lets you order rows returned by the subquery.

When specified with `CREATE TABLE`, this clause does not necessarily order data across the entire table. For example, it does not order across partitions. Specify this clause if you intend to create an index on the same key as the `ORDER BY` key column. Oracle Database will cluster data on the `ORDER BY` key so that it corresponds to the index key.

Restrictions on the Defining Query of a Table

The table query is subject to the following restrictions:

- The number of columns in the table must equal the number of expressions in the subquery.
- The column definitions can specify only column names, default values, and integrity constraints, not data types.
- You cannot define a foreign key constraint in a `CREATE TABLE` statement that contains `AS` *`subquery`* unless the table is reference partitioned and the constraint is the table's partitioning referential constraint. In all other cases, you must create the table without the constraint and then add it later with an `ALTER TABLE` statement.

object_table

The `OF` clause lets you explicitly create an **object table** of type *`object_type`*. The columns of an object table correspond to the top-level attributes of type *`object_type`*. Each row will contain an object instance, and each instance will be assigned a unique, system-generated object identifier when a row is inserted. If you omit *`schema`*, then the database creates the object table in your own schema.

Object tables, as well as `XMLType` tables, object views, and `XMLType` views, do not have any column names specified for them. Therefore, Oracle defines a system-generated pseudocolumn `OBJECT_ID`. You can use this column name in queries and to create object views with the `WITH OBJECT IDENTIFIER` clause.

> **See Also:**
>
> "Object Column and Table Examples"

object_table_substitution

Use the *`object_table_substitution`* clause to specify whether row objects corresponding to subtypes can be inserted into this object table.

NOT SUBSTITUTABLE AT ALL LEVELS

`NOT SUBSTITUTABLE AT ALL LEVELS` indicates that the object table being created is not substitutable. In addition, substitution is disabled for all embedded object attributes and elements of embedded nested tables and arrays. The default is `SUBSTITUTABLE AT ALL LEVELS`.

See Also:

- CREATE TYPE for more information about creating object types
- "User-Defined Types ", "About User-Defined Functions ", "About SQL Expressions ", CREATE TYPE , and *Oracle Database Object-Relational Developer's Guide* for more information about using `REF` types

object_properties

The properties of object tables are essentially the same as those of relational tables. However, instead of specifying columns, you specify attributes of the object.

For *`attribute`*, specify the qualified column name of an item in an object.

oid_clause

The *`oid_clause`* lets you specify whether the object identifier of the object table should be system generated or should be based on the primary key of the table. The default is `SYSTEM GENERATED`.

Restrictions on the *oid_clause*

This clause is subject to the following restrictions:

- You cannot specify `OBJECT IDENTIFIER IS PRIMARY KEY` unless you have already specified a `PRIMARY KEY` constraint for the table.
- You cannot specify this clause for a nested table.

Note:

A primary key object identifier is locally unique but not necessarily globally unique. If you require a globally unique identifier, then you must ensure that the primary key is globally unique.

oid_index_clause

This clause is relevant only if you have specified the *`oid_clause`* as `SYSTEM GENERATED`. It specifies an index, and optionally its storage characteristics, on the hidden object identifier column.

For *`index`*, specify the name of the index on the hidden system-generated object identifier column. If you omit *`index`*, then the database generates a name.

physical_properties* and *table_properties

The semantics of these clauses are documented in the corresponding sections under relational tables. See *physical_properties* and *table_properties*.

XMLType_table

Use the *XMLType_table* syntax to create a table of data type XMLType. Most of the clauses used to create an XMLType table have the same semantics that exist for object tables. The clauses specific to XMLType tables are described in this section.

Object tables, as well as XMLType tables, object views, and XMLType views, do not have any column names specified for them. Therefore, Oracle defines a system-generated pseudocolumn OBJECT_ID. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

XMLSchema_spec

This clause lets you specify the URL of a registered XMLSchema, either in the XMLSCHEMA clause or as part of the ELEMENT clause, and an XML element name.

You must specify an element, although the XMLSchema URL is optional. If you do specify an XMLSchema URL, then you must already have registered the XMLSchema using the DBMS_XMLSCHEMA package.

The optional STORE ALL VARRAYS AS clause lets you specify how all varrays in the XMLType table or column are to be stored.

- STORE ALL VARRAYS AS LOBS indicates that all varrays are to be stored as LOBs.
- STORE ALL VARRAYS AS TABLES indicates that all varrays are to be stored as tables.

The optional ALLOW | DISALLOW clauses are valid only if you have specified BINARY XML storage.

- ALLOW NONSCHEMA indicates that non-schema-based documents can be stored in the XMLType column.
- DISALLOW NONSCHEMA indicates that non-schema-based documents cannot be stored in the XMLType column. This is the default.
- ALLOW ANYSCHEMA indicates that any schema-based document can be stored in the XMLType column.
- DISALLOW ANYSCHEMA indicates that any schema-based document cannot be stored in the XMLType column. This is the default.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information on the DBMS_XMLSCHEMA package
- *Oracle XML DB Developer's Guide* for information on creating and working with XML data
- "XMLType Table Examples"

PARENT

You can use this clause to create a child table in a sharded table family.

A sharded table family is a set of tables that are sharded in the same way. Corresponding partitions of all tables in a table family are stored in the same shard. This enables you to minimize the number of multishard joins when querying data in the table family.

There are two methods for creating a sharded table family. The recommended method involves using reference partitioning. However, if it is impossible or undesirable to create the primary and foreign key constraints that are required for reference partitioning, then you can use the `PARENT` clause to create a sharded table family.

The rules for creating a sharded table family differ depending on which method you use. When you create a sharded table family by using the `PARENT` clause, the following rules apply:

- The sharded table family can contain only two levels of tables: a parent table, and one or more child tables.
- All tables in the family must be explicitly partitioned using the same partitioning scheme. Each table can use a different subpartitioning scheme, or none at all.
- You must first create the parent table, and it must be a sharded table.
- You can then use the `CREATE SHARDED TABLE` ... `PARENT` ... statement to create each child table. For *`table`*, specify the name of the parent table. For *`schema`*, specify the schema that contains the parent table. If you omit *`schema`*, then the database assumes the parent table is in your own schema.

Regardless of which method you use, you can create at most one sharded table family in a sharded database (SDB).

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on creating sharded table families using reference partitioning or the `PARENT` clause

Examples

Creating Tables: General Examples

This statement shows how the `employees` table owned by the sample human resources (`hr`) schema was created. A hypothetical name is given to the table and constraints so that you can duplicate this example in your test database:

```
CREATE TABLE employees_demo
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
         CONSTRAINT emp_last_name_nn_demo NOT NULL
    , email          VARCHAR2(25)
         CONSTRAINT emp_email_nn_demo     NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE  DEFAULT SYSDATE
         CONSTRAINT emp_hire_date_nn_demo  NOT NULL
    , job_id         VARCHAR2(10)
       CONSTRAINT     emp_job_nn_demo  NOT NULL
    , salary         NUMBER(8,2)
       CONSTRAINT     emp_salary_nn_demo  NOT NULL
```

```
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , dn             VARCHAR2(300)
    , CONSTRAINT     emp_salary_min_demo
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk_demo
                     UNIQUE (email)
    ) ;
```

This table contains twelve columns. The `employee_id` column is of data type `NUMBER`. The `hire_date` column is of data type `DATE` and has a default value of `SYSDATE`. The `last_name` column is of type `VARCHAR2` and has a `NOT NULL` constraint, and so on.

Creating a Table: Storage Example

To define the same `employees_demo` table in the `example` tablespace with a small storage capacity, issue the following statement:

```
CREATE TABLE employees_demo
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
         CONSTRAINT emp_last_name_nn_demo NOT NULL
    , email          VARCHAR2(25)
         CONSTRAINT emp_email_nn_demo     NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE  DEFAULT SYSDATE
         CONSTRAINT emp_hire_date_nn_demo  NOT NULL
    , job_id         VARCHAR2(10)
       CONSTRAINT     emp_job_nn_demo  NOT NULL
    , salary         NUMBER(8,2)
       CONSTRAINT     emp_salary_nn_demo  NOT NULL
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , dn             VARCHAR2(300)
    , CONSTRAINT     emp_salary_min_demo
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk_demo
                     UNIQUE (email)
    )
   TABLESPACE example
   STORAGE (INITIAL 8M);
```

Creating a Table with a DEFAULT ON NULL Column Value: Example

The following statement creates a table `myemp`, which can be used to store employee data. The `department_id` column is defined with a `DEFAULT ON NULL` column value of 50. Therefore, if a subsequent `INSERT` statement attempts to assign a NULL value to `department_id`, then the value of 50 will be assigned instead.

```
CREATE TABLE myemp (employee_id number, last_name varchar2(25),
                    department_id NUMBER DEFAULT ON NULL 50 NOT NULL);
```

In the `employees` table, `employee_id` 178 has a NULL value for `department_id`:

```
SELECT employee_id, last_name, department_id
  FROM employees
  WHERE department_id IS NULL;
```

```
EMPLOYEE_ID LAST_NAME                 DEPARTMENT_ID
----------- ------------------------- -------------
        178 Grant
```

Populate the myemp table with the employee_id, last_name, and department_id column data from the employees table:

```
INSERT INTO myemp (employee_id, last_name, department_id)
  (SELECT employee_id, last_name, department_id from employees);
```

In the myemp table, employee_id 178 has a value of 50 for department_id:

```
SELECT employee_id, last_name, department_id
  FROM myemp
  WHERE employee_id = 178;

EMPLOYEE_ID LAST_NAME                 DEPARTMENT_ID
----------- ------------------------- -------------
        178 Grant                                50
```

Creating a Table with an Identity Column: Examples

The following statement creates a table t1 with an identity column id. The sequence generator will always assign increasing integer values to id, starting with 1.

```
CREATE TABLE t1 (id NUMBER GENERATED AS IDENTITY);
```

The following statement creates a table t2 with an identity column id. The sequence generator will, by default, assign increasing integer values to id in increments of 10 starting with 100.

```
CREATE TABLE t2 (id NUMBER GENERATED BY DEFAULT AS IDENTITY (START WITH 100
INCREMENT BY 10));
```

Creating a Table: Temporary Table Example

The following statement creates a temporary table today_sales for use by sales representatives in the sample database. Each sales representative session can store its own sales data for the day in the table. The temporary data is deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE today_sales
   ON COMMIT PRESERVE ROWS
   AS SELECT * FROM orders WHERE order_date = SYSDATE;
```

Creating a Table with Deferred Segment Creation: Example

The following statement creates a table with deferred segment creation. Oracle Database will not create a segment for the data of this table until data is inserted into the table:

```
CREATE TABLE later (col1 NUMBER, col2 VARCHAR2(20))    SEGMENT CREATION DEFERRED;
```

Substitutable Table and Column Examples

The following statements create a type hierarchy, which can be used to create a substitutable table. Type employee_t inherits the name and ssn attributes from type person_t and in addition has department_id and salary attributes. Type part_time_emp_t inherits all of the attributes from employee_t and, through employee_t, those of person_t and in addition has a num_hrs attribute. Type part_time_emp_t is final by default, so no further subtypes can be created under it.

```
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;
/

CREATE TYPE employee_t UNDER person_t
   (department_id NUMBER, salary NUMBER) NOT FINAL;
/

CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
/
```

The following statement creates a substitutable table from the person_t type:

```
CREATE TABLE persons OF person_t;
```

The following statement creates a table with a substitutable column of type person_t:

```
CREATE TABLE books (title VARCHAR2(100), author person_t);
```

When you insert into persons or books, you can specify values for the attributes of person_t or any of its subtypes. Examples of insert statements appear in "Inserting into a Substitutable Tables and Columns: Examples".

You can extract data from such tables using built-in functions and conditions. For examples, see the functions TREAT and SYS_TYPEID , and the "IS OF *type* Condition " condition.

Creating a Table: Parallelism Examples

The following statement creates a table using an optimum number of parallel execution servers to scan employees and to populate dept_80:

```
CREATE TABLE dept_80
   PARALLEL
   AS SELECT * FROM employees
   WHERE department_id = 80;
```

Using parallelism speeds up the creation of the table, because the database uses parallel execution servers to create the table. After the table is created, querying the table is also faster, because the same degree of parallelism is used to access the table.

The following statement creates the same table serially. Subsequent DML and queries on the table will also be serially executed.

```
CREATE TABLE dept_80
   AS SELECT * FROM employees
   WHERE department_id = 80;
```

Creating a Table: ENABLE/DISABLE Examples

The following statement shows how the sample table departments was created. The example defines a NOT NULL constraint, and places it in ENABLE VALIDATE state. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TABLE departments_demo
    ( department_id    NUMBER(4)
    , department_name  VARCHAR2(30)
         CONSTRAINT    dept_name_nn  NOT NULL
    , manager_id       NUMBER(6)
```

```
    , location_id      NUMBER(4)
    , dn               VARCHAR2(300)
    ) ;
```

The following statement creates the same `departments_demo` table but also defines a disabled primary key constraint:

```
CREATE TABLE departments_demo
    ( department_id    NUMBER(4)   PRIMARY KEY DISABLE
    , department_name  VARCHAR2(30)
         CONSTRAINT    dept_name_nn  NOT NULL
    , manager_id       NUMBER(6)
    , location_id      NUMBER(4)
    , dn               VARCHAR2(300)
    ) ;
```

Nested Table Example

The following statement shows how the sample table `pm.print_media` was created with a nested table column `ad_textdocs_ntab`:

```
CREATE TABLE print_media
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab;
```

Creating a Table: Multilevel Collection Example

The following example shows how an account manager might create a table of customers using two levels of nested tables:

```
CREATE TYPE phone AS OBJECT (telephone NUMBER);
/
CREATE TYPE phone_list AS TABLE OF phone;
/
CREATE TYPE my_customers AS OBJECT (
   cust_name VARCHAR2(25),
   phones phone_list);
/
CREATE TYPE customer_list AS TABLE OF my_customers;
/
CREATE TABLE business_contacts (
   company_name VARCHAR2(25),
   company_reps customer_list)
   NESTED TABLE company_reps STORE AS outer_ntab
   (NESTED TABLE phones STORE AS inner_ntab);
```

The following variation of this example shows how to use the `COLUMN_VALUE` keyword if the inner nested table has no column or attribute name:

```
CREATE TYPE phone AS TABLE OF NUMBER;
/
CREATE TYPE phone_list AS TABLE OF phone;
```

```
/
CREATE TABLE my_customers (
   name VARCHAR2(25),
   phone_numbers phone_list)
   NESTED TABLE phone_numbers STORE AS outer_ntab
   (NESTED TABLE COLUMN_VALUE STORE AS inner_ntab);
```

Creating a Table: LOB Column Example

The following statement is a variation of the statement that created the `pm.print_media` table with some added LOB storage characteristics:

```
CREATE TABLE print_media_new
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab_new
    LOB (ad_sourcetext, ad_finaltext) STORE AS
      (TABLESPACE example
       STORAGE (INITIAL 6144)
       CHUNK 4000
       NOCACHE LOGGING);
```

In the example, the database rounds the value of `CHUNK` up to 4096 (the nearest multiple of the block size of 2048).

Index-Organized Table Example

The following statement is a variation of the sample table `hr.countries`, which is index organized:

```
CREATE TABLE countries_demo
    ( country_id      CHAR(2)
      CONSTRAINT country_id_nn_demo NOT NULL
    , country_name    VARCHAR2(40)
    , currency_name   VARCHAR2(25)
    , currency_symbol VARCHAR2(3)
    , region          VARCHAR2(15)
    , CONSTRAINT    country_c_id_pk_demo
                    PRIMARY KEY (country_id ) )
    ORGANIZATION INDEX
    INCLUDING   country_name
    PCTTHRESHOLD 2
    STORAGE
     ( INITIAL  4K )
   OVERFLOW
    STORAGE
      ( INITIAL  4K );
```

External Table Example

The following statement creates an external table that represents a subset of the sample table `hr.departments`. The `TYPE` clause specifies that the access driver type for the table is `ORACLE_LOADER`. The `ACCESS PARAMETERS()` clause specifies parameter values for the

ORACLE_LOADER access driver. These parameters are shown in italics and form the *opaque_format_spec*. The syntax for *opaque_format_spec* depends on the access driver type and is outside the scope of this document. Refer to *Oracle Database Utilities* for details on the ORACLE_LOADER access driver and the *opaque_format_spec* syntax.

```
CREATE TABLE dept_external (
   deptno     NUMBER(6),
   dname      VARCHAR2(20),
   loc        VARCHAR2(25)
)
ORGANIZATION EXTERNAL
(TYPE oracle_loader
 DEFAULT DIRECTORY admin
 ACCESS PARAMETERS
 (
  RECORDS DELIMITED BY newline
  BADFILE 'ulcase1.bad'
  DISCARDFILE 'ulcase1.dis'
  LOGFILE 'ulcase1.log'
  SKIP 20
  FIELDS TERMINATED BY ","  OPTIONALLY ENCLOSED BY '"'
  (
   deptno     INTEGER EXTERNAL(6),
   dname      CHAR(20),
   loc        CHAR(25)
  )
 )
 LOCATION ('ulcase1.ctl')
)
REJECT LIMIT UNLIMITED;
```

See Also:

"Creating a Directory: Examples" to see how the admin directory was created

XMLType Examples

This section contains brief examples of creating an XMLType table or XMLType column. For a more expanded version of these examples, refer to "Using XML in SQL Statements ".

XMLType Table Examples

The following example creates a very simple XMLType table with one implicit binary XML column:

```
CREATE TABLE xwarehouses OF XMLTYPE;
```

The following example creates an XMLSchema-based table. The XMLSchema must already have been created (see "Using XML in SQL Statements " for more information):

```
CREATE TABLE xwarehouses OF XMLTYPE
   XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
   ELEMENT "Warehouse";
```

You can define constraints on an XMLSchema-based table, and you can also create indexes on XMLSchema-based tables, which greatly enhance subsequent queries. You can create object-relational views on `XMLType` tables, and you can create `XMLType` views on object-relational tables.

> **See Also:**
>
> - "Using XML in SQL Statements " for an example of adding a constraint
> - "Creating an Index on an XMLType Table: Example" for an example of creating an index
> - "Creating an XMLType View: Example" for an example of creating an `XMLType` view

XMLType Column Examples

The following example creates a table with an `XMLType` column stored as a `CLOB`. This table does not require an XMLSchema, so the content structure is not predetermined:

```
CREATE TABLE xwarehouses (
   warehouse_id        NUMBER,
   warehouse_spec      XMLTYPE)
   XMLTYPE warehouse_spec STORE AS CLOB
   (TABLESPACE example
    STORAGE (INITIAL 6144)
    CHUNK 4000
    NOCACHE LOGGING);
```

The following example creates a similar table, but stores `XMLType` data in an object relational `XMLType` column whose structure is determined by the specified schema:

```
CREATE TABLE xwarehouses (
   warehouse_id    NUMBER,
   warehouse_spec  XMLTYPE)
   XMLTYPE warehouse_spec STORE AS OBJECT RELATIONAL
      XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
      ELEMENT "Warehouse";
```

The following example creates another similar table with an `XMLType` column stored as a SecureFiles `CLOB`. This table does not require an XMLSchema, so the content structure is not predetermined. SecureFiles LOBs require a tablespace with automatic segment-space management, so the example uses the tablespace created in "Specifying Segment Space Management for a Tablespace: Example".

```
CREATE TABLE xwarehouses (
  warehouse_id   NUMBER,
  warehouse_spec XMLTYPE)
  XMLTYPE        warehouse_spec STORE AS SECUREFILE CLOB
  (TABLESPACE auto_seg_ts
  STORAGE (INITIAL 6144)
  CACHE);
```

Partitioning Examples

Range Partitioning Example

The `sales` table in the sample schema `sh` is partitioned by range. The following example shows an abbreviated variation of the `sales` table. Constraints and storage elements have been omitted from the example.

```
CREATE TABLE range_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold         NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
  (PARTITION SALES_Q1_1998 VALUES LESS THAN (TO_DATE('01-APR-1998','DD-MON-YYYY')),
   PARTITION SALES_Q2_1998 VALUES LESS THAN (TO_DATE('01-JUL-1998','DD-MON-YYYY')),
   PARTITION SALES_Q3_1998 VALUES LESS THAN (TO_DATE('01-OCT-1998','DD-MON-YYYY')),
   PARTITION SALES_Q4_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
   PARTITION SALES_Q1_1999 VALUES LESS THAN (TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION SALES_Q2_1999 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION SALES_Q3_1999 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION SALES_Q4_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
   PARTITION SALES_Q1_2000 VALUES LESS THAN (TO_DATE('01-APR-2000','DD-MON-YYYY')),
   PARTITION SALES_Q2_2000 VALUES LESS THAN (TO_DATE('01-JUL-2000','DD-MON-YYYY')),
   PARTITION SALES_Q3_2000 VALUES LESS THAN (TO_DATE('01-OCT-2000','DD-MON-YYYY')),
   PARTITION SALES_Q4_2000 VALUES LESS THAN (MAXVALUE))
;
```

For information about partitioned table maintenance operations, see *Oracle Database VLDB and Partitioning Guide*.

Range Partitioning Live SQL Example

The following statement creates a table partitioned by range:

```
CREATE TABLE empl_h
  (
     employee_id  NUMBER(6) PRIMARY KEY,
     first_name   VARCHAR2(20),
     last_name    VARCHAR2(25),
     email        VARCHAR2(25),
     phone_number VARCHAR2(20),
     hire_date    DATE DEFAULT SYSDATE,
     job_id       VARCHAR2(10),
     salary       NUMBER(8, 2),
     part_name    VARCHAR2(25)
  ) PARTITION BY RANGE (hire_date) (
PARTITION hire_q1 VALUES less than(to_date('01-APR-2014', 'DD-MON-YYYY')),
PARTITION hire_q2 VALUES less than(to_date('01-JUL-2014', 'DD-MON-YYYY')),
PARTITION hire_q3 VALUES less than(to_date('01-OCT-2014', 'DD-MON-YYYY')),
PARTITION hire_q4 VALUES less than(to_date('01-JAN-2015', 'DD-MON-YYYY'))
);
```

The following statements insert rows into the partitions:

```
INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number,
hire_date, job_id, salary, Part_name)
VALUES (1, 'Jane', 'Doe', 'example.com', '415.555.0100', '10-Feb-2014', '1001',
5001,'HIRE_Q1');

INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number,
```

```
hire_date, job_id, salary, Part_name)
VALUES (2, 'John', 'Doe', 'example.net', '415.555.0101', '10-Apr-2014', '1002',
7001,'HIRE_Q2');

INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number,
hire_date, job_id, salary, Part_name)
VALUES (3, 'Isabelle', 'Owl', 'example.org', '415.555.0102', '10-Sep-2014', '1003',
10001,'HIRE_Q3');

INSERT INTO empl_h (employee_id, first_name, last_name, email, phone_number,
hire_date, job_id, salary, Part_name)
VALUES (4, 'Smith', 'Jones', 'example.in', '415.555.0103', '10-Dec-2014', '1004',
12001,'HIRE_Q4');
```

The following statements display the partition names using data dictionary tables:

```
SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE TABLE_NAME = 'EMPL_H';

PARTITION_NAME
----------------
HIRE_Q1
HIRE_Q2
HIRE_Q3
HIRE_Q4

SELECT TABLE_NAME, PARTITIONING_TYPE, STATUS FROM USER_PART_TABLES WHERE TABLE_NAME =
'EMPL_H';

TABLE_NAME    PARTITIONING_TYPE    STATUS
----------    -----------------    ------
EMPL_H        RANGE            VALID
```

The following statement creates a table named `parts` by selecting a particular column from the data dictionary table `user_tab_partitions`:

```
CREATE TABLE parts (p_name) AS SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE
TABLE_NAME = 'EMPL_H';
```

The following statement displays the table data:

```
select * from parts;

P_NAME
-----------
HIRE_Q1
HIRE_Q2
HIRE_Q3
HIRE_Q4
```

The following statement compares the columns from the two tables and displays the information based on the comparison:

```
select E.HIRE_DATE,E.JOB_ID,P.p_name from empl_h E, parts P where E.Part_name =
P.p_name;

HIRE_DATE JOB_ID     P_NAME
--------- ---------- ------------
10-FEB-14 1001         HIRE_Q1
10-APR-14 1002         HIRE_Q2
10-SEP-14 1003         HIRE_Q3
10-DEC-14 1004         HIRE_Q4
```

Interval Partitioning Example

The following example creates a variation of the `oe.customers` table that is partitioned by interval on the `credit_limit` column. One range partition is created to establish the transition point. All of the original data in the table is within the bounds of the range partition. Then data is added that exceeds the range partition, and the database creates a new interval partition.

```
CREATE TABLE customers_demo (
  customer_id number(6),
  cust_first_name varchar2(20),
  cust_last_name varchar2(20),
  credit_limit number(9,2))
PARTITION BY RANGE (credit_limit)
INTERVAL (1000)
(PARTITION p1 VALUES LESS THAN (5001));

INSERT INTO customers_demo
  (customer_id, cust_first_name, cust_last_name, credit_limit)
  (select customer_id, cust_first_name, cust_last_name, credit_limit
  from customers);
```

Query the `USER_TAB_PARTITIONS` data dictionary view before the database creates the interval partition:

```
SELECT partition_name, high_value FROM user_tab_partitions  WHERE table_name =
'CUSTOMERS_DEMO';

PARTITION_NAME                 HIGH_VALUE
------------------------------ ---------------
P1                             5001
```

Insert data into the table that exceeds the high value of the range partition:

```
INSERT INTO customers_demo
  VALUES (699, 'Fred', 'Flintstone', 5500);
```

Query the `USER_TAB_PARTITIONS` view again after the insert to learn the system-generated name of the interval partition created to accommodate the inserted data. (The system-generated name will vary for each session.)

```
SELECT partition_name, high_value FROM user_tab_partitions
  WHERE table_name = 'CUSTOMERS_DEMO'
  ORDER BY partition_name;

PARTITION_NAME                 HIGH_VALUE
------------------------------ ---------------
P1                             5001
SYS_P44                        6001
```

List Partitioning Example

The following statement shows how the sample table `oe.customers` might have been created as a list-partitioned table. Some columns and all constraints of the sample table have been omitted in this example.

```
CREATE TABLE list_customers
   ( customer_id             NUMBER(6)
   , cust_first_name         VARCHAR2(20)
   , cust_last_name          VARCHAR2(20)
```

```
   , cust_address            CUST_ADDRESS_TYP
   , nls_territory           VARCHAR2(30)
   , cust_email              VARCHAR2(40))
   PARTITION BY LIST (nls_territory) (
   PARTITION asia VALUES ('CHINA', 'THAILAND'),
   PARTITION europe VALUES ('GERMANY', 'ITALY', 'SWITZERLAND'),
   PARTITION west VALUES ('AMERICA'),
   PARTITION east VALUES ('INDIA'),
   PARTITION rest VALUES (DEFAULT));
```

Partitioned Table with LOB Columns Example

This statement creates a partitioned table `print_media_demo` with two partitions `p1` and `p2`, and a number of LOB columns. The statement uses the sample table `pm.print_media`.

```
CREATE TABLE print_media_demo
   ( product_id NUMBER(6)
   , ad_id NUMBER(6)
   , ad_composite BLOB
   , ad_sourcetext CLOB
   , ad_finaltext CLOB
   , ad_fltextn NCLOB
   , ad_textdocs_ntab textdoc_tab
   , ad_photo BLOB
   , ad_graphic BFILE
   , ad_header adheader_typ
   ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab_demo
     LOB (ad_composite, ad_photo, ad_finaltext)
     STORE AS(STORAGE (INITIAL 20M))
   PARTITION BY RANGE (product_id)
      (PARTITION p1 VALUES LESS THAN (3000) TABLESPACE tbs_01
         LOB (ad_composite, ad_photo)
         STORE AS (TABLESPACE tbs_02 STORAGE (INITIAL 10M))
         NESTED TABLE ad_textdocs_ntab STORE AS nt_p1 (TABLESPACE example),
       PARTITION P2 VALUES LESS THAN (MAXVALUE)
         LOB (ad_composite, ad_finaltext)
         STORE AS SECUREFILE (TABLESPACE auto_seg_ts)
         NESTED TABLE ad_textdocs_ntab STORE AS nt_p2
       )
   TABLESPACE tbs_03;
```

Partition `p1` will be in tablespace `tbs_01`. The LOB data partitions for `ad_composite` and `ad_photo` will be in tablespace `tbs_02`. The LOB data partition for the remaining LOB columns will be in tablespace `tbs_01`. The storage attribute `INITIAL` is specified for LOB columns `ad_composite` and `ad_photo`. Other attributes will be inherited from the default table-level specification. The default LOB storage attributes not specified at the table level will be inherited from the tablespace `tbs_02` for columns `ad_composite` and `ad_photo` and from tablespace `tbs_01` for the remaining LOB columns. LOB index partitions will be in the same tablespaces as the corresponding LOB data partitions. Other storage attributes will be based on values of the corresponding attributes of the LOB data partitions and default attributes of the tablespace where the index partitions reside. The nested table partition for ad_textdocs_ntab will be stored as `nt_p1` in tablespace `example`.

Partition `p2` will be in the default tablespace `tbs_03`. The LOB data for `ad_composite` and `ad_finaltext` will be in tablespace `auto_seg_ts` as SecureFiles LOBs. The LOB data for the remaining LOB columns will be in tablespace `tbs_03`. The LOB index for columns `ad_composite` and `ad_finaltext` will be in tablespace `auto_seg_ts`. The LOB index for the remaining LOB columns will be in tablespace `tbs_03`. The nested table partition for `ad_textdocs_ntab` will be stored as `nt_p2` in the default tablespace `tbs_03`.

Hash Partitioning Example

The sample table oe.product_information is not partitioned. However, you might want to partition such a large table by hash for performance reasons, as shown in this example. The tablespace names are hypothetical in this example.

```
CREATE TABLE hash_products
    ( product_id          NUMBER(6)   PRIMARY KEY
    , product_name        VARCHAR2(50)
    , product_description VARCHAR2(2000)
    , category_id         NUMBER(2)
    , weight_class        NUMBER(1)
    , warranty_period     INTERVAL YEAR TO MONTH
    , supplier_id         NUMBER(6)
    , product_status      VARCHAR2(20)
    , list_price          NUMBER(8,2)
    , min_price           NUMBER(8,2)
    , catalog_url         VARCHAR2(50)
    , CONSTRAINT          product_status_lov_demo
                          CHECK (product_status in ('orderable'
                                                  ,'planned'
                                                  ,'under development'
                                                  ,'obsolete')
 ) )
 PARTITION BY HASH (product_id)
 PARTITIONS 4
 STORE IN (tbs_01, tbs_02, tbs_03, tbs_04);
```

Reference Partitioning Example

The next statement uses the hash_products partitioned table created in the preceding example. It creates a variation of the oe.order_items table that is partitioned by reference to the hash partitioning on the product id of hash_products. The resulting child table will be created with five partitions. For each row of the child table part_order_items, the database evaluates the foreign key value (product_id) to determine the partition number of the parent table hash_products to which the referenced key belongs. The part_order_items row is placed in its corresponding partition.

```
CREATE TABLE part_order_items (
    order_id        NUMBER(12) PRIMARY KEY,
    line_item_id    NUMBER(3),
    product_id      NUMBER(6) NOT NULL,
    unit_price      NUMBER(8,2),
    quantity        NUMBER(8),
    CONSTRAINT product_id_fk
    FOREIGN KEY (product_id) REFERENCES hash_products(product_id))
 PARTITION BY REFERENCE (product_id_fk);
```

Composite-Partitioned Table Examples

The table created in the "Range Partitioning Example" divides data by time of sale. If you plan to access recent data according to distribution channel as well as time, then composite partitioning might be more appropriate. The following example creates a copy of that range_sales table but specifies range-hash composite partitioning. The partitions with the most recent data are subpartitioned with both system-generated and user-defined subpartition names. Constraints and storage attributes have been omitted from the example.

```
CREATE TABLE composite_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold         NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
SUBPARTITION BY HASH (channel_id)
  (PARTITION SALES_Q1_1998 VALUES LESS THAN (TO_DATE('01-APR-1998','DD-MON-YYYY')),
   PARTITION SALES_Q2_1998 VALUES LESS THAN (TO_DATE('01-JUL-1998','DD-MON-YYYY')),
   PARTITION SALES_Q3_1998 VALUES LESS THAN (TO_DATE('01-OCT-1998','DD-MON-YYYY')),
   PARTITION SALES_Q4_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
   PARTITION SALES_Q1_1999 VALUES LESS THAN (TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION SALES_Q2_1999 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION SALES_Q3_1999 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION SALES_Q4_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
   PARTITION SALES_Q1_2000 VALUES LESS THAN (TO_DATE('01-APR-2000','DD-MON-YYYY')),
   PARTITION SALES_Q2_2000 VALUES LESS THAN (TO_DATE('01-JUL-2000','DD-MON-YYYY'))
      SUBPARTITIONS 8,
   PARTITION SALES_Q3_2000 VALUES LESS THAN (TO_DATE('01-OCT-2000','DD-MON-YYYY'))
     (SUBPARTITION ch_c,
      SUBPARTITION ch_i,
      SUBPARTITION ch_p,
      SUBPARTITION ch_s,
      SUBPARTITION ch_t),
   PARTITION SALES_Q4_2000 VALUES LESS THAN (MAXVALUE)
      SUBPARTITIONS 4)
;
```

The following examples creates a partitioned table of customers based on the sample table `oe.customers`. In this example, the table is partitioned on the `credit_limit` column and list subpartitioned on the `nls_territory` column. The subpartition template determines the subpartitioning of any subsequently added partitions, unless you override the template by defining individual subpartitions. This composite partitioning makes it possible to query the table based on a credit limit range within a specified region:

```
CREATE TABLE customers_part (
   customer_id        NUMBER(6),
   cust_first_name    VARCHAR2(20),
   cust_last_name     VARCHAR2(20),
   nls_territory      VARCHAR2(30),
   credit_limit       NUMBER(9,2))
   PARTITION BY RANGE (credit_limit)
   SUBPARTITION BY LIST (nls_territory)
      SUBPARTITION TEMPLATE
         (SUBPARTITION east  VALUES
            ('CHINA', 'JAPAN', 'INDIA', 'THAILAND'),
          SUBPARTITION west VALUES
             ('AMERICA', 'GERMANY', 'ITALY', 'SWITZERLAND'),
          SUBPARTITION other VALUES (DEFAULT))
      (PARTITION p1 VALUES LESS THAN (1000),
       PARTITION p2 VALUES LESS THAN (2500),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Object Column and Table Examples

Creating Object Tables: Examples

Consider object type `department_typ`:

```
CREATE TYPE department_typ AS OBJECT
   ( d_name   VARCHAR2(100),
     d_address VARCHAR2(200) );
/
```

Object table `departments_obj_t` holds department objects of type `department_typ`:

```
CREATE TABLE departments_obj_t OF department_typ;
```

The following statement creates object table `salesreps` with a user-defined object type, `salesrep_typ`:

```
CREATE OR REPLACE TYPE salesrep_typ AS OBJECT
  ( repId NUMBER,
    repName VARCHAR2(64));

CREATE TABLE salesreps OF salesrep_typ;
```

Creating a Table with a User-Defined Object Identifier: Example

This example creates an object type and a corresponding object table whose object identifier is primary key based:

```
CREATE TYPE employees_typ AS OBJECT
   (e_no NUMBER, e_address CHAR(30));
/

CREATE TABLE employees_obj_t OF employees_typ (e_no PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

You can subsequently reference the `employees_obj_t` object table using either *`inline_ref_constraint`* or *`out_of_line_ref_constraint`* syntax:

```
CREATE TABLE departments_t
   (d_no    NUMBER,
    mgr_ref REF employees_typ SCOPE IS employees_obj_t);

CREATE TABLE departments_t (
    d_no NUMBER,
    mgr_ref REF employees_typ
       CONSTRAINT mgr_in_emp REFERENCES employees_obj_t);
```

Specifying Constraints on Type Columns: Example

The following example shows how to define constraints on attributes of an object type column:

```
CREATE TYPE address_t AS OBJECT
  ( hno    NUMBER,
    street VARCHAR2(40),
    city   VARCHAR2(20),
    zip    VARCHAR2(5),
    phone  VARCHAR2(10) );
/

CREATE TYPE person AS OBJECT
  ( name        VARCHAR2(40),
    dateofbirth DATE,
```

```
    homeaddress address_t,
    manager     REF person );
/

CREATE TABLE persons OF person
  ( homeaddress NOT NULL,
      UNIQUE (homeaddress.phone),
      CHECK (homeaddress.zip IS NOT NULL),
      CHECK (homeaddress.city <> 'San Francisco') );
```

CREATE TABLESPACE

Purpose

Use the `CREATE TABLESPACE` statement to create a **tablespace**, which is an allocation of space in the database that can contain schema objects.

- A **permanent tablespace** contains persistent schema objects. Objects in permanent tablespaces are stored in **data files**.
- An **undo tablespace** is a type of permanent tablespace used by Oracle Database to manage undo data if you are running your database in automatic undo management mode. Oracle strongly recommends that you use automatic undo management mode rather than using rollback segments for undo.
- A **temporary tablespace** contains schema objects only for the duration of a session. Objects in temporary tablespaces are stored in **temp files**.

When you create a tablespace, it is initially a read/write tablespace. You can subsequently use the `ALTER TABLESPACE` statement to take the tablespace offline or online, add data files or temp files to it, or make it a read-only tablespace.

You can also drop a tablespace from the database with the `DROP TABLESPACE` statement.

> **See Also:**
>
> - *Oracle Database Concepts* for information on tablespaces
> - ALTER TABLESPACE and DROP TABLESPACE for information on modifying and dropping tablespaces

Prerequisites

You must have the `CREATE TABLESPACE` system privilege. To create the `SYSAUX` tablespace, you must have the `SYSDBA` system privilege.

Before you can create a tablespace, you must create a database to contain it, and the database must be open.

> **See Also:**
>
> CREATE DATABASE

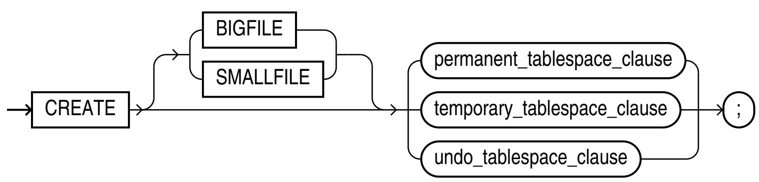
To use objects in a tablespace other than the `SYSTEM` tablespace:

- If you are running the database in automatic undo management mode, then at least one `UNDO` tablespace must be online.
- If you are running the database in manual undo management mode, then at least one rollback segment other than the `SYSTEM` rollback segment must be online.

> **Note:**
>
> Oracle strongly recommends that you run your database in automatic undo management mode. For more information, refer to *Oracle Database Administrator's Guide*.

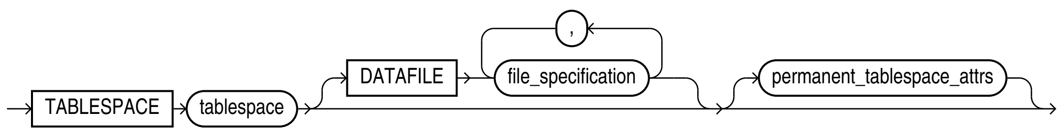
Syntax

***create_tablespace*::=**



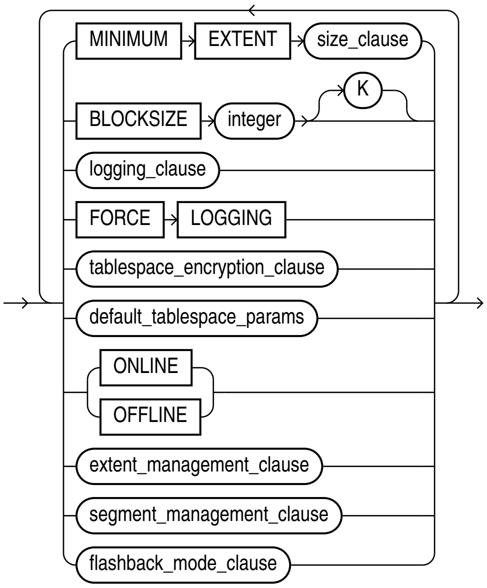
(*permanent_tablespace_clause::=*, *temporary_tablespace_clause::=*, *undo_tablespace_clause::=*)

***permanent_tablespace_clause*::=**



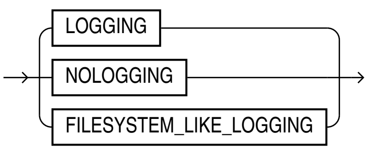
(*file_specification::=*, *permanent_tablespace_attrs::=*)

***permanent_tablespace_attrs*::=**

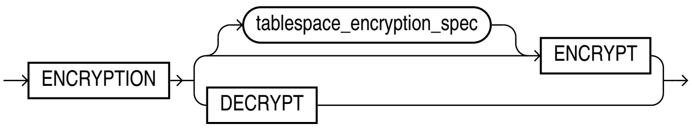


(*size_clause*::=, *logging_clause*::=, *tablespace_encryption_clause*::=, *default_tablespace_params*::=, *extent_management_clause*::=, *segment_management_clause*::=, *flashback_mode_clause*::=)

***logging_clause*::=**

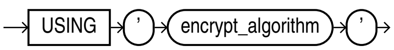


***tablespace_encryption_clause*::=**

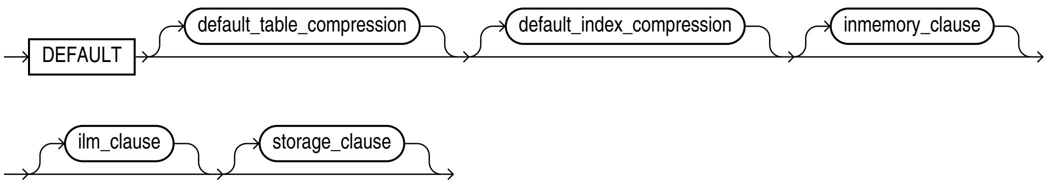


(*tablespace_encryption_spec*::=)

***tablespace_encryption_spec*::=**



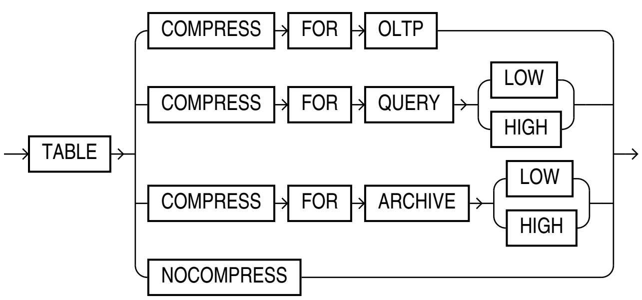
default_tablespace_params::=



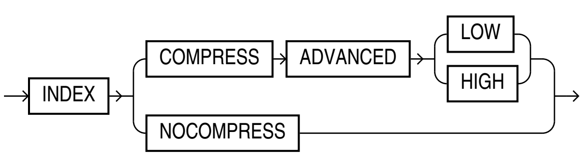
(*default_table_compression*::=, *default_index_compression*::=, *inmemory_clause*::=, *ilm_clause*::=—part of `CREATE TABLE` syntax, *storage_clause*::=)

> **Note:**
>
> If you specify the `DEFAULT` clause, then you must specify at least one of the clauses *`default_table_compression`*, *`default_index_compression`*, *`inmemory_clause`*, *`ilm_clause`*, or *`storage_clause`*.

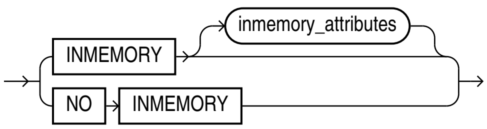
default_table_compression::=



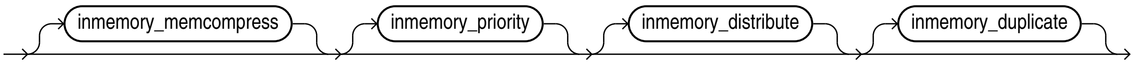
default_index_compression::=



inmemory_clause::=

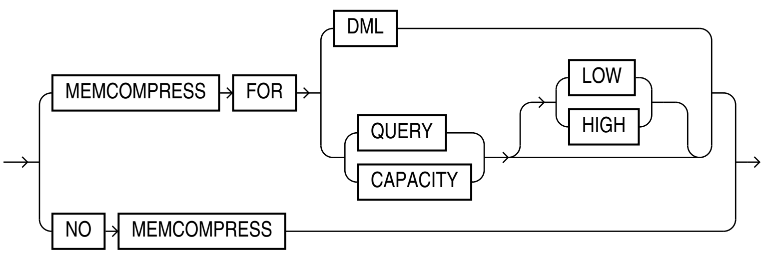


***inmemory_attributes*::=**

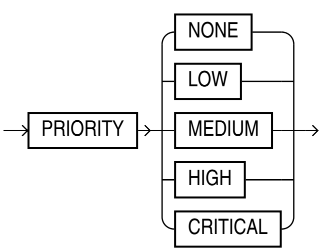


(*inmemory_memcompress::=*, *inmemory_priority::=*, *inmemory_distribute::=*, *inmemory_duplicate::=*)

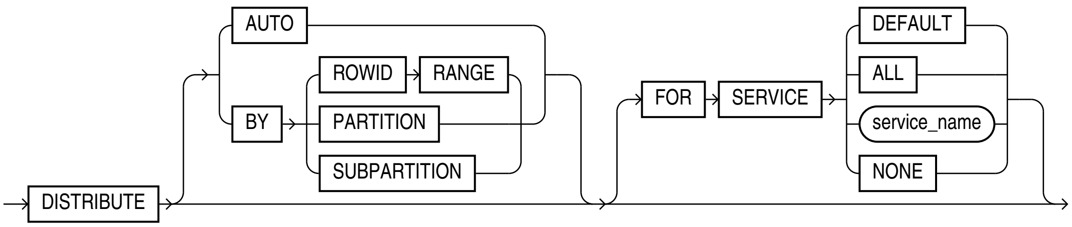
***inmemory_memcompress*::=**



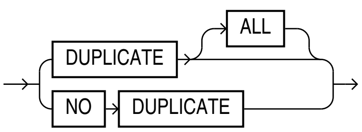
***inmemory_priority*::=**



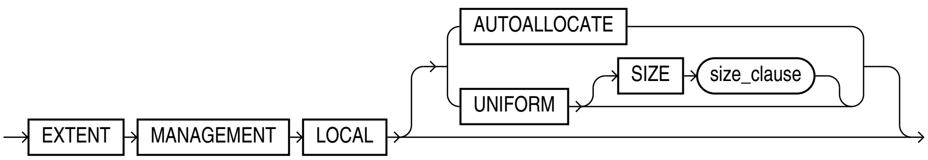
***inmemory_distribute*::=**



***inmemory_duplicate*::=**

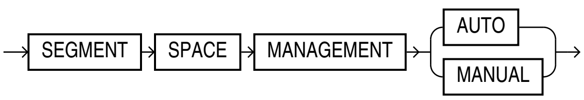


***extent_management_clause*::=**

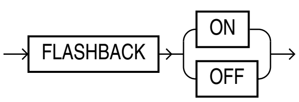


(*size_clause*::=)

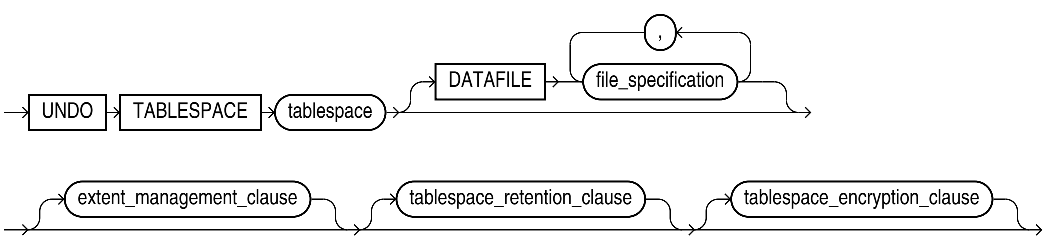
***segment_management_clause*::=**



***flashback_mode_clause*::=**

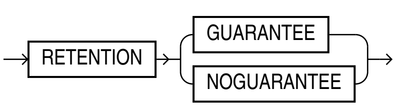


***undo_tablespace_clause*::=**

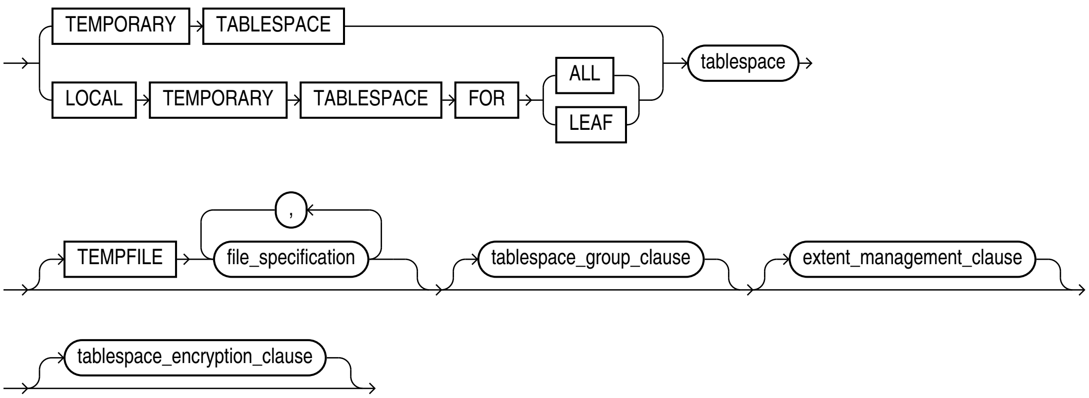


(*file_specification*::=, *extent_management_clause*::=, *tablespace_retention_clause*::=)

***tablespace_retention_clause*::=**

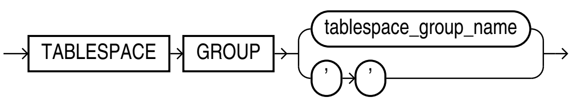


***temporary_tablespace_clause*::=**



(*file_specification*::=, *tablespace_group_clause*::=, *extent_management_clause*::=, *tablespace_encryption_clause*::=)

***tablespace_group_clause*::=**



Semantics

BIGFILE | SMALLFILE

Use this clause to determine whether the tablespace is a bigfile or smallfile tablespace. This clause overrides any default tablespace type setting for the database.

- A **bigfile tablespace** contains only one data file or temp file, which can contain up to approximately 4 billion (2^{32}) blocks. The minimum size of the single data file or temp file is 12 megabytes (MB) for a tablespace with 32K blocks and 7MB for a tablespace with 8K blocks. The maximum size of the single data file or temp file is 128 terabytes (TB) for a tablespace with 32K blocks and 32TB for a tablespace with 8K blocks.
- A **smallfile tablespace** is a traditional Oracle tablespace, which can contain 1022 data files or temp files, each of which can contain up to approximately 4 million (2^{22}) blocks.

If you omit this clause, then Oracle Database uses the current default tablespace type of permanent or temporary tablespace that is set for the database. If you specify `BIGFILE` for a permanent tablespace, then the database by default creates a locally managed tablespace with automatic segment-space management.

Restriction on Bigfile Tablespaces

You can specify only one data file in the `DATAFILE` clause or one temp file in the `TEMPFILE` clause.

See Also:

- *Oracle Database Administrator's Guide* for more information on using bigfile tablespaces
- "Creating a Bigfile Tablespace: Example"

permanent_tablespace_clause

Use the following clauses to create a permanent tablespace. (Some of these clauses are also used to create a temporary or undo tablespace.)

tablespace

Specify the name of the tablespace to be created. The name must satisfy the requirements listed in "Database Object Naming Rules ".

Note on the SYSAUX Tablespace

`SYSAUX` is a required auxiliary system tablespace. You must use the `CREATE TABLESPACE` statement to create the `SYSAUX` tablespace if you are upgrading from a release earlier than Oracle Database 11*g*. You must have the `SYSDBA` system privilege to specify this clause, and you must have opened the database in `UPGRADE` mode.

You must specify `EXTENT MANAGEMENT LOCAL` and `SEGMENT SPACE MANAGEMENT AUTO` for the `SYSAUX` tablespace. The `DATAFILE` clause is optional only if you have enabled Oracle Managed Files. See "DATAFILE | TEMPFILE Clause" for the behavior of the `DATAFILE` clause.

Take care to allocate sufficient space for the `SYSAUX` tablespace. For guidelines on creating this tablespace, refer to *Oracle Database Upgrade Guide*.

Restrictions on the SYSAUX Tablespace

You cannot specify `OFFLINE` or `TEMPORARY` for the `SYSAUX` tablespace.

DATAFILE | TEMPFILE Clause

Specify the data files to make up the permanent tablespace or the temp files to make up the temporary tablespace. Use the *`datafile_tempfile_spec`* form of *`file_specification`* to create regular data files and temp files in an operating system file system or to create Oracle Automatic Storage Management (Oracle ASM) disk group files.

You must specify the `DATAFILE` or `TEMPFILE` clause unless you have enabled Oracle Managed Files by setting a value for the `DB_CREATE_FILE_DEST` initialization parameter. For Oracle ASM disk group files, the parameter must be set to a multiple file creation form of Oracle ASM filenames. If this parameter is set, then the database creates a system-named 100 MB file in the default file destination specified in the parameter. The file has `AUTOEXTEND` enabled and an unlimited maximum size.

Note:

Media recovery does not recognize temp files.

See Also:

- *Oracle Automatic Storage Management Administrator's Guide* for more information on using Oracle ASM
- *file_specification* for a full description, including the `AUTOEXTEND` parameter and the multiple file creation form of Oracle ASM filenames

Notes on Specifying Data Files and Temp Files

- You can create a tablespace within an Oracle ASM disk group by providing only the disk group name in the *`datafile_tempfile_spec`*. In this case, Oracle ASM creates a data file in the specified disk group with a system-generated filename. The data file is auto-extensible with an unlimited maximum size and a default size of 100 MB. You can use the *`autoextend_clause`* to override the default size.
- If you use one of the reference forms of the *`ASM_filename`*, which refers to an existing file, then you must also specify `REUSE`.

Note:

On some operating systems, Oracle does not allocate space for a temp file until the temp file blocks are actually accessed. This delay in space allocation results in faster creation and resizing of temp files, but it requires that sufficient disk space is available when the temp files are later used. To avoid potential problems, before you create or resize a temp file, ensure that the available disk space exceeds the size of the new temp file or the increased size of a resized temp file. The excess space should allow for anticipated increases in disk space use by unrelated operations as well. Then proceed with the creation or resizing operation.

See Also:

- *file_specification* for a full description, including the `AUTOEXTEND` parameter
- "Enabling Autoextend for a Tablespace: Example" and "Creating Oracle Managed Files: Examples"

permanent_tablespace_attrs

Use the *`permanent_tablespace_attrs`* clauses to set the attributes of the tablespace.

MINIMUM EXTENT Clause

This clause is valid only for a dictionary-managed tablespace. Specify the minimum size of an extent in the tablespace. This clause lets you control free space fragmentation in the tablespace by ensuring that the size of every used or free extent in a tablespace is at least as large as, and is a multiple of, the value specified in the `size_clause`.

> **See Also:**
>
> *size_clause* for information on that clause and *Oracle Database VLDB and Partitioning Guide* for more information about using `MINIMUM EXTENT` to control fragmentation

BLOCKSIZE Clause

Use the `BLOCKSIZE` clause to specify a nonstandard block size for the tablespace. In order to specify this clause, the `DB_CACHE_SIZE` and at least one `DB_nK_CACHE_SIZE` parameter must be set, and the integer you specify in this clause must correspond with the setting of one `DB_nK_CACHE_SIZE` parameter setting.

Restriction on BLOCKSIZE

You cannot specify nonstandard block sizes for a temporary tablespace or if you intend to assign this tablespace as the temporary tablespace for any users.

> **Note:**
>
> Oracle recommend that you do not store tablespaces with a 2K block size on 4K sector size disks, because performance degradation can result.

> **See Also:**
>
> *Oracle Database Reference* for information on the `DB_nK_CACHE_SIZE` parameter and *Oracle Database Concepts* for information on multiple block sizes

logging_clause

Specify the default logging attributes of all tables, indexes, materialized views, materialized view logs, and partitions within the tablespace. This clause is not valid for a temporary or undo tablespace.

If you omit this clause, then the default is `LOGGING`. The exception is creating a tablespace in a PDB. In this case, if you omit this clause, then the tablespace uses the logging attribute of the PDB. Refer to the *logging_clause* of `CREATE PLUGGABLE DATABASE` for more information.

The tablespace-level logging attribute can be overridden by logging specifications at the table, index, materialized view, materialized view log, and partition levels.

> **See Also:**
>
> *logging_clause* for a full description of this clause

FORCE LOGGING

Use this clause to put the tablespace into `FORCE LOGGING` mode. Oracle Database will log all changes to all objects in the tablespace except changes to temporary segments, overriding any `NOLOGGING` setting for individual objects. The database must be open and in `READ WRITE` mode.

This setting does not exclude the `NOLOGGING` attribute. You can specify both `FORCE LOGGING` and `NOLOGGING`. In this case, `NOLOGGING` is the default logging mode for objects subsequently created in the tablespace, but the database ignores this default as long as the tablespace or the database is in `FORCE LOGGING` mode. If you subsequently take the tablespace out of `FORCE LOGGING` mode, then the `NOLOGGING` default is once again enforced.

> **Note:**
>
> `FORCE LOGGING` mode can have performance effects. Refer to *Oracle Database Administrator's Guide* for information on when to use this setting.

Restriction on Forced Logging

You cannot specify `FORCE LOGGING` for an undo or temporary tablespace.

tablespace_encryption_clause

Use this clause to specify whether to create an encrypted or unencrypted tablespace. If you create an encrypted tablespace, then Transparent Data Encryption (TDE) is applied to all data files of the tablespace.

`ENCRYPT | DECRYPT`
Specify `ENCRYPT` to create an encrypted tablespace. Specify `DECRYPT` to create an unencrypted tablespace.

If you omit this clause, then the value of the `ENCRYPT_NEW_TABLESPACES` initialization parameter determines whether the tablespace is encrypted upon creation. Refer to *Oracle Database Reference* for more information on the `ENCRYPT_NEW_TABLESPACES` initialization parameter.

Before issuing this clause, you must already have loaded the TDE master key into database memory or established a connection to the HSM. For more information, see the *open_keystore* clause of `ADMINISTER KEY MANAGEMENT` or "SET ENCRYPTION WALLET Clause" of `ALTER SYSTEM`.

tablespace_encryption_spec

Specify USING '*encrypt_algorithm*' to indicate the name of the encryption algorithm to be used. Valid algorithms are AES256, AES192, AES128 and 3DES168. If the COMPATIBLE initialization parameter is set to 12.2 or higher, then the following algorithms are also valid: ARIA128, ARIA192, ARIA256, GOST256, and SEED128.. If you omit this clause, then the database uses AES128.

> **See Also:**
>
> "Creating an Encrypted Tablespace: Example"

default_tablespace_params

The DEFAULT clause lets you specify default parameters for the tablespace.

default_table_compression

Use this clause to specify default compression of data for all tables created in the tablespace. This clause is not valid for a temporary tablespace. The subclauses of this clause have the same semantics as they have for the *table_compression* clause of the CREATE TABLE statement, with one exception: The COMPRESS FOR OLTP clause here is equivalent to the ROW STORE COMPRESS ADVANCED clause of CREATE TABLE. Refer to the *table_compression* clauses of CREATE TABLE for the full semantics of these subclauses.

default_index_compression

Use this clause to specify default compression of data for all indexes created in the tablespace. This clause is not valid for a temporary tablespace. The subclauses of this clause have the same semantics as they have for the *advanced_index_compression* clause of the CREATE INDEX statement. Refer to the advanced_index_compression clause of CREATE INDEX for the full semantics of these subclauses.

inmemory_clause

Use the *inmemory_clause* to specify the default In-Memory Column Store (IM column store) settings for all tables and materialized views created in the tablespace. This clause is not valid for a temporary tablespace.

- Specify INMEMORY to enable all tables and materialized views for the IM column store.

  You can optionally use the *inmemory_attributes* clause to specify how the table or materialized view data is stored in the IM column store. The *inmemory_attributes* clause has the same semantics in CREATE TABLE and CREATE TABLESPACE. Refer to the *inmemory_attributes* clause of CREATE TABLE for the full semantics of this clause.
- Specify NO INMEMORY to disable all tables and materialized views for the IM column store. This is the default.

ilm_clause

Use the *ilm_clause* to specify default Automatic Data Optimization settings for all tables created in the tablespace. This clause is not valid for a temporary tablespace. Refer to the *ilm_clause* of CREATE TABLE for the full semantics of this clause.

storage_clause

Use the *storage_clause* to specify storage parameters for all objects created in the tablespace. This clause is not valid for a temporary tablespace or a locally managed tablespace. For a dictionary-managed tablespace, you can specify the following storage parameters with this clause: ENCRYPT, INITIAL, NEXT, MINEXTENTS, MAXEXTENTS, MAXSIZE, and PCTINCREASE. Refer to *storage_clause* for more information.

> **Note:**
>
> The ENCRYPT clause of the *storage_clause* is supported for backward compatibility. However, beginning with Oracle Database 12*c* Release 2 (12.2), you can instead specify ENCRYPT in the *tablespace_encryption_clause*. Refer to *tablespace_encryption_clause* for more information.

> **See Also:**
>
> "Creating Basic Tablespaces: Examples"

ONLINE | OFFLINE Clauses

Use these clauses to determine whether the tablespace is online or offline. This clause is not valid for a temporary tablespace.

ONLINE

Specify ONLINE to make the tablespace available immediately after creation to users who have been granted access to the tablespace. This is the default.

OFFLINE

Specify OFFLINE to make the tablespace unavailable immediately after creation.

The data dictionary view DBA_TABLESPACES indicates whether each tablespace is online or offline.

extent_management_clause

The *extent_management_clause* lets you specify how the extents of the tablespace will be managed.

> **Note:**
>
> After you have specified extent management with this clause, you can change extent management only by migrating the tablespace.

- `AUTOALLOCATE` specifies that the tablespace is system managed. Users cannot specify an extent size. You cannot specify `AUTOALLOCATE` for a temporary tablespace.
- `UNIFORM` specifies that the tablespace is managed with uniform extents of `SIZE` bytes.The default `SIZE` is 1 megabyte. All extents of temporary tablespaces are of uniform size, so this keyword is optional for a temporary tablespace. However, you must specify `UNIFORM` in order to specify `SIZE`. You cannot specify `UNIFORM` for an undo tablespace.

If you do not specify `AUTOALLOCATE` or `UNIFORM`, then the default is `UNIFORM` for temporary tablespaces and `AUTOALLOCATE` for all other types of tablespaces.

If you do not specify the *`extent_management_clause`*, then Oracle Database interprets the `MINIMUM EXTENT` clause and the `DEFAULT` *`storage_clause`* to determine extent management.

> **Note:**
>
> The `DICTIONARY` keyword is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you create locally managed tablespaces. Locally managed tablespaces are much more efficiently managed than dictionary-managed tablespaces. The creation of new dictionary-managed tablespaces is scheduled for desupport.

> **See Also:**
>
> *Oracle Database Concepts* for a discussion of locally managed tablespaces

Restrictions on Extent Management

Extent management is subject to the following restrictions:

- A permanent locally managed tablespace can contain only permanent objects. If you need a locally managed tablespace to store temporary objects, for example, if you will assign it as a user's temporary tablespace, then use the *`temporary_tablespace_clause`*.
- If you specify this clause, then you cannot specify `DEFAULT` *`storage_clause`*, `MINIMUM EXTENT`, or the *`temporary_tablespace_clause`*.

See Also:

Oracle Database Administrator's Guide for information on changing extent management by migrating tablespaces and "Creating a Locally Managed Tablespace: Example"

segment_management_clause

The *`segment_management_clause`* is relevant only for permanent, locally managed tablespaces. It lets you specify whether Oracle Database should track the used and free space in the segments in the tablespace using free lists or bitmaps. This clause is not valid for a temporary tablespace.

AUTO

Specify `AUTO` if you want the database to manage the free space of segments in the tablespace using a bitmap. If you specify `AUTO`, then the database ignores any specification for `PCTUSED`, `FREELIST`, and `FREELIST GROUPS` in subsequent storage specifications for objects in this tablespace. This setting is called **automatic segment-space management** and is the default.

MANUAL

Specify `MANUAL` if you want the database to manage the free space of segments in the tablespace using free lists. Oracle strongly recommends that you do not use this setting and that you create tablespaces with automatic segment-space management.

To determine the segment management of an existing tablespace, query the `SEGMENT_SPACE_MANAGEMENT` column of the `DBA_TABLESPACES` or `USER_TABLESPACES` data dictionary view.

Note:

If you specify `AUTO` segment management, then:

- If you set extent management to `LOCAL UNIFORM`, then you must ensure that each extent contains at least 5 database blocks.
- If you set extent management to `LOCAL AUTOALLOCATE`, and if the database block size is 16K or greater, then Oracle manages segment space by creating extents with a minimum size of 5 blocks rounded up to 64K.

Restrictions on Automatic Segment-Space Management

This clause is subject to the following restrictions:

- You can specify this clause only for a permanent, locally managed tablespace.
- You cannot specify this clause for the `SYSTEM` tablespace.

> **See Also:**
>
> - *Oracle Automatic Storage Management Administrator's Guide* for information on automatic segment-space management and when to use it
> - *Oracle Database Reference* for information on the data dictionary views
> - "Specifying Segment Space Management for a Tablespace: Example"

flashback_mode_clause

Use this clause in conjunction with the `ALTER DATABASE FLASHBACK` clause to specify whether the tablespace can participate in `FLASHBACK DATABASE` operations. This clause is useful if you have the database in `FLASHBACK` mode but you do not want Oracle Database to maintain Flashback log data for this tablespace.

This clause is not valid for temporary or undo tablespaces.

FLASHBACK ON

Specify `FLASHBACK ON` to put the tablespace in `FLASHBACK` mode. Oracle Database will save Flashback log data for this tablespace and the tablespace can participate in a `FLASHBACK DATABASE` operation. If you omit the *`flashback_mode_clause`*, then `FLASHBACK ON` is the default.

FLASHBACK OFF

Specify `FLASHBACK OFF` to take the tablespace out of `FLASHBACK` mode. Oracle Database will not save any Flashback log data for this tablespace. You must take the data files in this tablespace offline or drop them prior to any subsequent `FLASHBACK DATABASE` operation. Alternatively, you can take the entire tablespace offline. In either case, the database does not drop existing Flashback logs.

> **Note:**
>
> The `FLASHBACK` mode of a tablespace is independent of the `FLASHBACK` mode of an individual table.

> **See Also:**
>
> - *Oracle Database Backup and Recovery User's Guide* for information on Oracle Flashback Database
> - ALTER DATABASE and FLASHBACK DATABASE for information on setting the `FLASHBACK` mode of the entire database and reverting the database to an earlier version
> - FLASHBACK TABLE and *flashback_query_clause*

undo_tablespace_clause

Specify UNDO to create an undo tablespace. When you run the database in automatic undo management mode, Oracle Database manages undo space using the undo tablespace instead of rollback segments. This clause is useful if you are now running in automatic undo management mode but your database was not created in automatic undo management mode.

Oracle Database always assigns an undo tablespace when you start up the database in automatic undo management mode. If no undo tablespace has been assigned to this instance, then the database uses the SYSTEM rollback segment. You can avoid this by creating an undo tablespace, which the database will implicitly assign to the instance if no other undo tablespace is currently assigned.

The DATAFILE clause is described in "DATAFILE | TEMPFILE Clause".

extent_management_clause

It is unnecessary to specify the *extent_management_clause* when creating an undo tablespace, because undo tablespaces must be locally managed tablespaces that use AUTOALLOCATE extent management. If you do specify this clause, then you must specify EXTENT MANAGEMENT LOCAL or EXTENT MANAGEMENT LOCAL AUTOALLOCATE, both of which are the same as omitting this clause. Refer to *extent_management_clause* for the full semantics of this clause.

tablespace_retention_clause

This clause is valid only for undo tablespaces.

- RETENTION GUARANTEE specifies that Oracle Database should preserve unexpired undo data in all undo segments of *tablespace* even if doing so forces the failure of ongoing operations that need undo space in those segments. This setting is useful if you need to issue an Oracle Flashback Query or an Oracle Flashback Transaction Query to diagnose and correct a problem with the data.
- RETENTION NOGUARANTEE returns the undo behavior to normal. Space occupied by unexpired undo data in undo segments can be consumed if necessary by ongoing transactions. This is the default.

tablespace_encryption_clause

This clause has the same semantics for undo tablespaces as for permanent tablespaces. Refer to *tablespace_encryption_clause* in the documentation on permanent tablespaces for full information.

Restrictions on Undo Tablespaces

Undo tablespaces are subject to the following restrictions:

- You cannot create database objects in this tablespace. It is reserved for system-managed undo data.
- The only clauses you can specify for an undo tablespace are the DATAFILE clause, the *tablespace_retention_clause*, the *tablespace_encryption_clause*, and the *extent_management_clause* to specify local AUTOALLOCATE extent management. You cannot specify local UNIFORM extent management or dictionary extent management using the *extent_management_clause*. All undo tablespaces are created permanent, read/write,

and in logging mode. Values for MINIMUM EXTENT and DEFAULT STORAGE are system generated.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on automatic undo management and undo tablespaces and *Oracle Database Reference* for information on the UNDO_MANAGEMENT parameter
> - CREATE DATABASE for information on creating an undo tablespace during database creation, and ALTER TABLESPACE and DROP TABLESPACE
> - "Creating an Undo Tablespace: Example"

temporary_tablespace_clause

Use this clause to create a temporary tablespace, which is an allocation of space in the database that can contain transient data that persists only for the duration of a session. This transient data cannot be recovered after process or instance failure.

The transient data can be user-generated schema objects such as temporary tables or system-generated data such as temp space used by hash joins and sort operations. When a temporary tablespace, or a tablespace group of which this tablespace is a member, is assigned to a particular user, then Oracle Database uses the tablespace for sorting operations in transactions initiated by that user.

You can create two types of temporary tablespaces:

- You can create a shared temporary tablespace by specifying the TEMPORARY TABLESPACE clause. A shared temporary tablespace stores temp files on shared disk, so that the temporary space is accessible to all database instances. Shared temporary tablespaces were available in prior releases of Oracle Database and were called "temporary tablespaces." Elsewhere in this guide, the term "temporary tablespace" refers to a shared temporary tablespace unless specified otherwise.
- Starting with Oracle Database 12*c* Release 2 (12.2), you can create a local temporary tablespace by specifying the LOCAL TEMPORARY TABLESPACE clause. Local temporary tablespaces are useful in an Oracle Clusterware environment. They store a separate, nonshared temp files for each database instance, which can improve I/O performance. A local temporary tablespace must be a BIGFILE tablespace.
  - Specify FOR ALL to instruct the database to create separate, nonshared temp files for all HUB and LEAF nodes.
  - Specify FOR LEAF to instruct the database to create separate nonshared temp files for only LEAF nodes.

TEMPFILE

The TEMPFILE clause is described in "DATAFILE | TEMPFILE Clause".

tablespace_group_clause

This clause is relevant only for temporary tablespaces. Use this clause to determine whether *tablespace* is a member of a tablespace group. A tablespace group lets you assign multiple temporary tablespaces to a single user and increases the addressability of temporary tablespaces.

- Specify a group name to indicate that *tablespace* is a member of this tablespace group. The group name cannot be the same as *tablespace* or any other existing tablespace. If the tablespace group already exists, then Oracle Database adds the new tablespace to that group. If the tablespace group does not exist, then the database creates the group and adds the new tablespace to that group.
- Specify an empty string (' ') to indicate that *tablespace* is not a member of any tablespace group.

Restriction on Tablespace Groups

Tablespace groups support only shared temporary tablespaces. You cannot add a local temporary tablespace to a tablespace group.

extent_management_clause

The *extent_management_clause* is described in *extent_management_clause* .

tablespace_encryption_clause

This clause has the same semantics for temporary tablespaces as for permanent tablespaces. Refer to *tablespace_encryption_clause* in the documentation on permanent tablespaces for full information.

> **See Also:**
>
> - ALTER TABLESPACE and "Adding a Temporary Tablespace to a Tablespace Group: Example" for information on adding a tablespace to a tablespace group
> - CREATE USER for information on assigning a temporary tablespace to a user
> - *Oracle Database Administrator's Guide* for more information on tablespace groups

Restrictions on Temporary Tablespaces

The data stored in temporary tablespaces persists only for the duration of a session. Therefore, only a subset of the CREATE TABLESPACE clauses are relevant for temporary tablespaces. The only clauses you can specify for a temporary tablespace are the TEMPFILE clause, the *tablespace_group_clause*, the *extent_management_clause*, and the *tablespace_encryption_clause*.

Examples

These examples assume that your database is using 8K blocks.

Creating a Bigfile Tablespace: Example

The following example creates a bigfile tablespace bigtbs_01 with a data file bigtbs_f1.dbf of 20 MB:

```
CREATE BIGFILE TABLESPACE bigtbs_01
  DATAFILE 'bigtbs_f1.dbf'
  SIZE 20M AUTOEXTEND ON;
```

Creating an Undo Tablespace: Example

The following example creates a 10 MB undo tablespace undots1:

```
CREATE UNDO TABLESPACE undots1
   DATAFILE 'undotbs_1a.dbf'
   SIZE 10M AUTOEXTEND ON
   RETENTION GUARANTEE;
```

Creating a Temporary Tablespace: Example

This statement shows how the temporary tablespace that serves as the default temporary tablespace for database users in the sample database was created:

```
CREATE TEMPORARY TABLESPACE temp_demo
   TEMPFILE 'temp01.dbf' SIZE 5M AUTOEXTEND ON;
```

Assuming that the default database block size is 2K, and that each bit in the map represents one extent, then each bit maps 2,500 blocks.

The following example sets the default location for data file creation and then creates a tablespace with an Oracle-managed temp file in the default location. The temp file is 100 M and is autoextensible with unlimited maximum size. These are the default values for Oracle Managed Files:

```
ALTER SYSTEM SET DB_CREATE_FILE_DEST = '$ORACLE_HOME/rdbms/dbs';

CREATE TEMPORARY TABLESPACE tbs_05;
```

Adding a Temporary Tablespace to a Tablespace Group: Example

The following statement creates the tbs_temp_02 temporary tablespace as a member of the tbs_grp_01 tablespace group. If the tablespace group does not already exist, then Oracle Database creates it during execution of this statement:

```
CREATE TEMPORARY TABLESPACE tbs_temp_02
  TEMPFILE 'temp02.dbf' SIZE 5M AUTOEXTEND ON
  TABLESPACE GROUP tbs_grp_01;
```

Creating Basic Tablespaces: Examples

This statement creates a tablespace named tbs_01 with one data file:

```
CREATE TABLESPACE tbs_01
   DATAFILE 'tbs_f2.dbf' SIZE 40M
   ONLINE;
```

This statement creates tablespace tbs_03 with one data file and allocates every extent as a multiple of 500K:

```
CREATE TABLESPACE tbs_03
   DATAFILE 'tbs_f03.dbf' SIZE 20M
   LOGGING;
```

Enabling Autoextend for a Tablespace: Example

This statement creates a tablespace named tbs_02 with one data file. When more space is required, 500 kilobyte extents will be added up to a maximum size of 100 megabytes:

```
CREATE TABLESPACE tbs_02
   DATAFILE 'diskb:tbs_f5.dbf' SIZE 500K REUSE
   AUTOEXTEND ON NEXT 500K MAXSIZE 100M;
```

Creating a Locally Managed Tablespace: Example

The following statement assumes that the database block size is 2K.

```
CREATE TABLESPACE tbs_04 DATAFILE 'file_1.dbf' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128K;
```

This statement creates a locally managed tablespace in which every extent is 128K and each bit in the bit map describes 64 blocks.

The following statement creates a locally managed tablespace with uniform extents and shows an example of a table stored in that tablespace:

```
CREATE TABLESPACE lmt1 DATAFILE 'lmt_file2.dbf' SIZE 100m REUSE
  EXTENT MANAGEMENT LOCAL UNIFORM SIZE 1M;

CREATE TABLE lmt_table1 (col1 NUMBER, col2 VARCHAR2(20))
  TABLESPACE lmt1 STORAGE (INITIAL 2m);
```

The initial segment size of the table is 2M.

The following example creates a locally managed tablespace without uniform extents:

```
CREATE TABLESPACE lmt2 DATAFILE 'lmt_file3.dbf' SIZE 100m REUSE
  EXTENT MANAGEMENT LOCAL;

CREATE TABLE lmt_table2 (col1 NUMBER, col2 VARCHAR2(20))
  TABLESPACE lmt2 STORAGE (INITIAL 2m MAXSIZE 100m);
```

The initial segment size of the table is 2M. Oracle Database determines the size of each extent and the total number of extents allocated to satisfy the initial segment size. The segment's maximum size is limited to 100M.

Creating an Encrypted Tablespace: Example

In the following example, the first statement enables encryption for the database by opening the wallet. The second statement creates an encrypted tablespace.

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN IDENTIFIED BY "wallet_password";

CREATE TABLESPACE encrypt_ts
  DATAFILE '$ORACLE_HOME/dbs/encrypt_df.dbf' SIZE 1M
  ENCRYPTION USING 'AES256' ENCRYPT;
```

Specifying Segment Space Management for a Tablespace: Example

The following example creates a tablespace with automatic segment-space management:

```
CREATE TABLESPACE auto_seg_ts DATAFILE 'file_2.dbf' SIZE 1M
   EXTENT MANAGEMENT LOCAL
   SEGMENT SPACE MANAGEMENT AUTO;
```

Creating Oracle Managed Files: Examples

The following example sets the default location for data file creation and creates a tablespace with a data file in the default location. The data file is 100M and is autoextensible with an unlimited maximum size:

```
ALTER SYSTEM SET DB_CREATE_FILE_DEST = '$ORACLE_HOME/rdbms/dbs';

CREATE TABLESPACE omf_ts1;
```

The following example creates a tablespace with an Oracle-managed data file of 100M that is not autoextensible:

```
CREATE TABLESPACE omf_ts2 DATAFILE AUTOEXTEND OFF;
```

CREATE TABLESPACE SET

> **Note:**
>
> This SQL statement is valid only if you are using Oracle Sharding. For more information on Oracle Sharding, refer to *Oracle Database Administrator's Guide*.

Purpose

Use the `CREATE TABLESPACE SET` statement to create a tablespace set. A tablespace set can be used in a sharded database as a logical storage unit for one or more sharded tables and indexes.

A tablespace set consists of multiple tablespaces distributed across shards in a shardspace. The database automatically creates the tablespaces in a tablespace set. The number of tablespaces is determined automatically and is equal to the number of chunks in the corresponding shardspace.

All tablespaces in a tablespace set are permanent bigfile tablespaces; a tablespace set does not contain `SYSTEM`, undo, or temporary tablespaces. The database automatically creates one data file for each tablespace. All tablespaces in a tablespace set share the same attributes. You can modify attributes for all tablespaces in a tablespace set with the `ALTER TABLESPACE SET` statement.

> **See Also:**
>
> ALTER TABLESPACE SET and DROP TABLESPACE SET

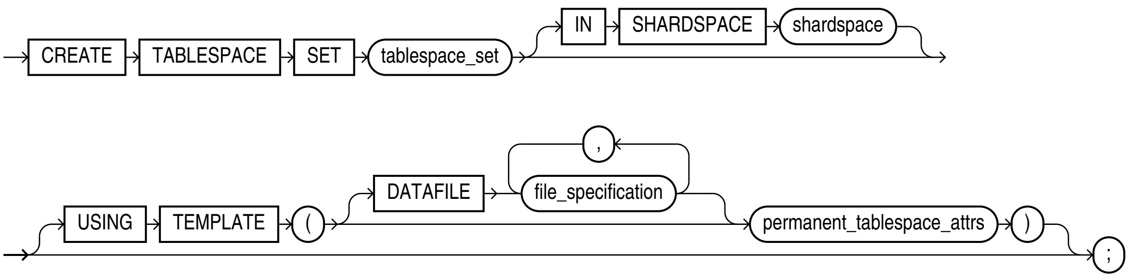
Prerequisites

You must be connected to a shard catalog database as an SDB user.

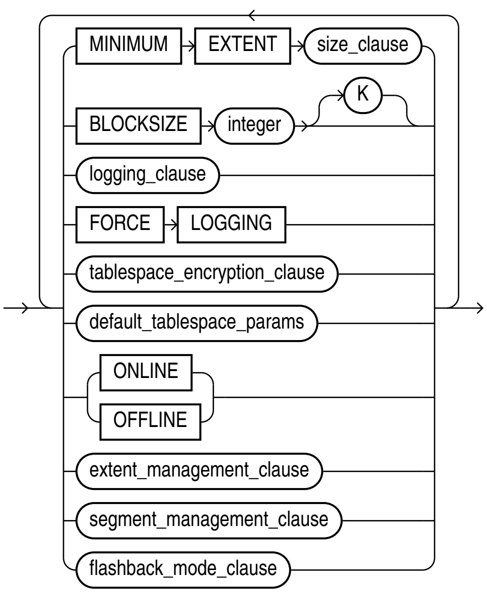
You must have the `CREATE TABLESPACE` system privilege.

Syntax

***create_tablespace_set*::=**



***permanent_tablespace_attrs*::=**



(*file_specification*::=, See the following clauses of `CREATE TABLESPACE`: *logging_clause*::=, *tablespace_encryption_clause*::=, *default_tablespace_params*::=, *extent_management_clause*::=, *segment_management_clause*::=, *flashback_mode_clause*::=)

Semantics

tablespace_set

Specify the name of the tablespace set to be created. The name must satisfy the requirements listed in Database Object Naming Rules .

IN SHARDSPACE

Specify this clause if you are using composite sharding. For *`shardspace_name`*, specify the name of the shardspace in which the tablespace set is to be created.

Omit this clause if you are using system-managed sharding. In this case, the tablespace set is created in the default shardspace for the sharded database.

USING TEMPLATE

The `USING TEMPLATE` clause allows you to specify attributes for the tablespaces in the tablespace set.

The `DATAFILE` and *`permanent_tablespace_attrs`* clauses have the same semantics here as for the `CREATE TABLESPACE` statement, with the following exceptions:

- For the `DATAFILE` *`file_specification`* clause, you can specify only the `SIZE` clause and the *`autoextend_clause`*.
- You cannot specify the `MINIMUM EXTENT` *`size_clause`*.
- For the *`segment_management_clause`*, you can specify only `SEGMENT SPACE MANAGEMENT AUTO`. The `MANUAL` setting is not supported.

> **See Also:**
>
> *file_specification* and *permanent_tablespace_attrs* in the documentation on `CREATE TABLESPACE` for the full semantics of these clauses

Examples

Creating a Tablespace Set: Example

The following statement creates tablespace set `ts1`:

```
CREATE TABLESPACE SET ts1
  IN SHARDSPACE sgr1
  USING TEMPLATE
  ( DATAFILE SIZE 100m
    EXTENT MANAGEMENT LOCAL
    SEGMENT SPACE MANAGEMENT AUTO
  );
```

CREATE TRIGGER

Purpose

Triggers are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the `CREATE TRIGGER` statement to create a **database trigger**, which is:

- A stored PL/SQL block associated with a table, a schema, or the database or
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

Oracle Database automatically executes a trigger when specified conditions occur.

See Also:

ALTER TRIGGER and DROP TRIGGER

Prerequisites

To create a trigger in your own schema on a table in your own schema or on your own schema (`SCHEMA`), you must have the `CREATE TRIGGER` system privilege.

To create a trigger in any schema on a table in any schema, or on another user's schema (*`schema`*.`SCHEMA`), you must have the `CREATE ANY TRIGGER` system privilege.

In addition to the preceding privileges, to create a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege.

To create a trigger on a pluggable database (PDB), the current container must be that PDB and you must have the `ADMINISTER DATABASE TRIGGER` system privilege. For information about PDBs, see *Oracle Database Administrator's Guide*.

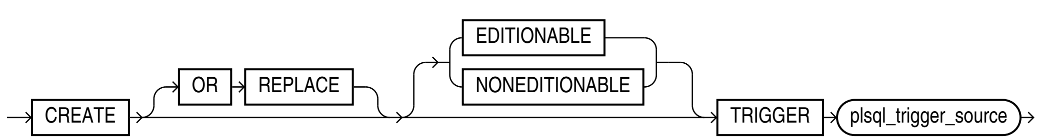
In addition to the preceding privileges, to create a crossedition trigger, you must be enabled for editions. For information about enabling editions for a user, see *Oracle Database Development Guide*.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.

Syntax

Triggers are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

create_trigger::=



(*`plsql_trigger_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the trigger if it already exists. Use this clause to change the definition of an existing trigger without first dropping it.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the trigger is an editioned or noneditioned object if editioning is enabled for the schema object type `TRIGGER` in *`schema`*. The default is

EDITIONABLE. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

Restriction on NONEDITIONABLE

You cannot specify NONEDITIONABLE for a crossedition trigger.

plsql_trigger_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *plsql_trigger_source*.

CREATE TYPE

Purpose

Object types are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the CREATE TYPE statement to create the specification of an **object type**, a **SQLJ object type**, a named varying array (**varray**), a **nested table type**, or an **incomplete object type**. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods that implement the type.

> **Note:**
>
> - If you create an object type for which the type specification declares only attributes but no methods, then you need not specify a type body.
> - If you create a SQLJ object type, then you cannot specify a type body. The implementation of the type is specified as a Java class.

An **incomplete type** is a type created by a forward type definition. It is called "incomplete" because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

> **See Also:**
>
> - CREATE TYPE BODY for information on creating the member methods of a type
> - *Oracle Database Object-Relational Developer's Guide* for more information about objects, incomplete types, varrays, and nested tables

Prerequisites

To create a type in your own schema, you must have the `CREATE TYPE` system privilege. To create a type in another user's schema, you must have the `CREATE ANY TYPE` system privilege. You can acquire these privileges explicitly or be granted them through a role.

To create a subtype, you must have the `UNDER ANY TYPE` system privilege or the `UNDER` object privilege on the supertype.

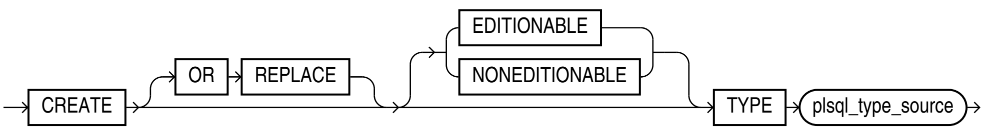
The owner of the type must be explicitly granted the `EXECUTE` object privilege in order to access all other types referenced within the definition of the type, or the type owner must be granted the `EXECUTE ANY TYPE` system privilege. The owner cannot obtain these privileges through roles.

If the type owner intends to grant other users access to the type, then the owner must be granted the `EXECUTE` object privilege on the referenced types with the `GRANT OPTION` or the `EXECUTE ANY TYPE` system privilege with the `ADMIN OPTION`. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

Syntax

Types are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_type*::=**



(*`plsql_type_source`*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the type if it already exists. Use this clause to change the definition of an existing type without first dropping it.

Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again.

If any function-based indexes depend on the type, then Oracle Database marks the indexes `DISABLED`.

[EDITIONABLE | NONEDITIONABLE]

Use these clauses to specify whether the type is an editioned or noneditioned object if editioning is enabled for the schema object type `TYPE` in *`schema`*.The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

plsql_type_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *plsql_type_source*.

CREATE TYPE BODY

Purpose

Type bodies are defined using PL/SQL. Therefore, this section provides some general information but refers to *Oracle Database PL/SQL Language Reference* for details of syntax and semantics.

Use the CREATE TYPE BODY to define or implement the member methods defined in the object type specification. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods that implement the type.

For each method specified in an object type specification for which you did not specify the *call_spec*, you must specify a corresponding method body in the object type body.

> **Note:**
>
> If you create a SQLJ object type, then specify it as a Java class.

> **See Also:**
>
> - CREATE TYPE for information on creating a type specification
> - ALTER TYPE for information on modifying a type specification

Prerequisites

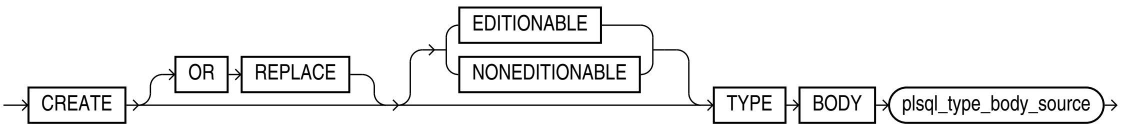
Every member declaration in the CREATE TYPE specification for object types must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create an object type in another user's schema, you must have the CREATE ANY TYPE system privilege. To replace an object type in another user's schema, you must have the DROP ANY TYPE system privilege.

Syntax

Type bodies are defined using PL/SQL. Therefore, the syntax diagram in this book shows only the SQL keywords. Refer to *Oracle Database PL/SQL Language Reference* for the PL/SQL syntax, semantics, and examples.

***create_type_body*::=**



(*plsql_type_body_source*: See *Oracle Database PL/SQL Language Reference*.)

Semantics

OR REPLACE

Specify OR REPLACE to re-create the type body if it already exists. Use this clause to change the definition of an existing type body without first dropping it.

Users previously granted privileges on the re-created object type body can use and reference the object type body without being granted privileges again.

You can use this clause to add new member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE statement.

[EDITIONABLE | NONEDITIONABLE]

If you do not specify this clause, then the type body inherits EDITIONABLE or NONEDITIONABLE from the type specification. If you do specify this clause, then it must match that of the type specification.

plsql_type_body_source

See *Oracle Database PL/SQL Language Reference* for the syntax and semantics of the *plsql_type_body_source*.

CREATE USER

Purpose

Use the CREATE USER statement to create and configure a database **user**, which is an account through which you can log in to the database, and to establish the means by which Oracle Database permits access by the user.

You can issue this statement in an Oracle Automatic Storage Management (Oracle ASM) cluster to add a user and password combination to the password file that is local to the Oracle ASM instance of the current node. Each node's Oracle ASM instance can use this statement to update its own password file. The password file itself must have been created by the ORAPWD utility.

You can enable a user to connect to the database through a proxy application or application server. For syntax and discussion, refer to ALTER USER .

Prerequisites

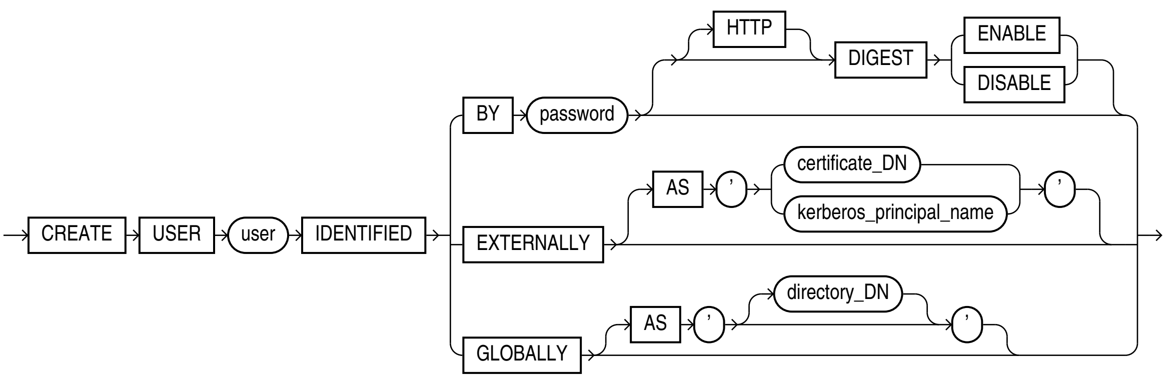
You must have the CREATE USER system privilege. When you create a user with the CREATE USER statement, the user's privilege domain is empty. To log on to Oracle Database, a user must have the `CREATE SESSION` system privilege. Therefore, after creating a user, you should grant the user at least the `CREATE SESSION` system privilege. Refer to GRANT for more information.

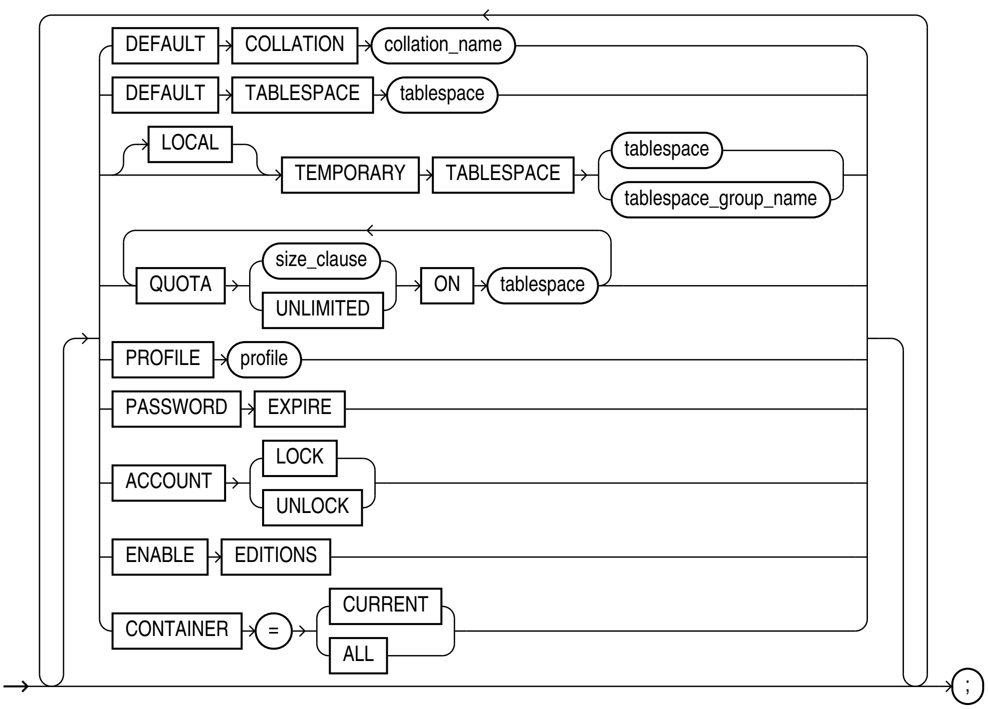
Only a user authenticated `AS SYSASM` can issue this command to modify the Oracle ASM instance password file.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB).

Syntax

***create_user*::=**





(*size_clause*::=)

Semantics

user

Specify the name of the user to be created. This name can contain only characters from your database character set and must follow the rules described in the section "Database Object Naming Rules ". Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

In a non-CDB, a user name cannot begin with `C##` or `c##`.

In a CDB, the requirements for a user name are as follows:

- The name of a **common user** must begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. By default, the prefix is `C##`.
- The name of a **local user** must not begin with characters that are a case-insensitive match to the prefix specified by the `COMMON_USER_PREFIX` initialization parameter. Regardless of the value of `COMMON_USER_PREFIX`, the name of a local user can never begin with `C##` or `c##`.

> **Note:**
>
> If the value of `COMMON_USER_PREFIX` is an empty string, then there are no requirements for common or local user names with one exception: the name of a local user can never begin with `C##` or `c##`. Oracle recommends against using an empty string value because it might result in conflicts between the names of local and common users when a PDB is plugged into a different CDB, or when opening a PDB that was closed when a common user was created.

> **Note:**
>
> Oracle recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform.

> **See Also:**
>
> "Creating a Database User: Example"

IDENTIFIED Clause

The `IDENTIFIED` clause lets you indicate how Oracle Database authenticates the user.

BY *password*

The `BY` *`password`* clause lets you creates a **local user** and indicates that the user must specify *`password`* to log on to the database. Passwords are case sensitive. Any subsequent `CONNECT` string used to connect this user to the database must specify the password using the same case (upper, lower, or mixed) that is used in this `CREATE USER` statement or a subsequent `ALTER USER` statement. Passwords can contain any single-byte, multibyte, or special characters, or any combination of these, from your database character set, with the exception of the double quotation mark (") and the return character. If a password starts with a non-alphabetic character, or contains a character other than an alphanumeric character, the underscore (_), dollar sign ($), or pound sign (#), then it must be enclosed in double quotation marks. Otherwise, enclosing a password in double quotation marks is optional.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about case-sensitive passwords, password complexity, and other password guidelines

Passwords must follow the rules described in the section "Database Object Naming Rules ", unless you are using one of the three Oracle Database password complexity verification routines. These routines requires a more complex combination of characters than the normal naming rules permit. You implement these routines with the `UTLPWDMG.SQL` script, which is further described in *Oracle Database Security Guide*.

> **Note:**
>
> Oracle recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform.

> **See Also:**
>
> *Oracle Database Security Guide* to for a detailed discussion of password management and protection

[HTTP] DIGEST Clause

This clause lets you `ENABLE` or `DISABLE` HTTP Digest Access Authentication for the user. The default is `DISABLE`.

The `HTTP` keyword is optional and is provided for semantic clarity.

Restriction on the [HTTP] DIGEST Clause

You cannot specify this clause for external or global users.

EXTERNALLY Clause

Specify `EXTERNALLY` to create an **external user.** Such a user must be authenticated by an external service, such as an operating system or a third-party service. In this case, Oracle Database relies on authentication by the operating system or third-party service to ensure that a specific external user has access to a specific database user.

AS '*certificate_DN*'

This clause is required for and used for SSL-authenticated external users only. The *`certificate_DN`* is the distinguished name in the user's PKI certificate in the user's wallet. The maximum length of *`certificate_DN`* is 1024 characters.

AS '*kerberos_principal_name*'

This clause is required for and used for Kerberos-authenticated external users only. The maximum length of *`kerberos_principal_name`* is 1024 characters.

> **Note:**
>
> Oracle strongly recommends that you do not use `IDENTIFIED EXTERNALLY` with operating systems that have inherently weak login security.

Restriction on Creating External Users

Oracle ASM does not support the creation of external users.

> **See Also:**
>
> - *Oracle Database Enterprise User Security Administrator's Guide* for more information on externally identified users
> - "Creating External Database Users: Examples"

GLOBALLY Clause

The `GLOBALLY` clause lets you create a **global user**. Such a user must be authorized by the enterprise directory service (Oracle Internet Directory).

The *`directory_DN`* string can take one of two forms:

- The X.509 name at the enterprise directory service that identifies this user. It should be of the form `CN=`*`username,other_attributes`*, where *`other_attributes`* is the rest of the user's distinguished name (DN) in the directory. This form creates a **private global schema**.
- A null string (' ') indicating that the enterprise directory service will map authenticated global users to this database schema with the appropriate roles. This form is the same as specifying the `GLOBALLY` keyword alone and creates a **shared global schema**.

The maximum length of *`directory_DN`* is 1024 characters.

You can control the ability of an application server to connect as the specified user and to activate that user's roles using the ALTER USER statement.

Restriction on Creating Global Users

Oracle ASM does not support the creation of global users.

> **See Also:**
>
> - *Oracle Database Security Guide* for more information on global users
> - ALTER USER
> - "Creating a Global Database User: Example"

DEFAULT COLLATION Clause

This clause lets you specify the default collation for the schema owned by the user. The default collation is assigned to tables, views, and materialized views that are subsequently created in the schema.

For *collation_name*, specify a valid named collation or pseudo-collation.

If you omit this clause, then the default collation for the schema owned by the user is set to the USING_NLS_COMP pseudo-collation.

You can override this clause and assign a different default collation to a particular table, materialized view, or view by specifying the DEFAULT COLLATION clause of the CREATE or ALTER statement for the table, materialized view, or view. You can also override the default collations of all schemas for the duration of a database session by setting the default collation for the session. See the DEFAULT_COLLATION clause of ALTER SESSION for more details.

You can specify the DEFAULT COLLATION clause only if the COMPATIBLE initialization parameter is set to 12.2 or greater, and the MAX_STRING_SIZE initialization parameter is set to EXTENDED.

DEFAULT TABLESPACE Clause

Specify the default tablespace for objects that are created in the user's schema. If you omit this clause, then the user's objects are stored in the database default tablespace. If no default tablespace has been specified for the database, then the user's objects are stored in the SYSTEM tablespace.

Restriction on Default Tablespaces

You cannot specify a locally managed temporary tablespace, including an undo tablespace, or a dictionary-managed temporary tablespace, as a user's default tablespace.

> **See Also:**
>
> - CREATE TABLESPACE for more information on tablespaces in general and undo tablespaces in particular
> - *Oracle Database Security Guide* for more information on assigning default tablespaces to users

[LOCAL] TEMPORARY TABLESPACE Clause

Specify the tablespace or tablespace group for the user's temporary segments. If you omit this clause, then the user's temporary segments are stored in the database default temporary tablespace or, if none has been specified, in the `SYSTEM` tablespace.

- Specify *`tablespace`* to indicate the user's temporary tablespace. Specify `TEMPORARY TABLESPACE` to indicate a shared temporary tablespace. Specify `LOCAL TEMPORARY TABLESPACE` to indicate a local temporary tablespace. If you are connected to a CDB, then you can specify `CDB$DEFAULT` to use the CDB-wide default temporary tablespace.
- Specify *`tablespace_group_name`* to indicate that the user can save temporary segments in any tablespace in the tablespace group specified by *`tablespace_group_name`*. Local temporary tablespaces cannot be part of a tablespace group.

Restrictions on Temporary Tablespace

This clause is subject to the following restrictions:

- The tablespace must be a temporary tablespace and must have a standard block size.
- The tablespace cannot be an undo tablespace or a tablespace with automatic segment-space management.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information about tablespace groups and *Oracle Database Security Guide* for information on assigning temporary tablespaces to users
> - CREATE TABLESPACE for more information on undo tablespaces and segment management
> - "Assigning a Tablespace Group: Example"

QUOTA Clause

Use the `QUOTA` clause to specify the maximum amount of space the user can allocate in the tablespace.

A `CREATE USER` statement can have multiple `QUOTA` clauses for multiple tablespaces.

`UNLIMITED` lets the user allocate space in the tablespace without bound.

The maximum amount of space that you can specify is 2 terabytes (TB). If you need more space, then specify `UNLIMITED`.

Restriction on the QUOTA Clause

You cannot specify this clause for a temporary tablespace.

> **See Also:**
>
> *size_clause* for information on that clause and *Oracle Database Security Guide* for more information on assigning tablespace quotas

PROFILE Clause

Specify the profile you want to assign to the user. The profile limits the amount of database resources the user can use. If you omit this clause, then Oracle Database assigns the `DEFAULT` profile to the user.

> **Note:**
>
> Oracle recommends that you use the Database Resource Manager rather SQL profiles to establish database resource limits. The Database Resource Manager offers a more flexible means of managing and tracking resource use. For more information on the Database Resource Manager, refer to *Oracle Database Administrator's Guide.*

> **See Also:**
>
> GRANT and CREATE PROFILE

PASSWORD EXPIRE Clause

Specify `PASSWORD EXPIRE` if you want the user's password to expire. This setting forces the user or the DBA to change the password before the user can log in to the database.

ACCOUNT Clause

Specify `ACCOUNT LOCK` to lock the user's account and disable access. Specify `ACCOUNT UNLOCK` to unlock the user's account and enable access to the account. The default is `ACCOUNT UNLOCK`.

ENABLE EDITIONS

This clause is not reversible. Specify `ENABLE EDITIONS` to allow the user to create multiple versions of editionable objects in this schema using editions. Editionable objects in schemas that are not editions-enabled cannot be editioned.

Restriction on Enabling Editions

You cannot enable editions for any schemas supplied by Oracle except for the sample schemas in the seed database.

CONTAINER Clause

The `CONTAINER` clause applies when you are connected to a CDB. However, it is not necessary to specify the `CONTAINER` clause because its default values are the only allowed values.

- To create a common user, you must be connected to the root. You can optionally specify `CONTAINER = ALL`, which is the default when you are connected to the root.
- To create a local user, you must be connected to a PDB. You can optionally specify `CONTAINER = CURRENT`, which is the default when you are connected to a PDB.

While creating a common user, any default tablespace, temporary tablespace, or profile specified using the following clauses must exist in all the containers belonging to the CDB:

- `DEFAULT TABLESPACE`
- `TEMPORARY TABLESPACE`
- `QUOTA`
- `PROFILE`

If these objects do not exist in all the containers, the `CREATE USER` statement fails.

Examples

All of the following examples use the `example` tablespace, which exists in the seed database and is accessible to the sample schemas.

Creating a Database User: Example

If you create a new user with `PASSWORD EXPIRE`, then the user's password must be changed before the user attempts to log in to the database. You can create the user `sidney` by issuing the following statement:

```
CREATE USER sidney
    IDENTIFIED BY out_standing1
    DEFAULT TABLESPACE example
    QUOTA 10M ON example
    TEMPORARY TABLESPACE temp
    QUOTA 5M ON system
    PROFILE app_user
    PASSWORD EXPIRE;
```

The user `sidney` has the following characteristics:

- The password `out_standing1`
- Default tablespace `example`, with a quota of 10 megabytes
- Temporary tablespace `temp`
- Access to the tablespace `SYSTEM`, with a quota of 5 megabytes
- Limits on database resources defined by the profile `app_user` (which was created in "Creating a Profile: Example")
- An expired password, which must be changed before `sidney` can log in to the database

Creating External Database Users: Examples

The following example creates an external user, who must be identified by an external source before accessing the database:

```
CREATE USER app_user1
   IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE example
   QUOTA 5M ON example
   PROFILE app_user;
```

The user `app_user1` has the following additional characteristics:

- Default tablespace `example`
- Default temporary tablespace `example`
- 5M of space on the tablespace `example` and unlimited quota on the temporary tablespace of the database
- Limits on database resources defined by the `app_user` profile

To create another user accessible only by an operating system account, prefix the user name with the value of the initialization parameter `OS_AUTHENT_PREFIX`. For example, if this value is "`ops$`", then you can create the externally identified user `external_user` with the following statement:

```
CREATE USER ops$external_user
   IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE example
   QUOTA 5M ON example
   PROFILE app_user;
```

Creating a Global Database User: Example

The following example creates a global user. When you create a global user, you can specify the X.509 name that identifies this user at the enterprise directory server:

```
CREATE USER global_user
   IDENTIFIED GLOBALLY AS 'CN=analyst, OU=division1, O=oracle, C=US'
   DEFAULT TABLESPACE example
   QUOTA 5M ON example;
```

Creating a Common User in a CDB

The following example creates a common user called c##`comm_user` in a CDB. Before you run this `CREATE USER` statement, ensure that the tablespaces `example` and `temp_tbs` exist in all of the containers in the CDB.

```
CREATE USER c##comm_user
   IDENTIFIED BY comm_pwd
   DEFAULT TABLESPACE example
   QUOTA 20M ON example
   TEMPORARY TABLESPACE temp_tbs;
```

The user `comm_user` has the following additional characteristics:

- The password `comm_pwd`
- Default tablespace `example`, with a quota of 20 megabytes
- Temporary tablespace `temp_tbs`

CREATE VIEW

Purpose

Use the `CREATE VIEW` statement to define a **view**, which is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called **base tables**.

You can also create an **object view** or a relational view that supports LOBs, object types, `REF` data types, nested table, or varray types on top of the existing view mechanism. An object view is a view of a user-defined type, where each row contains objects, each object with a unique object identifier.

You can also create `XMLType` views, which are similar to object views but display data from XMLSchema-based tables of `XMLType`.

> **See Also:**
>
> - *Oracle Database Concepts*, *Oracle Database Development Guide*, and *Oracle Database Administrator's Guide* for information on various types of views and their uses
> - *Oracle XML DB Developer's Guide* for information on `XMLType` views
> - ALTER VIEW and DROP VIEW for information on modifying a view and removing a view from the database

Prerequisites

To create a view in your own schema, you must have the `CREATE VIEW` system privilege. To create a view in another user's schema, you must have the `CREATE ANY VIEW` system privilege.

To create a subview, you must have the `UNDER ANY VIEW` system privilege or the `UNDER` object privilege on the superview.

The owner of the schema containing the view must have the privileges necessary to either select (`READ` or `SELECT` privilege), insert, update, or delete rows from all the tables or views on which the view is based. The owner must be granted these privileges directly, rather than through a role.

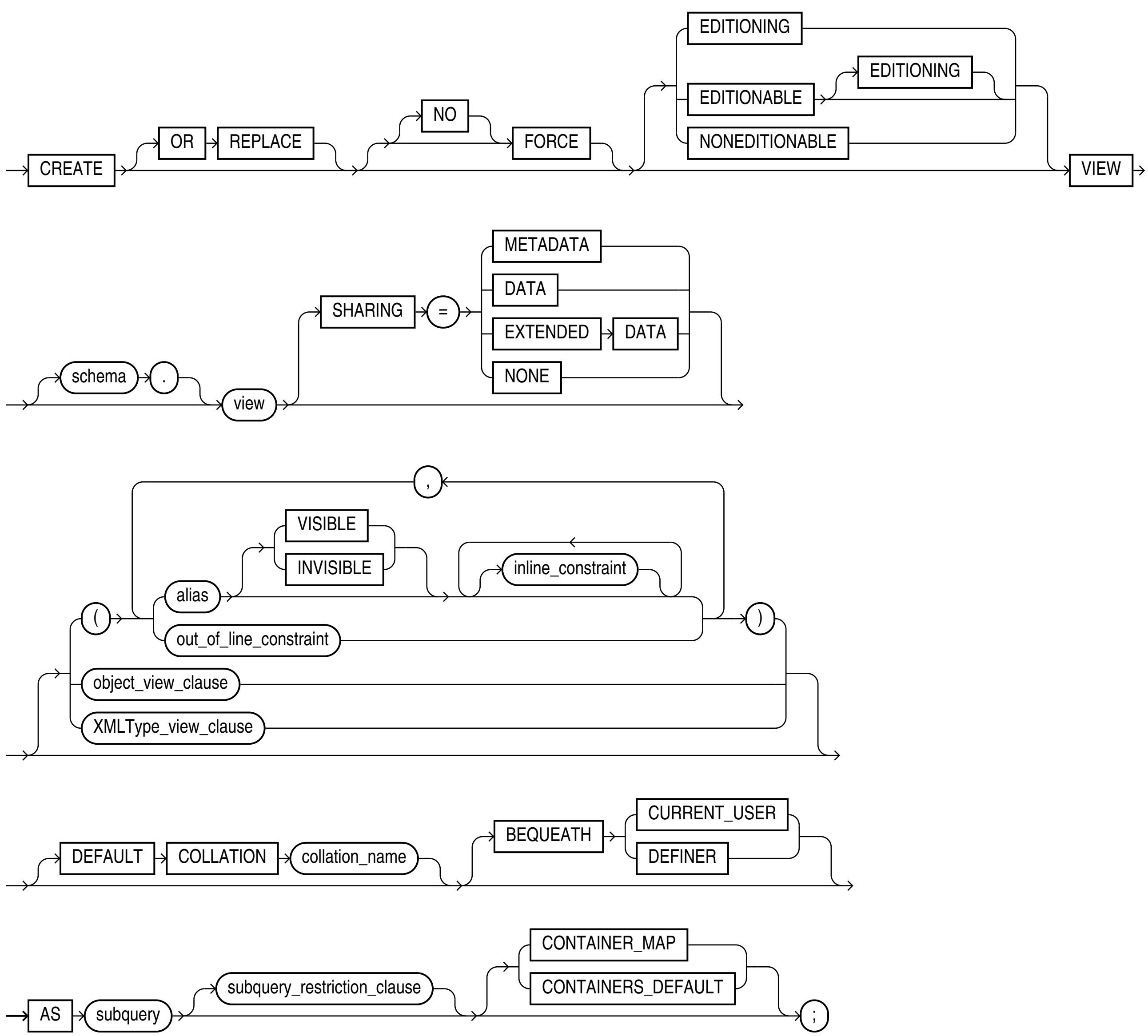
To use the basic constructor method of an object type when creating an object view, one of the following must be true:

- The object type must belong to the same schema as the view to be created.
- You must have the `EXECUTE ANY TYPE` system privileges.
- You must have the `EXECUTE` object privilege on that object type.

> **See Also:**
>
> SELECT , INSERT , UPDATE , and DELETE for information on the privileges required by the owner of a view on the base tables or views of the view being created

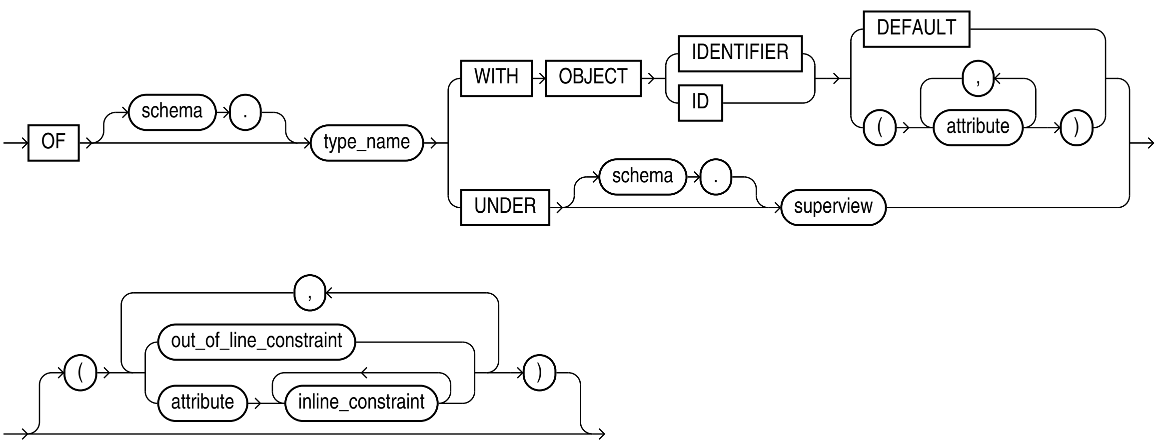
Syntax

***create_view*::=**



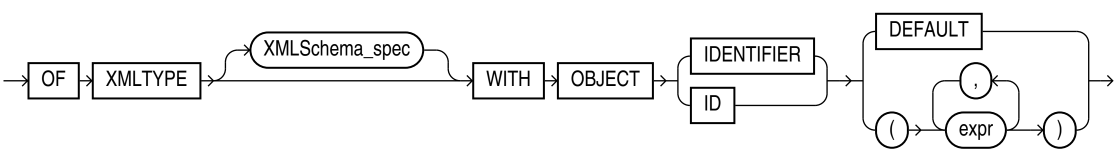
(*inline_constraint*::= and *out_of_line_constraint*::=, *object_view_clause*::=, *XMLType_view_clause*::=, *subquery*::=—part of `SELECT`, *subquery_restriction_clause*::=)

***object_view_clause*::=**

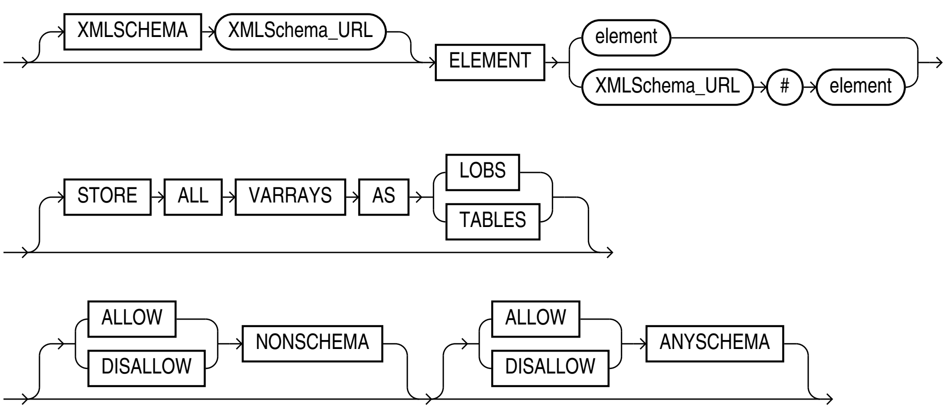


(*inline_constraint*::= and *out_of_line_constraint*::=)

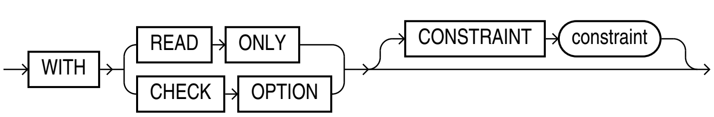
***XMLType_view_clause*::=**



***XMLSchema_spec*::=**



***subquery_restriction_clause*::=**



Semantics

OR REPLACE

Specify `OR REPLACE` to re-create the view if it already exists. You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it.

`INSTEAD OF` triggers defined on a conventional view are dropped when the view is re-created. DML triggers defined on an editioning view are retained when an editioning view is re-created. However, such triggers can be rendered permanently invalid if the editioning view has changed so that it can no longer be compiled—for example if an editioning view column referenced in the trigger definition has been dropped.

If any materialized views are dependent on *`view`*, then those materialized views will be marked `UNUSABLE` and will require a full refresh to restore them to a usable state. Invalid materialized views cannot be used by query rewrite and cannot be refreshed until they are recompiled.

You cannot replace a conventional view with an editioning view or an editioning view with a conventional view. See *Oracle Database Development Guide* for more information on editioning views.

> **See Also:**
>
> - ALTER MATERIALIZED VIEW for information on refreshing invalid materialized views
> - *Oracle Database Concepts* for information on materialized views in general
> - CREATE TRIGGER for more information about the `INSTEAD OF` clause

FORCE

Specify `FORCE` if you want to create the view regardless of whether the base tables of the view or the referenced object types exist or the owner of the schema containing the view has privileges on them. These conditions must be true before any `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements can be issued against the view.

If the view definition contains any constraints, `CREATE VIEW` ... `FORCE` fails if the base table does not exist or the referenced object type does not exist. `CREATE VIEW` ... `FORCE` also fails if the view definition names a constraint that does not exist.

NO FORCE

Specify `NOFORCE` if you want to create the view only if the base tables exist and the owner of the schema containing the view has privileges on them. This is the default.

EDITIONING

Use this clause to create an **editioning view**. An editioning view is a single-table view that selects all rows from the base table and displays a subset of the base table columns. You can use an editioning view to isolate an application from DDL changes to the base table during administrative operations such as upgrades. You can obtain information about the relationship of existing editioning view to their base tables by querying the `USER_`, `ALL_`, and `DBA_EDITIONING_VIEW` data dictionary views.

The owner of an editioning view must be editions-enabled. Refer to ENABLE EDITIONS for more information.

Notes on Editioning Views

Editioning views differ from conventional views in several important ways:

- Editioning views are intended only to select and provide aliases for a subset of columns in a table. Therefore, the syntax for creating an editioning view is more limited than the syntax for creating a conventional view. Any violation of the restrictions that follow causes the creation of the view to fail, even if you specify `FORCE`.
- You can create DML triggers on editioning views. In this case, the database considers the editioning view to be the base object of the trigger. Such triggers fire when a DML operation target the editioning view itself. They do not fire if the DML operation targets the base table.
- You cannot create `INSTEAD OF` triggers on editioning views.

Restrictions on Editioning Views

Editioning views are subject to the following restrictions:

- Within any edition, you can create only one editioning view for any single table.
- You cannot specify the *`object_view_clause`*, *`XMLType_view_clause`*, or `BEQUEATH` clause.
- You cannot define a constraint `WITH CHECK OPTION` on an editioning view.
- In the select list of the defining subquery, you can specify only simple references to the columns of the base table, and you can specify each column of the base table only once in the select list. The asterisk wildcard symbol `*` and *`t_alias`*`.*` are supported to designate all columns of a base table.
- The `FROM` clause of the defining subquery of the view can reference only a single existing database table. Joins are not permitted. The base table must be in the same schema as the view being created. You cannot use a synonym to identify the table, but you can specify a table alias.
- The following clauses of the defining subquery are not valid for editioning views: *`subquery_factoring_clause`*, `DISTINCT` or `UNIQUE`, *`where_clause`*, *`hierarchical_query_clause`*, *`group_by_clause`*, `HAVING` condition, *`model_clause`*, or the set operators (`UNION`, `INTERSECT`, or `MINUS`)

> **See Also:**
>
> - *Oracle Database Development Guide* for detailed information about editioning views
> - CREATE EDITION for information about editions, including an example of an editioning view

EDITIONABLE | NONEDITIONABLE

Use these clauses to specify whether the view becomes an editioned or noneditioned object if editioning is enabled for the schema object type `VIEW` in *`schema`*. The default is `EDITIONABLE`. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

schema

Specify the schema to contain the view. If you omit *`schema`*, then Oracle Database creates the view in your own schema.

view

Specify the name of the view or the object view. The name must satisfy the requirements listed in "Database Object Naming Rules ".

Restriction on Views

If a view has `INSTEAD OF` triggers, then any views created on it must have `INSTEAD OF` triggers, even if the views are inherently updatable.

> **See Also:**
>
> "Creating a View: Example"

SHARING

This clause applies only when creating a view in an application root. This type of view is called an application common object and its data can be shared with the application PDBs that belong to the application root. To determine how the view data is shared, specify one of the following sharing attributes:

- `METADATA` - A metadata link shares the view's metadata, but its data is unique to each container. This type of view is referred to as a **metadata-linked application common object**.
- `DATA` - A data link shares the view, and its data is the same for all containers in the application container. Its data is stored only in the application root. This type of view is referred to as a **data-linked application common object**.
- `EXTENDED DATA` - An extended data link shares the view, and its data in the application root is the same for all containers in the application container. However, each application PDB in the application container can store data that is unique to the application PDB. For this type of view, data is stored in the application root and, optionally, in each application PDB. This type of view is referred to as an **extended data-linked application common object**.
- `NONE` - The view is not shared.

If you omit this clause, then the database uses the value of the `DEFAULT_SHARING` initialization parameter to determine the sharing attribute of the view. If the `DEFAULT_SHARING` initialization parameter does not have a value, then the default is `METADATA`.

When creating a conventional view, you can specify METADATA, DATA, EXTENDED DATA, or NONE.

When creating an object view or an XMLTYPE view, you can specify only METADATA or NONE.

You cannot change the sharing attribute of a view after it is created.

> **See Also:**
>
> - *Oracle Database Reference* for more information on the DEFAULT_SHARING initialization parameter
> - *Oracle Database Administrator's Guide* for complete information on creating application common objects

alias

Specify names for the expressions selected by the defining query of the view. The number of aliases must match the number of expressions selected by the view. Aliases must follow the rules for naming Oracle Database schema objects. Aliases must be unique within the view.

If you omit the aliases, then the database derives them from the columns or column aliases in the query. For this reason, you must use aliases if the query contains expressions rather than only column names. Also, you must specify aliases if the view definition includes constraints.

Restriction on View Aliases

You cannot specify an alias when creating an object view.

> **See Also:**
>
> "Syntax for Schema Objects and Parts in SQL Statements"

VISIBLE | INVISIBLE

Use this clause to specify whether a view column is VISIBLE or INVISIBLE. By default, view columns are VISIBLE regardless of their visibility in the base tables, unless you specify INVISIBLE. This applies to conventional views and editioning views. For complete information on these clauses, refer to "VISIBLE | INVISIBLE" in the documentation on CREATE TABLE.

inline_constraint* and *out_of_line_constraint

You can specify constraints on views and object views. You define the constraint at the view level using the *out_of_line_constraint* clause. You define the constraint as part of column or attribute specification using the *inline_constraint* clause after the appropriate alias.

Oracle Database does not enforce view constraints. For a full discussion of view constraints, including restrictions, refer to "View Constraints ".

> **See Also:**
>
> "Creating a View with Constraints: Example"

object_view_clause

The `object_view_clause` lets you define a view on an object type.

> **See Also:**
>
> "Creating an Object View: Example"

OF *type_name* Clause

Use this clause to explicitly create an **object view** of type `type_name`. The columns of an object view correspond to the top-level attributes of type `type_name`. Each row will contain an object instance and each instance will be associated with an object identifier as specified in the `WITH OBJECT IDENTIFIER` clause. If you omit `schema`, then the database creates the object view in your own schema.

Object tables, as well as `XMLType` tables, object views, and `XMLType` views, do not have any column names specified for them. Therefore, Oracle Database defines a system-generated pseudocolumn `OBJECT_ID`. You can use this column name in queries and to create object views with the `WITH OBJECT IDENTIFIER` clause.

WITH OBJECT IDENTIFIER Clause

Use the `WITH OBJECT IDENTIFIER` clause to specify a top-level (root) object view. This clause lets you specify the attributes of the object type that will be used as a key to identify each row in the object view. In most cases these attributes correspond to the primary key columns of the base table. You must ensure that the attribute list is unique and identifies exactly one row in the view. The `WITH OBJECT IDENTIFIER` and `WITH OBJECT ID` clauses can be used interchangeably and are provided for semantic clarity.

Restrictions on Object Views

Object views are subject to the following restrictions:

- If you try to dereference or pin a primary key `REF` that resolves to more than one instance in the object view, then the database returns an error.
- You cannot specify this clause if you are creating a subview, because subviews inherit object identifiers from superviews.

If the object view is defined on an object table or an object view, then you can omit this clause or specify `DEFAULT`.

DEFAULT

Specify `DEFAULT` if you want the database to use the intrinsic object identifier of the underlying object table or object view to uniquely identify each row.

attribute

For *attribute*, specify an attribute of the object type from which the database should create the object identifier for the object view.

UNDER Clause

Use the UNDER clause to specify a subview based on an object superview.

Restrictions on Subviews

Subviews are subject to the following restrictions:

- You must create a subview in the same schema as the superview.
- The object type *type_name* must be the immediate subtype of *superview*.
- You can create only one subview of a particular type under the same superview.

> **See Also:**
>
> - CREATE TYPE for information about creating objects
> - *Oracle Database Reference* for information on data dictionary views

XMLType_view_clause

Use this clause to create an XMLType view, which displays data from an XMLSchema-based table of type XMLType. The *XMLSchema_spec* indicates the XMLSchema to be used to map the XML data to its object-relational equivalents. The XMLSchema must already have been created before you can create an XMLType view.

The WITH OBJECT IDENTIFIER and WITH OBJECT ID clauses can be used interchangeably and are provided for semantic clarity.

Object tables, as well as XMLType tables, object views, and XMLType views, do not have any column names specified for them. Therefore, Oracle Database defines a system-generated pseudocolumn OBJECT_ID. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

> **See Also:**
>
> - *Oracle XML DB Developer's Guide* for information on XMLType views and XMLSchemas
> - "Creating an XMLType View: Example" and "Creating a View on an XMLType Table: Example"

DEFAULT COLLATION

Use this clause to specify the default collation for the view. The default collation is used as the derived collation for all the character literals included in the defining query of the view. The default collation is not used by the view columns; the collations for the view columns are derived from the view's defining subquery. The CREATE VIEW statement fails with an error if any of its character columns is based on an expression in the defining subquery that has no derived collation.

For *collation_name*, specify a valid named collation or pseudo-collation.

If you omit this clause, then the default collation for the view is set to the effective schema default collation of the schema containing the view. Refer to the DEFAULT_COLLATION clause of ALTER SESSION for more information on the effective schema default collation.

You can specify the DEFAULT COLLATION clause only if the COMPATIBLE initialization parameter is set to 12.2 or greater, and the MAX_STRING_SIZE initialization parameter is set to EXTENDED.

To change the default collation for a view, you must recreate the view.

Restriction on the Default Collation for Views

If the defining query of the view contains the WITH *plsql_declarations* clause, then the default collation of the view must be USING_NLS_COMP.

BEQUEATH

Use the BEQUEATH clause to specify whether functions referenced in the view are executed using the view invoker's rights or the view definer's rights.

CURRENT_USER

If you specify BEQUEATH CURRENT_USER, then functions referenced by the view are executed using the view invoker's rights as long as one of the following conditions is met:

- The view owner has the INHERIT PRIVILEGES object privilege on the invoking user.
- The view owner has the INHERIT ANY PRIVILEGES system privilege.

If a query of the view invokes an identity- or privilege-sensitive SQL function, or an invoker's rights PL/SQL or Java function, then the current schema, current user, and currently enabled roles within the operation's execution are inherited from the querying user's environment, rather than from the owner of the view.

This clause does not turn the view itself into an invoker's rights object. Name resolution within the view is still handled using the view owner's schema, and privilege checking for the view is done using the view owner's privileges.

DEFINER

If you specify BEQUEATH DEFINER, then functions referenced by the view are executed using the view definer's rights. If a query on the view invokes an identity- or privilege-sensitive SQL function, or an invoker's rights PL/SQL or Java function, then the current schema, current user, and currently enabled roles within the operation's execution are inherited from the owner of the view.

Name resolution within the view is handled using the view owner's schema, and privilege checking for the view is done using the view owner's privileges.

This is the default.

Restriction on the BEQUEATH Clause

You cannot specify this clause with the EDITIONING clause.

> **See Also:**
>
> *Oracle Database Security Guide* for more information on controlling invoker's rights and definer's rights in views

AS *subquery*

Specify a subquery that identifies columns and rows of the table(s) that the view is based on. The select list of the subquery can contain up to 1000 expressions.

If you create views that refer to remote tables and views, then the database links you specify must have been created using the `CONNECT TO` clause of the `CREATE DATABASE LINK` statement, and you must qualify them with a schema name in the view subquery.

If you create a view with the *`flashback_query_clause`* in the defining query, then the database does not interpret the `AS OF` expression at create time but rather each time a user subsequently queries the view.

> **See Also:**
>
> "Creating a Join View: Example" and *Oracle Database Development Guide* for more information on Oracle Flashback Query

Restrictions on the Defining Query of a View

The view query is subject to the following restrictions:

- The subquery cannot select the `CURRVAL` or `NEXTVAL` pseudocolumns.
- If the subquery selects the `ROWID`, `ROWNUM`, or `LEVEL` pseudocolumns, then those columns must have aliases in the view subquery.
- If the subquery uses an asterisk (*) to select all columns of a table, and you later add new columns to the table, then the view will not contain those columns until you re-create the view by issuing a `CREATE OR REPLACE VIEW` statement.
- For object views, the number of elements in the subquery select list must be the same as the number of top-level attributes for the object type. The data type of each of the selecting elements must be the same as the corresponding top-level attribute.
- You cannot specify the `SAMPLE` clause.

The preceding restrictions apply to materialized views as well.

Notes on Updatable Views

The following notes apply to updatable views:

An updatable view is one you can use to insert, update, or delete base table rows. You can create a view to be inherently updatable, or you can create an `INSTEAD OF` trigger on any view to make it updatable.

To learn whether and in what ways the columns of an inherently updatable view can be modified, query the `USER_UPDATABLE_COLUMNS` data dictionary view. The information displayed by this view is meaningful only for inherently updatable views. For a view to be inherently updatable, the following conditions must be met:

- Each column in the view must map to a column of a single table. For example, if a view column maps to the output of a `TABLE` clause (an unnested collection), then the view is not inherently updatable.
- The view must not contain any of the following constructs:

  A set operator

  A `DISTINCT` operator

  An aggregate or analytic function

  A `GROUP BY`, `ORDER BY`, `MODEL`, `CONNECT BY`, or `START WITH` clause

  A collection expression in a `SELECT` list

  A subquery in a `SELECT` list

  A subquery designated `WITH READ ONLY`

  Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*
- In addition, if an inherently updatable view contains pseudocolumns or expressions, then you cannot update base table rows with an `UPDATE` statement that refers to any of these pseudocolumns or expressions.
- If you want a join view to be updatable, then all of the following conditions must be true:
  – The DML statement must affect only one table underlying the join.
  – For an `INSERT` statement, the view must not be created `WITH CHECK OPTION`, and all columns into which values are inserted must come from a **key-preserved table**. A key-preserved table is one for which every primary key or unique key value in the base table is also unique in the join view.
  – For an `UPDATE` statement, the view must not be created `WITH CHECK OPTION`, and all columns updated must be extracted from a key-preserved table.
  – For a `DELETE` statement, if the join results in more than one key-preserved table, then Oracle Database deletes from the first table named in the `FROM` clause, whether or not the view was created `WITH CHECK OPTION`.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for more information on updatable views
> - "Creating an Updatable View: Example", "Creating a Join View: Example" for an example of updatable join views and key-preserved tables, and *Oracle Database PL/SQL Language Reference* for an example of an `INSTEAD OF` trigger on a view that is not inherently updatable

subquery_restriction_clause

Use the *`subquery_restriction_clause`* to restrict the defining query of the view in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `READ ONLY` or `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*n*, where *n* is an integer that makes the constraint name unique within the database.

> **Note:**
>
> For tables, `WITH CHECK OPTION` guarantees that inserts and updates result in tables that the defining table subquery can select. For views, `WITH CHECK OPTION` cannot make this guarantee if:
>
> - There is a subquery within the defining query of this view or any view on which this view is based or
> - `INSERT`, `UPDATE`, or `DELETE` operations are performed using `INSTEAD OF` triggers.

Restriction on the *subquery_restriction_clause*

You cannot specify this clause if you specify an `ORDER BY` clause.

> **See Also:**
>
> "Creating a Read-Only View: Example"

CONTAINER_MAP

Specify the `CONTAINER_MAP` clause to enable the view to be queried using a container map.

CONTAINERS_DEFAULT

Specify the `CONTAINERS_DEFAULT` clause to enable the view for the `CONTAINERS` clause.

Examples

Creating a View: Example

The following statement creates a view of the sample table `employees` named `emp_view`. The view shows the employees in department 20 and their annual salary:

```
CREATE VIEW emp_view AS
   SELECT last_name, salary*12 annual_salary
```

```
   FROM employees
   WHERE department_id = 20;
```

The view declaration need not define a name for the column based on the expression `salary*12`, because the subquery uses a column alias (`annual_salary`) for this expression.

Creating an Editioning View: Example

The following statement creates an editioning view of the `orders` table:

```
CREATE EDITIONING VIEW ed_orders_view (o_id, o_date, o_status)
  AS SELECT order_id, order_date, order_status FROM orders
  WITH READ ONLY;
```

You can use this view to isolate an application from DDL changes to the `orders` table during an administrative operation such as an upgrade. You can create a DML trigger on this view, so that the trigger fires when a DML operation targets the view itself, but does not fire if the DML operation targets the `orders` table.

Creating a View with Constraints: Example

The following statement creates a restricted view of the sample table `hr.employees` and defines a unique constraint on the `email` view column and a primary key constraint for the view on the `emp_id` view column:

```
CREATE VIEW emp_sal (emp_id, last_name,
      email UNIQUE RELY DISABLE NOVALIDATE,
   CONSTRAINT id_pk PRIMARY KEY (emp_id) RELY DISABLE NOVALIDATE)
   AS SELECT employee_id, last_name, email FROM employees;
```

Creating an Updatable View: Example

The following statement creates an updatable view named `clerk` of all clerks in the `employees` table. Only the employees' IDs, last names, department numbers, and jobs are visible in this view, and these columns can be updated only in rows where the employee is a kind of clerk:

```
CREATE VIEW clerk AS
   SELECT employee_id, last_name, department_id, job_id
   FROM employees
   WHERE job_id = 'PU_CLERK'
      or job_id = 'SH_CLERK'
      or job_id = 'ST_CLERK';
```

This view lets you change the `job_id` of a purchasing clerk to purchasing manager (`PU_MAN`):

```
UPDATE clerk SET job_id = 'PU_MAN' WHERE employee_id = 118;
```

The next example creates the same view `WITH CHECK OPTION`. You cannot subsequently insert a new row into `clerk` if the new employee is not a clerk. You can update an employee's `job_id` from one type of clerk to another type of clerk, but the update in the preceding statement would fail, because the view cannot access employees with non-clerk `job_id`.

```
CREATE VIEW clerk AS
   SELECT employee_id, last_name, department_id, job_id
   FROM employees
   WHERE job_id = 'PU_CLERK'
```

```
      or job_id = 'SH_CLERK'
      or job_id = 'ST_CLERK'
   WITH CHECK OPTION;
```

Creating a Join View: Example

A join view is one whose view subquery contains a join. If at least one column in the join has a unique index, then it may be possible to modify one base table in a join view. You can query `USER_UPDATABLE_COLUMNS` to see whether the columns in a join view are updatable. For example:

```
CREATE VIEW locations_view AS
   SELECT d.department_id, d.department_name, l.location_id, l.city
   FROM departments d, locations l
   WHERE d.location_id = l.location_id;

SELECT column_name, updatable
   FROM user_updatable_columns
   WHERE table_name = 'LOCATIONS_VIEW'
   ORDER BY column_name, updatable;

COLUMN_NAME                    UPD
------------------------------ ---
DEPARTMENT_ID                  YES
DEPARTMENT_NAME                YES
LOCATION_ID                    NO
CITY                           NO
```

In the preceding example, the primary key index on the `location_id` column of the `locations` table is not unique in the `locations_view` view. Therefore, `locations` is not a key-preserved table and columns from that base table are not updatable.

```
INSERT INTO locations_view VALUES
   (999, 'Entertainment', 87, 'Roma');
INSERT INTO locations_view VALUES
*
ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join view
```

You can insert, update, or delete a row from the `departments` base table, because all the columns in the view mapping to the `departments` table are marked as updatable and because the primary key of `departments` is retained in the view.

```
INSERT INTO locations_view (department_id, department_name)
   VALUES (999, 'Entertainment');

1 row created.
```

Note:

For you to insert into the table using the view, the view must contain all `NOT NULL` columns of all tables in the join, unless you have specified `DEFAULT` values for the `NOT NULL` columns.

See Also:

Oracle Database Administrator's Guide for more information on updating join views

Creating a Read-Only View: Example

The following statement creates a read-only view named `customer_ro` of the `oe.customers` table. Only the customers' last names, language, and credit limit are visible in this view:

```
CREATE VIEW customer_ro (name, language, credit)
      AS SELECT cust_last_name, nls_language, credit_limit
      FROM customers
      WITH READ ONLY;
```

Creating an Object View: Example

The following example shows the creation of the type `inventory_typ` in the `oc` schema, and the `oc_inventories` view that is based on that type:

```
CREATE TYPE inventory_typ
 OID '82A4AF6A4CD4656DE034080020E0EE3D'
 AS OBJECT
    ( product_id          NUMBER(6)
    , warehouse           warehouse_typ
    , quantity_on_hand    NUMBER(8)
    ) ;
/
CREATE OR REPLACE VIEW oc_inventories OF inventory_typ
 WITH OBJECT OID (product_id)
 AS SELECT i.product_id,
           warehouse_typ(w.warehouse_id, w.warehouse_name, w.location_id),
           i.quantity_on_hand
    FROM inventories i, warehouses w
    WHERE i.warehouse_id=w.warehouse_id;
```

Creating a View on an XMLType Table: Example

The following example builds a regular view on the `XMLType` table `xwarehouses`, which was created in "Examples ":

```
CREATE VIEW warehouse_view AS
   SELECT VALUE(p) AS warehouse_xml
   FROM xwarehouses p;
```

You select from such a view as follows:

```
SELECT e.warehouse_xml.getclobval()
   FROM warehouse_view e
   WHERE EXISTSNODE(warehouse_xml, '//Docks') =1;
```

Creating an XMLType View: Example

In some cases you may have an object-relational table upon which you would like to build an `XMLType` view. The following example creates an object-relational table resembling the `XMLType` column `warehouse_spec` in the sample table `oe.warehouses`, and then creates an `XMLType` view of that table:

```
CREATE TABLE warehouse_table
(
  WarehouseID        NUMBER,
  Area               NUMBER,
  Docks              NUMBER,
  DockType           VARCHAR2(100),
  WaterAccess        VARCHAR2(10),
  RailAccess         VARCHAR2(10),
  Parking            VARCHAR2(20),
  VClearance         NUMBER
);

INSERT INTO warehouse_table
   VALUES(5, 103000,3,'Side Load','false','true','Lot',15);

CREATE VIEW warehouse_view OF XMLTYPE
 XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
    ELEMENT "Warehouse"
    WITH OBJECT ID
    (extract(OBJECT_VALUE, '/Warehouse/Area/text()').getnumberval())
 AS SELECT XMLELEMENT("Warehouse",
            XMLFOREST(WarehouseID as "Building",
                      area as "Area",
                      docks as "Docks",
                      docktype as "DockType",
                      wateraccess as "WaterAccess",
                      railaccess as "RailAccess",
                      parking as "Parking",
                      VClearance as "VClearance"))
  FROM warehouse_table;
```

You query this view as follows:

```
SELECT VALUE(e) FROM warehouse_view e;
```

DELETE

Purpose

Use the `DELETE` statement to remove rows from:

- An unpartitioned or partitioned table
- The unpartitioned or partitioned base table of a view
- The unpartitioned or partitioned container table of a writable materialized view
- The unpartitioned or partitioned master table of an updatable materialized view

Prerequisites

For you to delete rows from a table, the table must be in your own schema or you must have the `DELETE` object privilege on the table.

For you to delete rows from an updatable materialized view, the materialized view must be in your own schema or you must have the `DELETE` object privilege on the materialized view.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have the `DELETE` object privilege on the base table. Also, if the view is in a schema other than your own, then you must have the `DELETE` object privilege on the view.

The DELETE ANY TABLE system privilege also allows you to delete rows from any table or table partition or from the base table of any view.

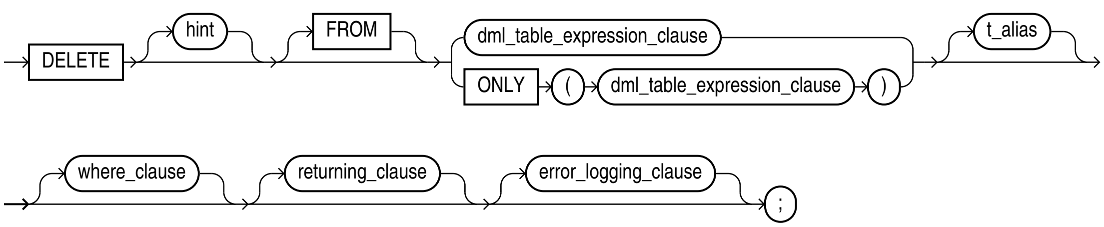
To delete rows from an object on a remote database, you must also have the READ or SELECT object privilege on the object.

If the SQL92_SECURITY initialization parameter is set to TRUE and the DELETE operation references table columns, such as the columns in a *where_clause*, then you must also have the SELECT object privilege on the object from which you want to delete rows.

You cannot delete rows from a table if a function-based index on the table has become invalid. You must first validate the function-based index.

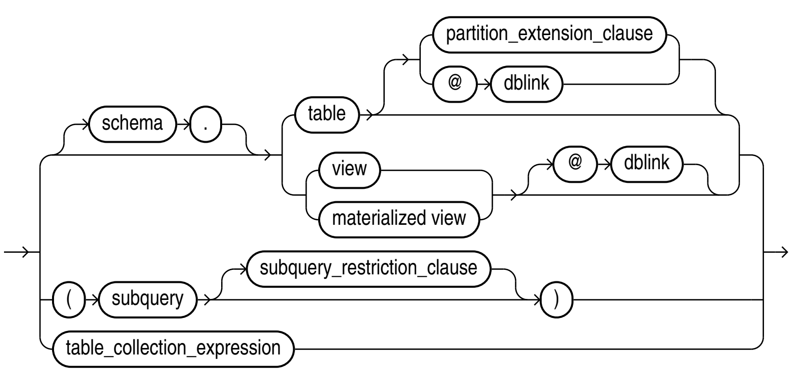
Syntax

***delete*::=**



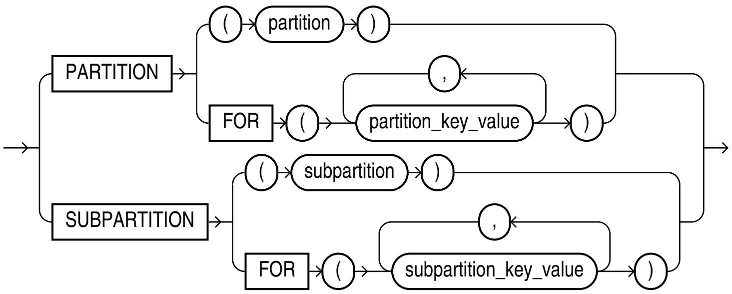
(*DML_table_expression_clause::=*, *where_clause::=*, *returning_clause::=*, *error_logging_clause::=*)

***DML_table_expression_clause*::=**

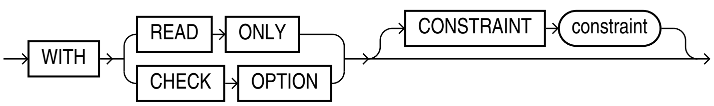


(*partition_extension_clause::=*, *subquery::=*, *subquery_restriction_clause::=*, *table_collection_expression::=*)

partition_extension_clause::=

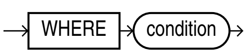


subquery_restriction_clause::=

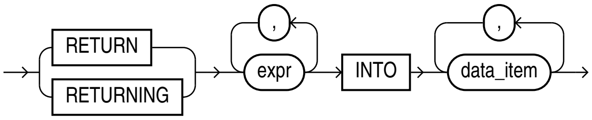


table_collection_expression::=

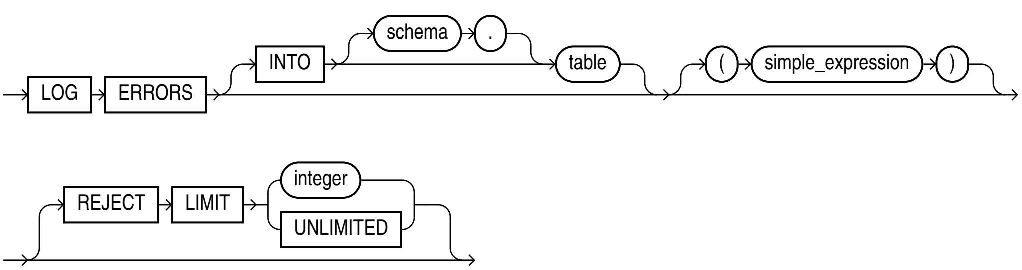
where_clause::=



returning_clause::=



error_logging_clause::=



Semantics

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:**
>
> "Hints " for the syntax and description of hints

from_clause

Use the `FROM` clause to specify the database objects from which you are deleting rows.

The `ONLY` syntax is relevant only for views. Use the `ONLY` clause if the view in the `FROM` clause belongs to a view hierarchy and you do not want to delete rows from any of its subviews.

DML_table_expression_clause

Use this clause to specify the objects from which data is being deleted.

schema

Specify the schema containing the table or view. If you omit *`schema`*, then Oracle Database assumes the table or view is in your own schema.

table* | *view* | *materialized view* | *subquery

Specify the name of a table, view, materialized view, or the column or columns resulting from a subquery, from which the rows are to be deleted.

When you delete rows from an updatable view, Oracle Database deletes rows from the base table.

You cannot delete rows from a read-only materialized view. If you delete rows from a writable materialized view, then the database removes the rows from the underlying container table. However, the deletions are overwritten at the next refresh operation. If you delete rows from an updatable materialized view that is part of a materialized view group, then the database also removes the corresponding rows from the master table.

If *`table`* or the base table of *`view`* or the master table of *`materialized_view`* contains one or more domain index columns, then this statement executes the appropriate indextype delete routine.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on these routines

Issuing a `DELETE` statement against a table fires any `DELETE` triggers defined on the table.

All table or index space released by the deleted rows is retained by the table and index.

partition_extension_clause

Specify the name or partition key value of the partition or subpartition targeted for deletes within the object.

You need not specify the partition name when deleting values from a partitioned object. However, in some cases, specifying the partition name is more efficient than a complicated *`where_clause`*.

> **See Also:**
>
> "References to Partitioned Tables and Indexes " and "Deleting Rows from a Partition: Example"

dblink

Specify the complete or partial name of a database link to a remote database where the object is located. You can delete rows from a remote object only if you are using Oracle Database distributed functionality.

> **Note:**
>
> Starting with Oracle Database 12*c* Release 2 (12.2), the `DELETE` statement accepts remote LOB locators as bind variables. Refer to the “Distributed LOBs” chapter in *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

> **See Also:**
>
> "References to Objects in Remote Databases " for information on referring to database links and "Deleting Rows from a Remote Database: Example"

If you omit *`dblink`*, then the database assumes that the object is located on the local database.

subquery_restriction_clause

The *`subquery_restriction_clause`* lets you restrict the subquery in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_Cn`, where n is an integer that makes the constraint name unique within the database.

> **See Also:**
>
> "Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *`table_collection_expression`* lets you inform Oracle that the value of *`collection_expression`* should be treated as a table for purposes of query and DML operations. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the `TABLE` collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

> **Note:**
>
> In earlier releases of Oracle, when *`collection_expression`* was a subquery, *`table_collection_expression`* was expressed as `THE` *`subquery`*. That usage is now deprecated.

You can use a *`table_collection_expression`* in a correlated subquery to delete rows with values that also exist in another table.

> **See Also:**
>
> "Table Collections: Examples"

collection_expression

Specify a subquery that selects a nested table column from the object from which you are deleting.

Restrictions on the *dml_table_expression_clause* Clause

This clause is subject to the following restrictions:

- You cannot execute this statement if *table* or the base or master table of *view* or *materialized_view* contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- You cannot specify the ORDER BY clause in the subquery of the *DML_table_expression_clause*.
- You cannot delete from a view except through INSTEAD OF triggers if the defining query of the view contains one of the following constructs:

  A set operator
  A DISTINCT operator
  An aggregate or analytic function
  A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
  A collection expression in a SELECT list
  A subquery in a SELECT list
  A subquery designated WITH READ ONLY
  Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the DELETE statement will fail unless the SKIP_UNUSABLE_INDEXES initialization parameter has been set to true.

> **See Also:**
>
> ALTER SESSION

t_alias

Provide a **correlation name** for the table, view, materialized view, subquery, or collection value to be referenced elsewhere in the statement. This alias is required if the *DML_table_expression_clause* references any object type attributes or object type methods. Table aliases are generally used in DELETE statements with correlated queries.

where_clause

Use the *where_clause* to delete only rows that satisfy the condition. The condition can reference the object from which you are deleting and can contain a subquery. You can delete rows from a remote object only if you are using Oracle Database distributed functionality. Refer to Conditions for the syntax of *condition*.

If this clause contains a *subquery* that refers to remote objects, then the DELETE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *DML_table_expression_clause* refers to any remote objects, then the DELETE operation will run serially without notification. Refer to the *parallel_clause* in the CREATE TABLE documentation for additional information.

If you omit *dblink*, then the database assumes that the table or view is located on the local database.

If you omit the *where_clause*, then the database deletes all rows of the object.

returning_clause

This clause lets you return values from deleted columns, and thereby eliminate the need to issue a `SELECT` statement following the `DELETE` statement.

The returning clause retrieves the rows affected by a DML statement. You can specify this clause for tables and materialized views and for views with a single base table.

When operating on a single row, a DML statement with a *`returning_clause`* can retrieve column expressions using the affected row, rowid, and `REF`s to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the *`returning_clause`* stores values from expressions, rowids, and `REF`s involving the affected rows in bind arrays.

expr

Each item in the *`expr`* list must be a valid expression syntax.

INTO

The `INTO` clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *`data_item`* list.

data_item

Each *`data_item`* is a host variable or PL/SQL variable that stores the retrieved *`expr`* value.

For each expression in the `RETURNING` list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the `INTO` list.

Restrictions on the RETURNING Clause

The following restrictions apply to the `RETURNING` clause:

- The *`expr`* is restricted as follows:
  - For `UPDATE` and `DELETE` statements each *`expr`* must be a simple expression or a single-set aggregate function expression. You cannot combine simple expressions and single-set aggregate function expressions in the same *`returning_clause`*. For `INSERT` statements, each *`expr`* must be a simple expression. Aggregate functions are not supported in an `INSERT` statement `RETURNING` clause.
  - Single-set aggregate function expressions cannot include the `DISTINCT` keyword.
- If the *`expr`* list contains a primary key column or other `NOT NULL` column, then the update statement fails if the table has a `BEFORE UPDATE` trigger defined on it.
- You cannot specify the *`returning_clause`* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve `LONG` types with this clause.
- You cannot specify this clause for a view on which an `INSTEAD OF` trigger has been defined.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for information on using the `BULK COLLECT` clause to return multiple values to collection variables
> - "Using the RETURNING Clause: Example"

error_logging_clause

The `error_logging_clause` has the same behavior in `DELETE` statement as it does in an `INSERT` statement. Refer to the `INSERT` statement *error_logging_clause* for more information.

> **See Also:**
>
> "Inserting Into a Table with Error Logging: Example"

Examples

Deleting Rows: Examples

The following statement deletes all rows from the sample table `oe.product_descriptions` where the value of the `language_id` column is `AR`:

```
DELETE FROM product_descriptions
   WHERE language_id = 'AR';
```

The following statement deletes from the sample table `hr.employees` purchasing clerks whose commission rate is less than 10%:

```
DELETE FROM employees
   WHERE job_id = 'SA_REP'
   AND commission_pct < .2;
```

The following statement has the same effect as the preceding example, but uses a subquery:

```
DELETE FROM (SELECT * FROM employees)
   WHERE job_id = 'SA_REP'
   AND commission_pct < .2;
```

Deleting Rows from a Remote Database: Example

The following statement deletes specified rows from the `locations` table owned by the user `hr` on a database accessible by the database link `remote`:

```
DELETE FROM hr.locations@remote
   WHERE location_id > 3000;
```

Deleting Nested Table Rows: Example

For an example that deletes nested table rows, refer to "Table Collections: Examples".

Deleting Rows from a Partition: Example

The following example removes rows from partition `sales_q1_1998` of the `sh.sales` table:

```
DELETE FROM sales PARTITION (sales_q1_1998)
   WHERE amount_sold > 1000;
```

Using the RETURNING Clause: Example

The following example returns column `salary` from the deleted rows and stores the result in bind variable `:bnd1`. The bind variable must already have been declared.

```
DELETE FROM employees
   WHERE job_id = 'SA_REP'
   AND hire_date + TO_YMINTERVAL('01-00') < SYSDATE
   RETURNING salary INTO :bnd1;
```

DISASSOCIATE STATISTICS

Purpose

Use the `DISASSOCIATE STATISTICS` statement to disassociate default statistics or a statistics type from columns, standalone functions, packages, types, domain indexes, or indextypes.

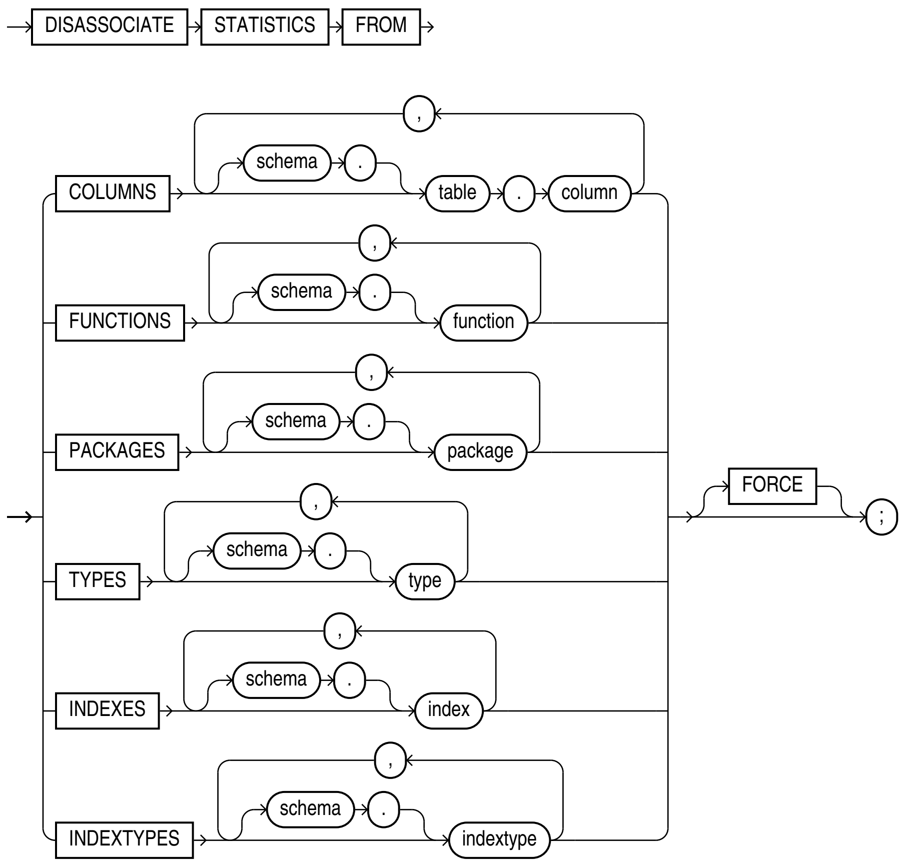
> **See Also:**
>
> ASSOCIATE STATISTICS for more information on statistics type associations

Prerequisites

To issue this statement, you must have the appropriate privileges to alter the underlying table, function, package, type, domain index, or indextype.

Syntax

***disassociate_statistics*::=**



Semantics

FROM COLUMNS | FUNCTIONS | PACKAGES | TYPES | INDEXES | INDEXTYPES

Specify one or more columns, standalone functions, packages, types, domain indexes, or indextypes from which you are disassociating statistics.

If you do not specify `schema`, then Oracle Database assumes the object is in your own schema.

If you have collected user-defined statistics on the object, then the statement fails unless you specify `FORCE`.

FORCE

Specify `FORCE` to remove the association regardless of whether any statistics exist for the object using the statistics type. If statistics do exist, then the statistics are deleted before the association is deleted.

> **Note:**
>
> When you drop an object with which a statistics type has been associated, Oracle Database automatically disassociates the statistics type with the `FORCE` option and drops all statistics that have been collected with the statistics type.

Examples

Disassociating Statistics: Example

This statement disassociates statistics from the `emp_mgmt` package. See *Oracle Database PL/SQL Language Reference* for the example that creates this package in the `hr` schema.

```
DISASSOCIATE STATISTICS FROM PACKAGES hr.emp_mgmt;
```

DROP ANALYTIC VIEW

Purpose

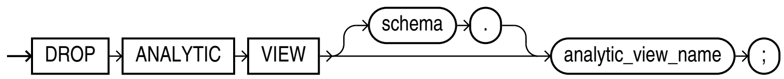
Use the `DROP ANALYTIC VIEW` statement to drop an analytic view. An `ANALYTIC VIEW` object is a component of analytic views.

Prerequisites

To drop an analytic view in your own schema, you must have the `DROP ANALYTIC VIEW` system privilege. To drop an analytic view in another user's schema, you must have the `DROP ANY ANALYTIC VIEW` system privilege.

Syntax

***drop_analytic_view*::=**



Semantics

schema

Specify the schema in which the analytic view exists. If you do not specify a schema, then Oracle Database looks for the analytic view in your own schema.

analytic_view_name

Specify the name of the analytic view to drop.

Example

The following statement drops the specified analytic view object:

```
DROP ANALYTIC VIEW sales_av;
```

DROP ATTRIBUTE DIMENSION

Purpose

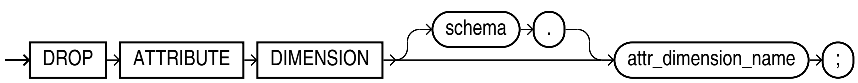
Use the `DROP ATTRIBUTE DIMENSION` statement to drop an attribute dimension. An `ATTRIBUTE DIMENSION` object is a component of analytic views.

Prerequisites

To drop an attribute dimension in your own schema, you must have the `DROP ATTRIBUTE DIMENSION` system privilege. To drop an analytic view in another user's schema, you must have the `DROP ANY ATTRIBUTE DIMENSION` system privilege.

Syntax

***drop_attribute_dimension*::=**



Semantics

schema

Specify the schema in which the attribute dimension exists. If you do not specify a schema, then Oracle Database looks for the attribute dimension in your own schema.

attr_dimension_name

Specify the name of the attribute dimension to drop.

Example

The following statement drops the specified attribute dimension object:

```
DROP ATTRIBUTE DIMENSION product_attr_dim;
```

DROP AUDIT POLICY (Unified Auditing)

This section describes the `DROP AUDIT POLICY` statement for **unified auditing**. This type of auditing is new beginning with Oracle Database 12*c* and provides a full set of enhanced auditing features. Refer to *Oracle Database Security Guide* for more information on unified auditing.

Purpose

Use the `DROP AUDIT POLICY` statement to remove a unified audit policy from the database.

> **See Also:**
>
> - CREATE AUDIT POLICY (Unified Auditing)
> - ALTER AUDIT POLICY (Unified Auditing)
> - AUDIT (Unified Auditing)
> - NOAUDIT (Unified Auditing)

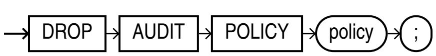
Prerequisites

You must have the `AUDIT SYSTEM` system privilege or the `AUDIT_ADMIN` role.

To drop a common unified audit policy, the current container must be the root and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role. To drop a local unified audit policy, the current container must be the container in which the audit policy was created and you must have the commonly granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` common role, or you must have the locally granted `AUDIT SYSTEM` privilege or the `AUDIT_ADMIN` local role in the container.

Syntax

***drop_audit_policy*::=**



Semantics

policy

Specify the name of the unified audit policy you want to drop. The policy must have been created using the `CREATE AUDIT POLICY` statement.

You can find the names of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view and the names of all *enabled* unified audit policies by querying the `AUDIT_UNIFIED_ENABLED_POLICIES` view

Restriction on Dropping Unified Audit Policies

You cannot drop an enabled unified audit policy. You must first disable the policy using the `NOAUDIT` statement.

> **See Also:**
>
> - CREATE AUDIT POLICY (Unified Auditing)
> - *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES` and `AUDIT_UNIFIED_ENABLED_POLICIES` views

Examples

Dropping a Unified Audit Policy: Example

The following statement drops unified audit policy `table_pol`:

```
DROP AUDIT POLICY table_pol;
```

DROP CLUSTER

Purpose

Use the `DROP CLUSTER` clause to remove a cluster from the database.

> **Note:**
>
> When you drop a cluster, any tables in the recycle bin that were once part of that cluster are purged from the recycle bin and can no longer be recovered with a `FLASHBACK TABLE` operation.

You cannot uncluster an individual table. Instead you must perform these steps:

1. Create a new table with the same structure and contents as the old one, but with no `CLUSTER` clause.
2. Drop the old table.
3. Use the `RENAME` statement to give the new table the name of the old one.
4. Grant privileges on the new unclustered table. Grants on the old clustered table do not apply.

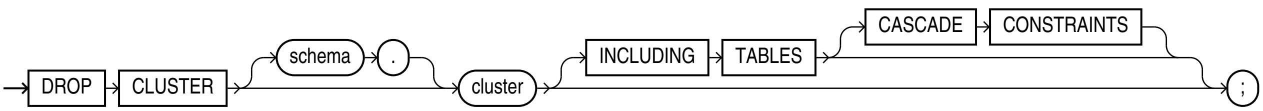
> **See Also:**
>
> CREATE TABLE, DROP TABLE , RENAME , GRANT for information on these steps

Prerequisites

The cluster must be in your own schema or you must have the `DROP ANY CLUSTER` system privilege.

Syntax

***drop_cluster*::=**



Semantics

schema

Specify the schema containing the cluster. If you omit *`schema`*, then the database assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be dropped. Dropping a cluster also drops the cluster index and returns all cluster space, including data blocks for the index, to the appropriate tablespace(s).

INCLUDING TABLES

Specify `INCLUDING TABLES` to drop all tables that belong to the cluster.

CASCADE CONSTRAINTS

Specify `CASCADE CONSTRAINTS` to drop all referential integrity constraints from tables outside the cluster that refer to primary and unique keys in tables of the cluster. If you omit this clause and such referential integrity constraints exist, then the database returns an error and does not drop the cluster.

Examples

Dropping a Cluster: Examples

The following examples drop the clusters created in the "Examples" section of `CREATE CLUSTER`.

The following statements drops the `language` cluster:

```
DROP CLUSTER language;
```

The following statement drops the `personnel` cluster as well as tables `dept_10` and `dept_20` and any referential integrity constraints that refer to primary or unique keys in those tables:

```
DROP CLUSTER personnel
   INCLUDING TABLES
   CASCADE CONSTRAINTS;
```

16 SQL Statements: DROP CONTEXT to DROP JAVA

This chapter contains the following SQL statements:

- DROP CONTEXT
- DROP DATABASE
- DROP DATABASE LINK
- DROP DIMENSION
- DROP DIRECTORY
- DROP DISKGROUP
- DROP EDITION
- DROP FLASHBACK ARCHIVE
- DROP FUNCTION
- DROP HIERARCHY
- DROP INDEX
- DROP INDEXTYPE
- DROP INMEMORY JOIN GROUP
- DROP JAVA

DROP CONTEXT

Purpose

Use the `DROP CONTEXT` statement to remove a context namespace from the database.

Removing a context namespace does not invalidate any context under that namespace that has been set for a user session. However, the context will be invalid when the user next attempts to set that context.

> **See Also:**
>
> CREATE CONTEXT and *Oracle Database Security Guide* for more information on contexts

Prerequisites

You must have the `DROP ANY CONTEXT` system privilege.

Syntax

***drop_context*::=**

Semantics

namespace

Specify the name of the context namespace to drop. You cannot drop the built-in namespace `USERENV`.

> **See Also:**
>
> SYS_CONTEXT for information on the `USERENV` namespace

Examples

Dropping an Application Context: Example

The following statement drops the context created in CREATE CONTEXT:

```
DROP CONTEXT hr_context;
```

DROP DATABASE

Purpose

> **Note:**
>
> You cannot roll back a `DROP DATABASE` statement.

Use the `DROP DATABASE` statement to drop the database. This statement is useful when you want to drop a test database or drop an old database after successful migration to a new host.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information on dropping the database

Prerequisites

You must have the `SYSDBA` system privilege to issue this statement. The database must be mounted in exclusive and restricted mode, and it must be closed.

Syntax

***drop_database*::=**

Semantics

When you issue this statement, Oracle Database drops the database and deletes all control files and data files listed in the control file. If the database used a server parameter file (spfile), then the spfile is also deleted.

Archived logs and backups are not removed, but you can use Recovery Manager (RMAN) to remove them. If the database is on raw disks, then this statement does not delete the actual raw disk special files.

DROP DATABASE LINK

Purpose

Use the `DROP DATABASE LINK` statement to remove a database link from the database.

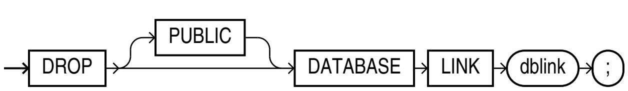
> **See Also:**
>
> CREATE DATABASE LINK for information on creating database links

Prerequisites

A private database link must be in your own schema. To drop a `PUBLIC` database link, you must have the `DROP PUBLIC DATABASE LINK` system privilege.

Syntax

***drop_database_link*::=**



Semantics

PUBLIC

You must specify `PUBLIC` to drop a `PUBLIC` database link.

dblink

Specify the name of the database link to be dropped.

Restriction on Dropping Database Links

You cannot drop a database link in another user's schema, and you cannot qualify `dblink` with the name of a schema, because periods are permitted in names of database links. Therefore, Oracle Database interprets the entire name, such as `ralph.linktosales`, as the name of a database link in your schema rather than as a database link named `linktosales` in the schema `ralph`.

Examples

Dropping a Database Link: Example

The following statement drops the public database link named `remote`, which was created in "Defining a Public Database Link: Example":

```
DROP PUBLIC DATABASE LINK remote;
```

DROP DIMENSION

Purpose

Use the `DROP DIMENSION` statement to remove the named dimension.

This statement does not invalidate materialized views that use relationships specified in dimensions. However, requests that have been rewritten by query rewrite may be invalidated, and subsequent operations on such views may execute more slowly.

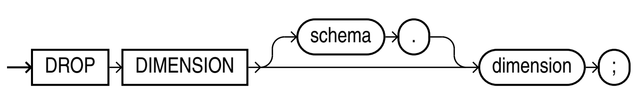
> **See Also:**
>
> - CREATE DIMENSION and ALTER DIMENSION for information on creating and modifying a dimension
> - *Oracle Database Concepts* for general information about dimensions

Prerequisites

The dimension must be in your own schema or you must have the `DROP ANY DIMENSION` system privilege to use this statement.

Syntax

***drop_dimension*::=**



Semantics

schema

Specify the name of the schema in which the dimension is located. If you omit *schema*, then Oracle Database assumes the dimension is in your own schema.

dimension

Specify the name of the dimension you want to drop. The dimension must already exist.

Examples

Dropping a Dimension: Example

This example drops the `sh.customers_dim` dimension:

```
DROP DIMENSION customers_dim;
```

> **See Also:**
>
> "Creating a Dimension: Examples" and "Modifying a Dimension: Examples" for examples of creating and modifying this dimension

DROP DIRECTORY

Purpose

Use the `DROP DIRECTORY` statement to remove a directory object from the database.

> **See Also:**
>
> CREATE DIRECTORY for information on creating a directory

Prerequisites

To drop a directory, you must have the `DROP ANY DIRECTORY` system privilege.

> **Note:**
>
> Do not drop a directory when files in the associated file system are being accessed by PL/SQL or OCI programs.

Syntax

***drop_directory*::=**

Semantics

directory_name

Specify the name of the directory database object to be dropped.

Oracle Database removes the directory object but does not delete the associated operating system directory on the server file system.

Examples

Dropping a Directory: Example

The following statement drops the directory object `bfile_dir`:

```
DROP DIRECTORY bfile_dir;
```

> **See Also:**
>
> "Creating a Directory: Examples"

DROP DISKGROUP

> **Note:**
>
> This SQL statement is valid only if you are using Oracle ASM and you have started an Oracle ASM instance. You must issue this statement from within the Oracle ASM instance, not from a normal database instance. For information on starting an Oracle ASM instance, refer to *Oracle Automatic Storage Management Administrator's Guide*.

Purpose

The `DROP DISKGROUP` statement lets you drop an Oracle ASM disk group along with all the files in the disk group. Oracle ASM first ensures that no files in the disk group are open. It then drops the disk group and all its member disks and clears the disk header.

See Also:

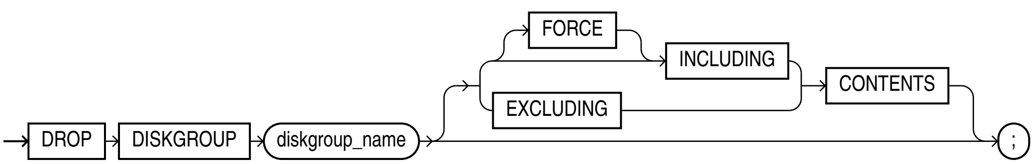
- CREATE DISKGROUP and ALTER DISKGROUP for information on creating and modifying disk groups
- *Oracle Automatic Storage Management Administrator's Guide* for information on Oracle ASM and using disks groups to simplify database administration

Prerequisites

You must have the `SYSASM` system privilege and you must have an Oracle ASM instance started, from which you issue this statement. The disk group to be dropped must be mounted.

Syntax

***drop_diskgroup*::=**



Semantics

diskgroup_name

Specify the name of the disk group you want to drop.

INCLUDING CONTENTS

Specify `INCLUDING CONTENTS` to confirm that Oracle ASM should drop all the files in the disk group. You must specify this clause if the disk group contains any files.

FORCE

This clause clears the headers on the disk belonging to a disk group that cannot be mounted by the Oracle ASM instance. The disk group cannot be mounted by any instance of the database.

The Oracle ASM instance first determines whether the disk group is being used by any other Oracle ASM instance using the same storage subsystem. If it is being used, and if the disk group is in the same cluster, or on the same node, then the statement fails. If the disk group is in a different cluster, then the system further checks to determine whether the disk group is mounted by any instance in the other cluster. If it is mounted elsewhere, then the statement fails. However, this latter check is not as definitive as the checks for disk groups in the same cluster. Therefore, use this clause with caution.

EXCLUDING CONTENTS

Specify `EXCLUDING CONTENTS` to ensure that Oracle ASM drops the disk group only when the disk group is empty. This is the default. If the disk group is not empty, then an error will be returned.

Examples

Dropping a Diskgroup: Example

The following statement drops the Oracle ASM disk group `dgroup_01`, which was created in "Creating a Diskgroup: Example", and all of the files in the disk group:

```
DROP DISKGROUP dgroup_01 INCLUDING CONTENTS;
```

DROP EDITION

Purpose

Use the `DROP EDITION` statement to drop an edition, along with all actual editionable objects it contains. An actual editionable object is an editionable object that has been created or modified in an edition.

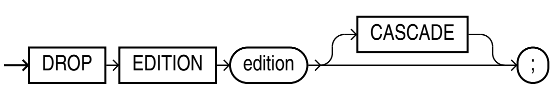
> **See Also:**
>
> CREATE EDITION for a listing of editionable object types

Prerequisites

You must have the `DROP ANY EDITION` system privilege, granted either directly or through a role.

Syntax

***drop_edition*::=**



Semantics

When successful, this statement drops the specified edition, including versions of any objects associated with that edition. Versions of the same objects in other editions are not dropped. Objects that are not editionable, or that are editionable but have not been actualized in the current edition, are not dropped.

You must specify `CASCADE` if the specified edition contains any actual editionable objects.

This statement is subject to the following conditions and restrictions:

- The specified edition cannot have both a parent edition and a child edition.
- The specified edition cannot contain any actual editionable objects that are inherited by a child edition, even if you specify `CASCADE`.
- `DROP EDITION` will fail if you attempt to drop the default edition.

- DROP EDITION will fail if you attempt to drop the root edition and the recycle bin contains at least one object that used to be in that edition before it was dropped. Under these circumstances, even DROP EDITION CASCADE will fail. In this case, you can purge all objects from the recycle bin with the PURGE DBA_RECYCLEBIN statement and then drop the edition. Refer to PURGE for more information.

  DROP EDITION will also fail if you attempt to drop the leaf edition and the recycle bin contains at least one object that used to be in that edition before it was dropped. However, under these circumstances, DROP EDITION CASCADE will succeed.

  The only type of editioned object that might be in the recycle bin is a trigger.

Examples

For examples that use this statement, refer to CREATE EDITION .

DROP FLASHBACK ARCHIVE

Purpose

Use the DROP FLASHBACK ARCHIVE clause to remove a flashback data archive from the system. This statement removes the flashback data archive and all the historical data in it, but does not drop the tablespaces that were used by the flashback data archive.

Prerequisites

You must have the FLASHBACK ARCHIVE ADMINISTER system privilege to drop a flashback data archive.

Syntax

***drop_flashback_archive*::=**

Semantics

flashback_archive

Specify the name of the flashback data archive you want to drop.

> **See Also:**
>
> CREATE FLASHBACK ARCHIVE for information on creating flashback data archives and for some simple examples of using flashback data archives

DROP FUNCTION

Purpose

Functions are defined using PL/SQL. Refer to *Oracle Database PL/SQL Language Reference* for complete information on creating, altering, and dropping functions.

Use the `DROP FUNCTION` statement to remove a standalone stored function from the database.

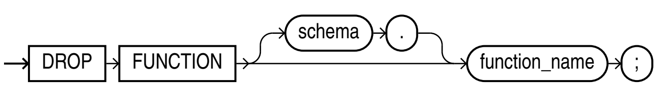
> **Note:**
>
> Do not use this statement to remove a function that is part of a package. Instead, either drop the entire package using the `DROP PACKAGE` statement or redefine the package without the function using the `CREATE PACKAGE` statement with the `OR REPLACE` clause.

Prerequisites

The function must be in your own schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

***drop_function*::=**



Semantics

schema

Specify the schema containing the function. If you omit *`schema`*, then Oracle Database assumes the function is in your own schema.

function_name

Specify the name of the function to be dropped.

Oracle Database invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped function.

If any statistics types are associated with the function, then the database disassociates the statistics types with the `FORCE` option and drops any user-defined statistics collected with the statistics type.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on how Oracle Database maintains dependencies among schema objects, including remote objects
> - ASSOCIATE STATISTICS and DISASSOCIATE STATISTICS for more information on statistics type associations

Examples

Dropping a Function: Example

The following statement drops the function `SecondMax` in the sample schema `oe` and invalidates all objects that depend upon `SecondMax`:

```
DROP FUNCTION oe.SecondMax;
```

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for the example that creates the `SecondMax` function

DROP HIERARCHY

Purpose

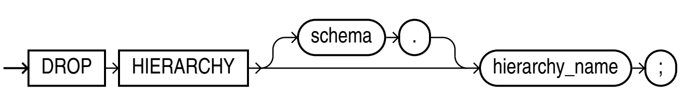
Use the `DROP HIERARCHY` statement to drop a hierarchy. A `HIERARCHY` object is a component of analytic views.

Prerequisites

To drop a hierarchy in your own schema, you must have the `DROP HIERARCHY` system privilege. To drop a hierarchy in another user's schema, you must have the `DROP ANY HIERARCHY` system privilege.

Syntax

***drop_hierarchy*::=**



Semantics

schema

Specify the schema in which the hierarchy exists. If you do not specify a schema, then Oracle Database looks for the hierarchy in your own schema.

hierarchy_name

Specify the name of the hierarchy to drop.

Example

The following statement drops the specified hierarchy object:

```
DROP HIERARCHY product_hier;
```

DROP INDEX

Purpose

Use the `DROP INDEX` statement to remove an index or domain index from the database.

When you drop a global partitioned index, a range-partitioned index, or a hash-partitioned index, all the index partitions are also dropped. If you drop a composite-partitioned index, then all the index partitions and subpartitions are also dropped.

In addition, when you drop a domain index:

- Oracle Database invokes the appropriate routine.
- If any statistics are associated with the domain index, then Oracle Database disassociates the statistics types with the `FORCE` clause and removes the user-defined statistics collected with the statistics type.

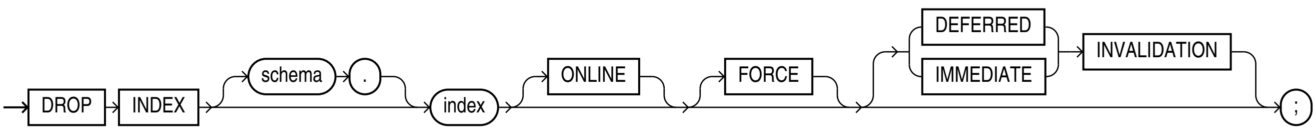
> **See Also:**
>
> – *Oracle Database Data Cartridge Developer's Guide* for information on the routines
>
> – CREATE INDEX and ALTER INDEX for information on creating and modifying an index
>
> – The *`domain_index_clause`* of CREATE INDEX for more information on domain indexes
>
> – ASSOCIATE STATISTICS and DISASSOCIATE STATISTICS for more information on statistics type associations

Prerequisites

The index must be in your own schema or you must have the `DROP ANY INDEX` system privilege.

Syntax

***drop_index*::=**



Semantics

schema

Specify the schema containing the index. If you omit *schema*, then Oracle Database assumes the index is in your own schema.

index

Specify the name of the index to be dropped. When the index is dropped, all data blocks allocated to the index are returned to the tablespace that contained the index.

ONLINE

Specify ONLINE to indicate that DML operations on the table or partition will be allowed while dropping the index.

FORCE

FORCE applies only to domain indexes. This clause drops the domain index even if the indextype routine invocation returns an error or the index is marked IN PROGRESS. Without FORCE, you cannot drop a domain index if its indextype routine invocation returns an error or the index is marked IN PROGRESS.

{ DEFERRED | IMMEDIATE } INVALIDATION

This clause lets you control when the database invalidates dependent cursors while dropping the index. It has the same semantics here as for the ALTER INDEX statement, with the following addition: When you drop an index with DEFERRED INVALIDATION, Oracle database will immediately invalidate any DML statement or query that references the dropped index in its plan.

See { DEFERRED | IMMEDIATE } INVALIDATION in the documentation on ALTER INDEX for the full semantics of this clause.

Restrictions on Dropping Indexes

The following restrictions apply to dropping indexes:

- You cannot drop a domain index if the index or any of its index partitions is marked IN_PROGRESS.
- You cannot specify the ONLINE clause when dropping a domain index, a cluster index, or an index on a queue table.

Examples

Dropping an Index: Example

This statement drops an index named `ord_customer_ix_demo`, which was created in "Compressing an Index: Example":

```
DROP INDEX ord_customer_ix_demo;
```

DROP INDEXTYPE

Purpose

Use the `DROP INDEXTYPE` statement to drop an indextype as well as any association with a statistics type.

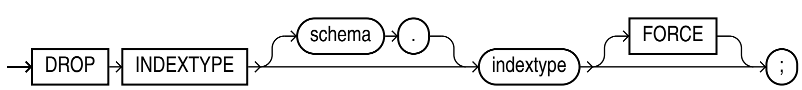
> **See Also:**
>
> CREATE INDEXTYPE for more information on indextypes

Prerequisites

The indextype must be in your own schema or you must have the `DROP ANY INDEXTYPE` system privilege.

Syntax

***drop_indextype*::=**



Semantics

schema

Specify the schema containing the indextype. If you omit *`schema`*, then Oracle Database assumes the indextype is in your own schema.

indextype

Specify the name of the indextype to be dropped.

If any statistics types have been associated with indextype, then the database disassociates the statistics type from the indextype and drops any statistics that have been collected using the statistics type.

> **See Also:**
>
> ASSOCIATE STATISTICS and DISASSOCIATE STATISTICS for more information on statistics associations

FORCE

Specify `FORCE` to drop the indextype even if the indextype is currently being referenced by one or more domain indexes. Oracle Database marks those domain indexes `INVALID`. Without `FORCE`, you cannot drop an indextype if any domain indexes reference the indextype.

Examples

Dropping an Indextype: Example

The following statement drops the indextype `position_indextype`, created in "Using Extensible Indexing ", and marks `INVALID` any domain indexes defined on this indextype:

```
DROP INDEXTYPE position_indextype FORCE;
```

DROP INMEMORY JOIN GROUP

Purpose

Use the `DROP INMEMORY JOIN GROUP` statement to remove a join group from the database.

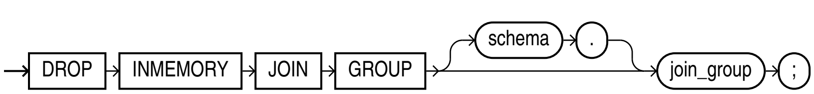
> **See Also:**
>
> - CREATE INMEMORY JOIN GROUP and ALTER INMEMORY JOIN GROUP
> - *Oracle Database In-Memory Guide* for more information on join groups

Prerequisites

If the join group is in another user's schema, then you must have the `DROP ANY TABLE` system privilege.

Syntax

***drop_inmemory_join_group*::=**



Semantics

schema

Specify the schema containing the join group. If you omit `schema`, then the database assumes the join group is in your own schema.

join_group

Specify the name of the join group to be dropped.

You can view existing join groups by querying the `DBA_JOINGROUPS` or `USER_JOINGROUPS` data dictionary view. Refer to *Oracle Database Reference* for more information on these views.

Examples

The following statement drops the join group `prod_id1`:

```
DROP INMEMORY JOIN GROUP prod_id1;
```

DROP JAVA

Purpose

Use the `DROP JAVA` statement to drop a Java source, class, or resource schema object.

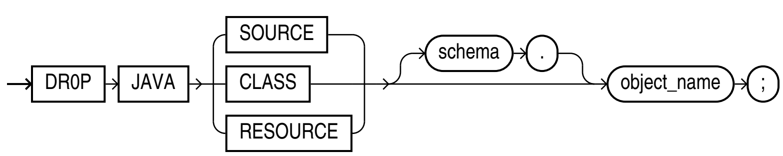
> **See Also:**
>
> - CREATE JAVA for information on creating Java objects
> - *Oracle Database Java Developer's Guide* for more information on resolving Java sources, classes, and resources

Prerequisites

The Java source, class, or resource must be in your own schema or you must have the `DROP ANY PROCEDURE` system privilege. You also must have the `EXECUTE` object privilege on Java classes to use this command.

Syntax

drop_java::=



Semantics

JAVA SOURCE

Specify `SOURCE` to drop a Java source schema object and all Java class schema objects derived from it.

JAVA CLASS

Specify `CLASS` to drop a Java class schema object.

JAVA RESOURCE

Specify `RESOURCE` to drop a Java resource schema object.

object_name

Specify the name of an existing Java class, source, or resource schema object. Enclose the *object_name* in double quotation marks to preserve lower- or mixed-case names.

Examples

Dropping a Java Class Object: Example

The following statement drops the Java class `Agent`, created in "Creating a Java Class Object: Example":

```
DROP JAVA CLASS "Agent";
```

17

SQL Statements: DROP LIBRARY to DROP SYNONYM

This chapter contains the following SQL statements:

- DROP LIBRARY
- DROP LOCKDOWN PROFILE
- DROP MATERIALIZED VIEW
- DROP MATERIALIZED VIEW LOG
- DROP MATERIALIZED ZONEMAP
- DROP OPERATOR
- DROP OUTLINE
- DROP PACKAGE
- DROP PLUGGABLE DATABASE
- DROP PROCEDURE
- DROP PROFILE
- DROP RESTORE POINT
- DROP ROLE
- DROP ROLLBACK SEGMENT
- DROP SEQUENCE
- DROP SYNONYM

DROP LIBRARY

Purpose

Use the `DROP LIBRARY` statement to remove an external procedure library from the database.

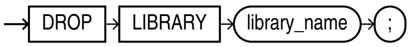
> **See Also:**
>
> CREATE LIBRARY for information on creating a library

Prerequisites

You must have the `DROP ANY LIBRARY` system privilege.

Syntax

***drop_library*::=**



Semantics

library_name

Specify the name of the external procedure library being dropped.

Examples

Dropping a Library: Example

The following statement drops the `ext_lib` library:

```
DROP LIBRARY ext_lib;
```

DROP LOCKDOWN PROFILE

Purpose

Use the `DROP LOCKDOWN PROFILE` statement to remove a PDB lockdown profile from the database.

If the lockdown profile is the value of the `PDB_LOCKDOWN` initialization parameter for a CDB, an application root, or a PDB, then the effect of the lockdown profile will be disabled when you drop it. However, the lockdown profile will remain as the value of the `PDB_LOCKDOWN` initialization parameter. You can subsequently modify the `PDB_LOCKDOWN` parameter to unset the value of the dropped lockdown profile.

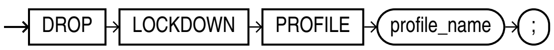
> **See Also:**
>
> - CREATE LOCKDOWN PROFILE and ALTER LOCKDOWN PROFILE
> - *Oracle Database Security Guide* for more information on PDB lockdown profiles

Prerequisites

The `DROP LOCKDOWN PROFILE` statement is valid only in a CDB. The current container must be the CDB root and you must have the `DROP LOCKDOWN PROFILE` system privilege, either granted commonly or granted locally in the CDB root.

Syntax

***drop_lockdown_profile*::=**



Semantics

profile_name

Specify the name of the PDB lockdown profile to be dropped.

You can find the names of existing PDB lockdown profiles by querying the `DBA_LOCKDOWN_PROFILES` data dictionary view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the `DBA_LOCKDOWN_PROFILES` data dictionary view and the `PDB_LOCKDOWN` initialization parameter

Example

The following statement drops PDB lockdown profile `hr_prof`:

```
DROP LOCKDOWN PROFILE hr_prof;
```

DROP MATERIALIZED VIEW

Purpose

Use the `DROP MATERIALIZED VIEW` statement to remove an existing materialized view from the database.

When you drop a materialized view, Oracle Database does not place it in the recycle bin. Therefore, you cannot subsequently either purge or undrop the materialized view.

> **Note:**
>
> The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

> **See Also:**
>
> - CREATE MATERIALIZED VIEW for more information on the various types of materialized views
> - ALTER MATERIALIZED VIEW for information on modifying a materialized view
> - *Oracle Database Administrator's Guide* for information on materialized views in a replication environment
> - *Oracle Database Data Warehousing Guide* for information on materialized views in a data warehousing environment

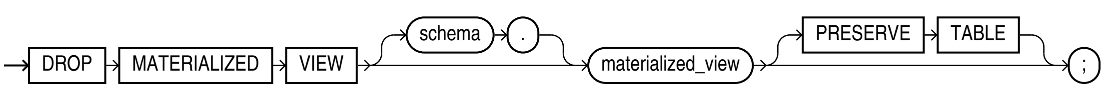
Prerequisites

The materialized view must be in your own schema or you must have the `DROP ANY MATERIALIZED VIEW` system privilege. You must also have the privileges to drop the internal table, views, and index that the database uses to maintain the materialized view data.

> **See Also:**
>
> DROP TABLE , DROP VIEW , and DROP INDEX for information on privileges required to drop objects that the database uses to maintain the materialized view

Syntax

***drop_materialized_view*::=**



Semantics

schema

Specify the schema containing the materialized view. If you omit *`schema`*, then Oracle Database assumes the materialized view is in your own schema.

materialized_view

Specify the name of the existing materialized view to be dropped.

- If you drop a simple materialized view that is the least recently refreshed materialized view of a master table, then the database automatically purges from the master table materialized view log only the rows needed to refresh the dropped materialized view.

- If you drop a materialized view that was created on a prebuilt table, then the database drops the materialized view, and the prebuilt table reverts to its identity as a table.
- When you drop a master table, the database does not automatically drop materialized views based on the table. However, the database returns an error when it tries to refresh a materialized view based on a master table that has been dropped.
- If you drop a materialized view, then any compiled requests that were rewritten to use the materialized view will be invalidated and recompiled automatically. If the materialized view was prebuilt on a table, then the table is not dropped, but it can no longer be maintained by the materialized view refresh mechanism.

PRESERVE TABLE Clause

This clause lets you retain the materialized view container table and its contents after the materialized view object is dropped. The resulting table has the same name as the dropped materialized view.

Oracle Database removes all metadata associated with the materialized view. However, indexes created on the container table automatically during creation of the materialized view are preserved, with one exception: the index created during the creation of a rowid materialized view is dropped. Also, if the materialized view has any nested table columns, then the storage tables for those columns are preserved, along with their metadata.

Restriction on the PRESERVE TABLE Clause

This clause is not valid for materialized views that have been imported from releases earlier than Oracle9*i*, when these objects were called "snapshots".

Examples

Dropping a Materialized View: Examples

The following statement drops the materialized view `emp_data` in the sample schema `hr`:

```
DROP MATERIALIZED VIEW emp_data;
```

The following statement drops the `sales_by_month_by_state` materialized view and the underlying table of the materialized view, unless the underlying table was registered in the `CREATE MATERIALIZED VIEW` statement with the `ON PREBUILT TABLE` clause:

```
DROP MATERIALIZED VIEW sales_by_month_by_state;
```

DROP MATERIALIZED VIEW LOG

Purpose

Use the `DROP MATERIALIZED VIEW LOG` statement to remove a materialized view log from the database.

> **Note:**
>
> The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

See Also:

- CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW for more information on materialized views
- CREATE MATERIALIZED VIEW LOG for information on materialized view logs
- *Oracle Database Administrator's Guide* for information on materialized views in a replication environment
- *Oracle Database Data Warehousing Guide* for information on materialized views in a data warehousing environment

Prerequisites

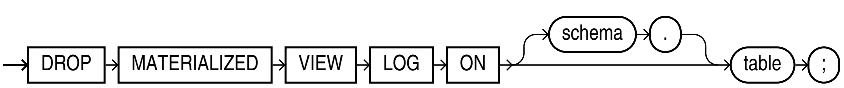
To drop a materialized view log, you must have the privileges needed to drop a table.

See Also:

DROP TABLE

Syntax

***drop_materialized_view_log*::=**



Semantics

schema

Specify the schema containing the materialized view log and its master table. If you omit `schema`, then Oracle Database assumes the materialized view log and master table are in your own schema.

table

Specify the name of the master table associated with the materialized view log to be dropped.

After you drop a materialized view log that was created `FOR FAST REFRESH`, some materialized views based on the materialized view log master table can no longer be fast refreshed. These materialized views include rowid materialized views, primary key materialized views, and subquery materialized views.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for a description of these types of materialized views

After you drop a materialized view log that was created `FOR SYNCHRONOUS REFRESH` (a staging log), the materialized views based on the staging log master table can no longer be synchronous refreshed.

Examples

Dropping a Materialized View Log: Example

The following statement drops the materialized view log on the `oe.customers` master table:

```
DROP MATERIALIZED VIEW LOG ON customers;
```

DROP MATERIALIZED ZONEMAP

Purpose

Use the `DROP MATERIALIZED ZONEMAP` statement to remove an existing zone map from the database.

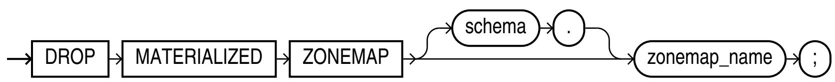
Prerequisites

The zone map must be in your own schema or you must have the `DROP ANY MATERIALIZED VIEW` system privilege. You must also have the privileges to drop the internal table and indexes that the database uses to maintain the zone map data.

> **See Also:**
>
> DROP TABLE and DROP INDEX for information on privileges required to drop objects that the database uses to maintain the zone map

Syntax

***drop_materialized_zonemap*::=**



Semantics

schema

Specify the schema containing the zone map. If you omit *`schema`*, then Oracle Database assumes the zone map is in your own schema.

zonemap_name

Specify the name of the existing zone map to be dropped.

Example

Dropping a Zone Map: Examples

The following statement drops the zone map `sales_zmap`:

```
DROP MATERIALIZED ZONEMAP sales_zmap;
```

DROP OPERATOR

Purpose

Use the `DROP OPERATOR` statement to drop a user-defined operator.

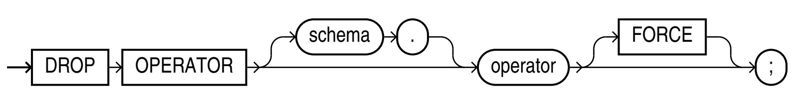
> **See Also:**
>
> - CREATE OPERATOR and ALTER OPERATOR for information on creating and modifying operators
> - "User-Defined Operators " and *Oracle Database Data Cartridge Developer's Guide* for more information on operators in general
> - ALTER INDEXTYPE for information on dropping an operator of a user-defined indextype

Prerequisites

The operator must be in your schema or you must have the `DROP ANY OPERATOR` system privilege.

Syntax

drop_operator::=



Semantics

schema

Specify the schema containing the operator. If you omit *`schema`*, then Oracle Database assumes the operator is in your own schema.

operator

Specify the name of the operator to be dropped.

FORCE

Specify FORCE to drop the operator even if it is currently being referenced by one or more schema objects, such as indextypes, packages, functions, procedures, and so on. The database marks any such dependent objects INVALID. Without FORCE, you cannot drop an operator if any schema objects reference it.

Examples

Dropping a User-Defined Operator: Example

The following statement drops the operator eq_op:

```
DROP OPERATOR eq_op;
```

Because the FORCE clause is not specified, this operation will fail if any of the bindings of this operator are referenced by an indextype.

DROP OUTLINE

Purpose

> **Note:**
>
> - Stored outlines are deprecated. They are still supported for backward compatibility. However, Oracle recommends that you use SQL plan management instead. SQL plan management creates SQL plan baselines, which offer superior SQL performance stability compared with stored outlines.
> - You can migrate existing stored outlines to SQL plan baselines by using the MIGRATE_STORED_OUTLINE function of the DBMS_SPM package or Enterprise Manager Cloud Control. When the migration is complete, the stored outlines are marked as migrated and can be removed. You can drop all migrated stored outlines on your system by using the DROP_MIGRATED_STORED_OUTLINE function of the DBMS_SPM package.
> - **See Also:** *Oracle Database SQL Tuning Guide* for more information about SQL plan management and *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_SPM package

Use the DROP OUTLINE statement to drop a stored outline.

> **See Also:**
>
> CREATE OUTLINE for information on creating an outline

Prerequisites

To drop an outline, you must have the DROP ANY OUTLINE system privilege.

Syntax

***drop_outline*::=**

Semantics

outline

Specify the name of the outline to be dropped.

After the outline is dropped, if the SQL statement for which the stored outline was created is compiled, then the optimizer generates a new execution plan without the influence of the outline.

Examples

Dropping an Outline: Example

The following statement drops the stored outline called `salaries`.

```
DROP OUTLINE salaries;
```

DROP PACKAGE

Purpose

Packages are defined using PL/SQL. Refer to *Oracle Database PL/SQL Language Reference* for complete information on creating, altering, and dropping packages.

Use the `DROP PACKAGE` statement to remove a stored package from the database. This statement drops the body and specification of a package.

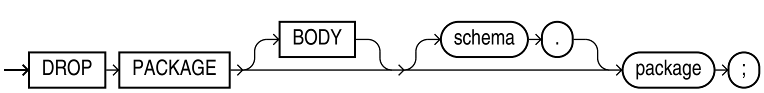
> **Note:**
>
> Do not use this statement to remove a single object from a package. Instead, re-create the package without the object using the `CREATE PACKAGE` and `CREATE PACKAGE BODY` statements with the `OR REPLACE` clause.

Prerequisites

The package must be in your own schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

***drop_package*::=**



Semantics

BODY

Specify `BODY` to drop only the body of the package. If you omit this clause, then Oracle Database drops both the body and specification of the package.

When you drop only the body of a package but not its specification, the database does not invalidate dependent objects. However, you cannot call one of the procedures or stored functions declared in the package specification until you re-create the package body.

schema

Specify the schema containing the package. If you omit *`schema`*, then the database assumes the package is in your own schema.

package

Specify the name of the package to be dropped.

Oracle Database invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, then the database disassociates the statistics types with the `FORCE` clause and drops any user-defined statistics collected with the statistics types.

> **See Also:**
>
> ASSOCIATE STATISTICS and DISASSOCIATE STATISTICS

Examples

Dropping a Package: Example

The following statement drops the specification and body of the `emp_mgmt` package, invalidating all objects that depend on the specification. See *Oracle Database PL/SQL Language Reference* for the example that creates this package.

```
DROP PACKAGE emp_mgmt;
```

DROP PLUGGABLE DATABASE

Purpose

Use the `DROP PLUGGABLE DATABASE` statement to drop a pluggable database (PDB). The PDB can be a traditional PDB, an application container, an application seed, or an application PDB.

When you drop a PDB, the control file of the multitenant container database (CDB) is modified to remove all references to the dropped PDB and its data files. Archived logs and backups associated with the dropped PDB are not deleted. You can delete them using Oracle

Recovery Manager (RMAN), or you can retain them in case you subsequently want to perform point-in-time recovery of the PDB.

> **Caution:**
>
> You cannot roll back a `DROP PLUGGABLE DATABASE` statement.

Prerequisites

You must be connected to a CDB.

To drop a traditional PDB or an application container, the current container must be the root, you must be authenticated `AS SYSDBA` or `AS SYSOPER`, and the `SYSDBA` or `SYSOPER` privilege must be either granted to you commonly, or granted to you locally in the root and locally in traditional PDB or application container you want to drop. The application container must be empty, that is, it must not contain an application seed or any application PDBs.

To drop an application seed, the current container must be the root or the application root, you must be authenticated `AS SYSDBA` or `AS SYSOPER`, and the `SYSDBA` or `SYSOPER` privilege must be either granted to you commonly, or granted to you locally in the root or application root.

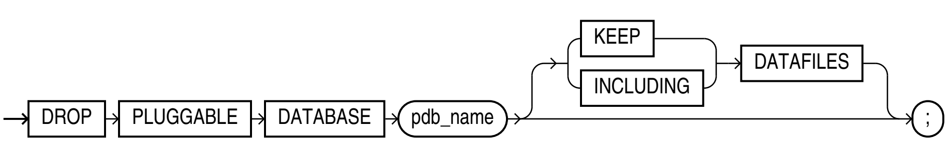
To drop an application PDB, the current container must be the root or the application root, you must be authenticated `AS SYSDBA` or `AS SYSOPER`, and the `SYSDBA` or `SYSOPER` privilege must be either granted to you commonly, or granted to you locally in the root or application root, and locally in the application PDB you want to drop.

To specify `KEEP DATAFILES` (the default), the PDB you want to drop must be unplugged.

To specify `INCLUDING DATAFILES`, the PDB you want to drop must be in mounted mode or it must be unplugged.

Syntax

drop_pluggable_database::=



Semantics

pdb_name

Specify the name of the PDB you want to drop. You cannot drop the seed (`PDB$SEED`). However, you can drop an application seed.

KEEP DATAFILES

Specify `KEEP DATAFILES` to retain the data files associated with the PDB after the PDB is dropped. The temp file for the PDB is deleted because it is no longer needed. This is the default.

Keeping data files may be useful in scenarios where a PDB that is unplugged from one CDB is plugged into another CDB, with both CDBs sharing storage devices.

INCLUDING DATAFILES

Specify `INCLUDING DATAFILES` to delete the data files associated with the PDB being dropped. The temp file for the PDB is also deleted.

Restriction on Dropping SNAPSHOT COPY PDBs

If a PDB was created with the `SNAPSHOT COPY` clause, then you must specify `INCLUDING DATAFILES` when you drop the PDB.

Examples

Dropping a PDB: Example

The following statement drops the PDB `pdb1` and its associated data files:

```
DROP PLUGGABLE DATABASE pdb1
  INCLUDING DATAFILES;
```

DROP PROCEDURE

Purpose

Procedures are defined using PL/SQL. Refer to *Oracle Database PL/SQL Language Reference* for complete information on creating, altering, and dropping procedures.

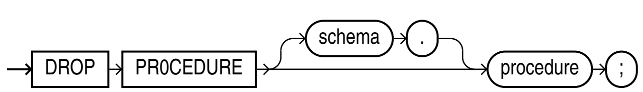
Use the `DROP PROCEDURE` statement to remove a standalone stored procedure from the database. Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the `DROP PACKAGE` statement, or redefine the package without the procedure using the `CREATE PACKAGE` statement with the `OR REPLACE` clause.

Prerequisites

The procedure must be in your own schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

***drop_procedure*::=**



Semantics

schema

Specify the schema containing the procedure. If you omit *schema*, then Oracle Database assumes the procedure is in your own schema.

procedure

Specify the name of the procedure to be dropped.

When you drop a procedure, Oracle Database invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

Examples

Dropping a Procedure: Example

The following statement drops the procedure `remove_emp` owned by the user `hr` and invalidates all objects that depend upon `remove_emp`:

```
DROP PROCEDURE hr.remove_emp;
```

DROP PROFILE

Purpose

Use the `DROP PROFILE` statement to remove a profile from the database. You can drop any profile except the `DEFAULT` profile.

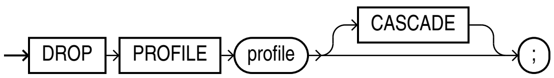
> **See Also:**
>
> CREATE PROFILE and ALTER PROFILE on creating and modifying a profile

Prerequisites

You must have the `DROP PROFILE` system privilege.

Syntax

drop_profile::=



Semantics

profile

Specify the name of the profile to be dropped.

CASCADE

Specify `CASCADE` to deassign the profile from any users to whom it is assigned. Oracle Database automatically assigns the `DEFAULT` profile to such users. You must specify this clause to drop a profile that is currently assigned to users.

Examples

Dropping a Profile: Example

The following statement drops the profile `app_user`, which was created in "Creating a Profile: Example". Oracle Database drops the profile `app_user` and assigns the `DEFAULT` profile to any users currently assigned the `app_user` profile:

```
DROP PROFILE app_user CASCADE;
```

DROP RESTORE POINT

Purpose

Use the `DROP RESTORE POINT` statement to remove a normal restore point or a guaranteed restore point from the database.

- You need not drop normal restore points. The database automatically drops the oldest restore points when necessary, as described in the semantics for *restore_point*. However, you can drop a normal restore point if you want to reuse the name.
- Guaranteed restore points are not dropped automatically. Therefore, if you want to remove a guaranteed restore point from the database, then you must do so explicitly using this statement.

> **See Also:**
>
> CREATE RESTORE POINT , FLASHBACK DATABASE, and FLASHBACK TABLE for information on creating and using restore points

Prerequisites

To drop a normal restore point, you must have the `SELECT ANY DICTIONARY`, `FLASHBACK ANY TABLE`, `SYSBACKUP`, or `SYSDG` system privilege. To drop a guaranteed restore point, you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege.

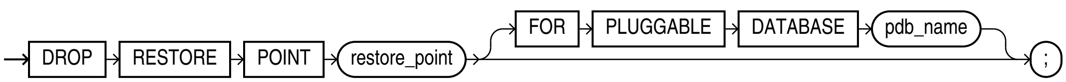
You can drop a restore point when connected to a multitenant container database (CDB) as follows:

- To drop a normal CDB restore point, the current container must be the root and you must have the `SELECT ANY DICTIONARY` or `FLASHBACK ANY TABLE` system privilege, either granted commonly or granted locally in the root, or the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege granted commonly.
- To drop a guaranteed CDB restore point, the current container must be the root and you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege granted commonly.

- To drop a normal PDB restore point, the current container must be the root and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly, or the current container must be the PDB in which you want to create the restore point and you must have the SELECT ANY DICTIONARY, FLASHBACK ANY TABLE, SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.
- To drop a guaranteed PDB restore point, the current container must be the root and you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly, or the current container must be the PDB in which you want to create the restore point and you must have the SYSDBA, SYSBACKUP, or SYSDG system privilege, granted commonly or granted locally in that PDB.

Syntax

***drop_restore_point*::=**



Semantics

restore_point

Specify the name of the restore point you want to drop.

FOR PLUGGABLE DATABASE

This clause enables you to drop a PDB restore point when you are connected to the root. For *pdb_name*, specify the name of the PDB that contains the restore point you want to drop.

If you are connected to the PDB from which you want to drop the restore point, then it is not necessary to specify this clause. However, if you specify this clause, then you must specify the name of the PDB to which you are connected.

Examples

Dropping a Restore Point: Example

The following example drops the good_data restore point, which was created in "Creating and Using a Restore Point: Example":

```
DROP RESTORE POINT good_data;
```

DROP ROLE

Purpose

Use the DROP ROLE statement to remove a role from the database. When you drop a role, Oracle Database revokes it from all users and roles to whom it has been granted and removes it from the database. User sessions in which the role is already enabled are not affected. However, no new user session can enable the role after it is dropped.

> **See Also:**
>
> - CREATE ROLE and ALTER ROLE for information on creating roles and changing the authorization needed to enable a role
> - SET ROLE for information on disabling roles for the current session

Prerequisites

You must have been granted the role with the `ADMIN OPTION` or you must have the `DROP ANY ROLE` system privilege.

Syntax

drop_role::=

Semantics

role

Specify the name of the role to be dropped.

Examples

Dropping a Role: Example

To drop the role `dw_manager`, which was created in "Creating a Role: Example", issue the following statement:

```
DROP ROLE dw_manager;
```

DROP ROLLBACK SEGMENT

Purpose

Use the `DROP ROLLBACK SEGMENT` to remove a rollback segment from the database. When you drop a rollback segment, all space allocated to the rollback segment returns to the tablespace.

> **Note:**
>
> If your database is running in automatic undo mode, then this is the only valid operation on rollback segments. In that mode, you cannot create or alter a rollback segment.

Prerequisites

You must have the `DROP ROLLBACK SEGMENT` system privilege, and the rollback segment must be offline.

Syntax

***drop_rollback_segment*::=**

Semantics

rollback_segment

Specify the name the rollback segment to be dropped.

Restrictions on Dropping Rollback Segments

This statement is subject to the following restrictions:

- You can drop a rollback segment only if it is offline. To determine whether a rollback segment is offline, query the data dictionary view `DBA_ROLLBACK_SEGS`. Offline rollback segments have the value `AVAILABLE` in the `STATUS` column. You can take a rollback segment offline with the `OFFLINE` clause of the `ALTER ROLLBACK SEGMENT` statement.
- You cannot drop the `SYSTEM` rollback segment.

Examples

Dropping a Rollback Segment: Example

The following syntax drops the rollback segment created in "Creating a Rollback Segment: Example":

```
DROP ROLLBACK SEGMENT rbs_one;
```

DROP SEQUENCE

Purpose

Use the `DROP SEQUENCE` statement to remove a sequence from the database.

You can also use this statement to restart a sequence by dropping and then re-creating it. For example, if you have a sequence with a current value of 150 and you would like to restart the sequence with a value of 27, then you can drop the sequence and then re-create it with the same name and a `START WITH` value of 27.

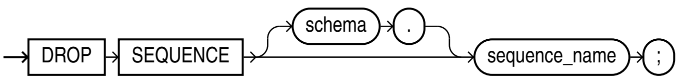
> **See Also:**
>
> CREATE SEQUENCE and ALTER SEQUENCE for more information on creating and modifying a sequence

Prerequisites

The sequence must be in your own schema or you must have the DROP ANY SEQUENCE system privilege.

Syntax

***drop_sequence*::=**



Semantics

schema

Specify the schema containing the sequence. If you omit *schema*, then Oracle Database assumes the sequence is in your own schema.

sequence_name

Specify the name of the sequence to be dropped.

Examples

Dropping a Sequence: Example

The following statement drops the sequence customers_seq owned by the user oe, which was created in "Creating a Sequence: Example". To issue this statement, you must either be connected as user oe or have the DROP ANY SEQUENCE system privilege:

```
DROP SEQUENCE oe.customers_seq;
```

DROP SYNONYM

Purpose

Use the DROP SYNONYM statement to remove a synonym from the database or to change the definition of a synonym by dropping and re-creating it.

> **See Also:**
>
> CREATE SYNONYM for more information on synonyms

Prerequisites

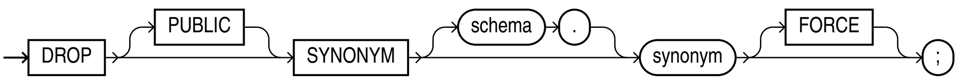
To drop a private synonym, either the synonym must be in your own schema or you must have the DROP ANY SYNONYM system privilege.

To drop a PUBLIC synonym, you must have the DROP PUBLIC SYNONYM system privilege.

Syntax

***drop_synonym*::=**

drop_synonym::=



Semantics

PUBLIC

You must specify PUBLIC to drop a public synonym. You cannot specify *schema* if you have specified PUBLIC.

schema

Specify the schema containing the synonym. If you omit *schema*, then Oracle Database assumes the synonym is in your own schema.

synonym

Specify the name of the synonym to be dropped.

If you drop a synonym for the master table of a materialized view, and if the defining query of the materialized view specified the synonym rather than the actual table name, then Oracle Database marks the materialized view unusable.

If an object type synonym has any dependent tables or user-defined types, then you cannot drop the synonym unless you also specify FORCE.

FORCE

Specify FORCE to drop the synonym even if it has dependent tables or user-defined types.

> **Note:**
>
> Oracle does not recommend that you specify FORCE to drop object type synonyms with dependencies. This operation can result in invalidation of other user-defined types or marking UNUSED the table columns that depend on the synonym. For information about type dependencies, see *Oracle Database Object-Relational Developer's Guide*.

Examples

Dropping a Synonym: Example

To drop the public synonym named `customers`, which was created in "Oracle Database Resolution of Synonyms: Example", issue the following statement:

```
DROP PUBLIC SYNONYM customers;
```

18

SQL Statements: DROP TABLE to LOCK TABLE

This chapter contains the following SQL statements:

- DROP TABLE
- DROP TABLESPACE
- DROP TABLESPACE SET
- DROP TRIGGER
- DROP TYPE
- DROP TYPE BODY
- DROP USER
- DROP VIEW
- EXPLAIN PLAN
- FLASHBACK DATABASE
- FLASHBACK TABLE
- GRANT
- INSERT
- LOCK TABLE

DROP TABLE

Purpose

Use the `DROP TABLE` statement to move a table or object table to the recycle bin or to remove the table and all its data from the database entirely.

> **Note:**
>
> Unless you specify the `PURGE` clause, the `DROP TABLE` statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count toward the user's space quota.

For an external table, this statement removes only the table metadata in the database. It has no affect on the actual data, which resides outside of the database.

When you drop a table that is part of a cluster, the table is moved to the recycle bin. However, if you subsequently drop the cluster, then the table is purged from the recycle bin and can no longer be recovered with a `FLASHBACK TABLE` operation.

Dropping a table invalidates dependent objects and removes object privileges on the table. If you want to re-create the table, then you must regrant object privileges on the table, re-create the indexes, integrity constraints, and triggers for the table, and respecify its storage parameters. Truncating has none of these effects. Therefore, removing rows with the `TRUNCATE` statement can be more efficient than dropping and re-creating a table.

> **See Also:**
>
> - CREATE TABLE and ALTER TABLE for information on creating and modifying tables
> - TRUNCATE TABLE and DELETE for information on removing data from a table
> - FLASHBACK TABLE for information on retrieving a dropped table from the recycle bin

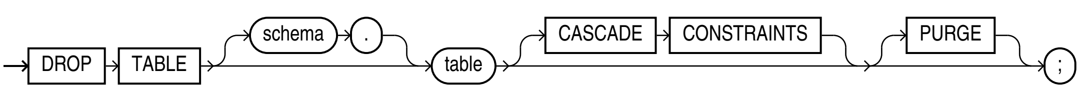
Prerequisites

The table must be in your own schema or you must have the `DROP ANY TABLE` system privilege.

You can perform DDL operations (such as `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`) on a temporary table only when no session is bound to it. A session becomes bound to a temporary table by performing an `INSERT` operation on the table. A session becomes unbound to the temporary table by issuing a `TRUNCATE` statement or at session termination, or, for a transaction-specific temporary table, by issuing a `COMMIT` or `ROLLBACK` statement.

Syntax

***drop_table*::=**



Semantics

schema

Specify the schema containing the table. If you omit *`schema`*, then Oracle Database assumes the table is in your own schema.

table

Specify the name of the table to be dropped. Oracle Database automatically performs the following operations:

- All rows from the table are dropped.
- All table indexes and domain indexes are dropped, as well as any triggers defined on the table, regardless of who created them or whose schema contains them. If

table is partitioned, then any corresponding local index partitions are also dropped.

- All the storage tables of nested tables and LOBs of *table* are dropped.
- When you drop a range-, hash-, or list-partitioned table, then the database drops all the table partitions. If you drop a composite-partitioned table, then all the partitions and subpartitions are also dropped.
- When you drop a partitioned table with the PURGE keyword, the statement executes as a series of subtransactions, each of which drops a subset of partitions or subpartitions and their metadata. This division of the drop operation into subtransactions optimizes the processing of internal system resource consumption (for example, the library cache), especially for the dropping of very large partitioned tables. As soon as the first subtransaction commits, the table is marked UNUSABLE. If any of the subtransactions fails, then the only operation allowed on the table is another DROP TABLE ... PURGE statement. Such a statement will resume work from where the previous DROP TABLE statement failed, assuming that you have corrected any errors that the previous operation encountered.

  You can list the tables marked UNUSABLE by such a drop operation by querying the status column of the *_TABLES, *_PART_TABLES, *_ALL_TABLES, or *_OBJECT_TABLES data dictionary views, as appropriate.

  > **See Also:**
  >
  > *Oracle Database VLDB and Partitioning Guide* for more information on dropping partitioned tables.

- For an index-organized table, any mapping tables defined on the index-organized table are dropped.
- For a domain index, the appropriate drop routines are invoked. Refer to *Oracle Database Data Cartridge Developer's Guide* for more information on these routines.
- If any statistics types are associated with the table, then the database disassociates the statistics types with the FORCE clause and removes any user-defined statistics collected with the statistics type.

  > **See Also:**
  >
  > ASSOCIATE STATISTICS and DISASSOCIATE STATISTICS for more information on statistics type associations

- If the table is not part of a cluster, then the database returns all data blocks allocated to the table and its indexes to the tablespaces containing the table and its indexes.

  To drop a cluster and all its the tables, use the DROP CLUSTER statement with the INCLUDING TABLES clause to avoid dropping each table individually. See DROP CLUSTER .

- If the table is a base table for a view, a container or master table of a materialized view, or if it is referenced in a stored procedure, function, or package, then the database invalidates these dependent objects but does not drop them. You cannot use these objects unless you re-create the table or drop and re-create the objects so that they no longer depend on the table.

If you choose to re-create the table, then it must contain all the columns selected by the subqueries originally used to define the materialized views and all the columns referenced in the stored procedures, functions, or packages. Any users previously granted object privileges on the views, stored procedures, functions, or packages need not be regranted these privileges.

If the table is a master table for a materialized view, then the materialized view can still be queried, but it cannot be refreshed unless the table is re-created so that it contains all the columns selected by the defining query of the materialized view.

If the table has a materialized view log, then the database drops this log and any other direct-path `INSERT` refresh information associated with the table.

Restrictions on Dropping Tables

- You cannot directly drop the storage table of a nested table. Instead, you must drop the nested table column using the `ALTER TABLE` ... `DROP COLUMN` clause.
- You cannot drop the parent table of a reference-partitioned table. You must first drop all reference-partitioned child tables.
- You cannot drop a table that uses a flashback data archive for historical tracking. You must first disable the table's use of the flashback data archive.

CASCADE CONSTRAINTS

Specify `CASCADE CONSTRAINTS` to drop all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this clause, and such referential integrity constraints exist, then the database returns an error and does not drop the table.

PURGE

Specify `PURGE` if you want to drop the table and release the space associated with it in a single step. If you specify `PURGE`, then the database does not place the table and its dependent objects into the recycle bin.

> **Note:**
>
> You cannot roll back a `DROP TABLE` statement with the `PURGE` clause, nor can you recover the table if you have dropped it with the `PURGE` clause.

Using this clause is equivalent to first dropping the table and then purging it from the recycle bin. This clause lets you save one step in the process. It also provides enhanced security if you want to prevent sensitive material from appearing in the recycle bin.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on the recycle bin and naming conventions for objects in the recycle bin

Examples

Dropping a Table: Example

The following statement drops the `oe.list_customers` table created in "List Partitioning Example".

```
DROP TABLE list_customers PURGE;
```

DROP TABLESPACE

Purpose

Use the `DROP TABLESPACE` statement to remove a tablespace from the database.

When you drop a tablespace, Oracle Database does not place it in the recycle bin. Therefore, you cannot subsequently either purge or undrop the tablespace.

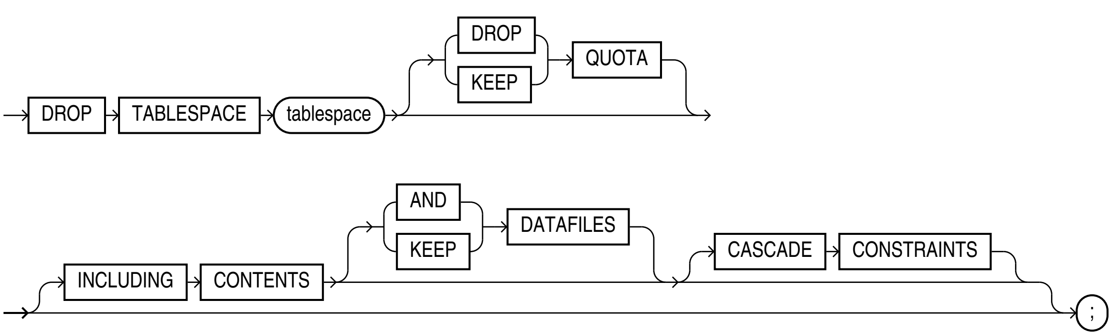
> **See Also:**
>
> CREATE TABLESPACE and ALTER TABLESPACE for information on creating and modifying a tablespace

Prerequisites

You must have the `DROP TABLESPACE` system privilege. You cannot drop a tablespace if it contains any rollback segments holding active transactions.

Syntax

***drop_tablespace*::=**



Semantics

tablespace

Specify the name of the tablespace to be dropped.

You can drop a tablespace regardless of whether it is online or offline. Oracle recommends that you take the tablespace offline before dropping it to ensure that no SQL statements in currently running transactions access any of the objects in the tablespace.

You cannot drop the `SYSTEM` tablespace. You can drop the `SYSAUX` tablespace only if you have the `SYSDBA` system privilege and you have started the database in `UPGRADE` mode.

You may want to alert any users who have been assigned the tablespace as either a default or temporary tablespace. After the tablespace has been dropped, these users cannot allocate space for objects or sort areas in the tablespace. You can reassign users new default and temporary tablespaces with the `ALTER USER` statement.

Any objects that were previously dropped from the tablespace and moved to the recycle bin are purged from the recycle bin. Oracle Database removes from the data dictionary all metadata about the tablespace and all data files and temp files in the tablespace. The database also automatically drops from the operating system any Oracle-managed data files and temp files in the tablespace. Other data files and temp files are not removed from the operating system unless you specify `INCLUDING CONTENTS AND DATAFILES`.

You cannot use this statement to drop a tablespace group. However, if *`tablespace`* is the only tablespace in a tablespace group, then Oracle Database removes the tablespace group from the data dictionary as well.

Restrictions on Dropping Tablespaces

Dropping tablespaces is subject to the following restrictions:

- You cannot drop a tablespace that contains a domain index or any objects created by a domain index.
- You cannot drop an undo tablespace if it is being used by any instance or if it contains any undo data needed to roll back uncommitted transactions.
- You cannot drop a tablespace that has been designated as the default tablespace for the database. You must first reassign another tablespace as the default tablespace and then drop the old default tablespace.
- You cannot drop a temporary tablespace if it is part of the database default temporary tablespace group. You must first remove the tablespace from the database default temporary tablespace group and then drop it.
- You cannot drop a temporary tablespace if it contains segments that are in use by existing sessions. In this case, no error is raised. The database waits until there are no segments in use by existing sessions and then drops the tablespace.
- You cannot drop a tablespace, even with the `INCLUDING CONTENTS` and `CASCADE CONSTRAINTS` clauses, if doing so would disable a primary key or unique constraint in another tablespace. For example, if the tablespace being dropped contains a primary key index, but the primary key column itself is in a different tablespace, then you cannot drop the tablespace until you have manually disabled the primary key constraint in the other tablespace.

See Also:

Oracle Database Data Cartridge Developer's Guide and *Oracle Database Concepts* for more information on domain indexes

{ DROP | KEEP } QUOTA

Specify DROP QUOTA to drop all user quotas for the tablespace. Specify KEEP QUOTA to retain all user quotas for the tablespace. The default is KEEP QUOTA.

You can view all user quotas for a tablespace by querying the DBA_TS_QUOTAS data dictionary view.

INCLUDING CONTENTS

Specify INCLUDING CONTENTS to drop all the contents of the tablespace. You must specify this clause to drop a tablespace that contains any database objects. If you omit this clause, and the tablespace is not empty, then the database returns an error and does not drop the tablespace.

DROP TABLESPACE fails, even if you specify INCLUDING CONTENTS, if the tablespace contains some, but not all, of the partitions or subpartitions of a single table. If all the partitions or subpartitions of a partitioned table reside in *tablespace*, then DROP TABLESPACE ... INCLUDING CONTENTS drops *tablespace*, as well as any associated index segments, LOB data and index segments, and nested table data and index segments of *table* in other tablespace(s).

For a partitioned index-organized table, if all the primary key index segments are in this tablespace, then this clause will also drop any overflow segments that exist in other tablespaces, as well as any associated mapping table in other tablespaces. If some of the primary key index segments are *not* in this tablespace, then the statement will fail. In that case, before you can drop the tablespace, you must use ALTER TABLE ... MOVE PARTITION to move those primary key index segments into this tablespace, drop the partitions whose overflow data segments are not in this tablespace, and drop the partitioned index-organized table.

If the tablespace contains a master table of a materialized view, then the database invalidates the materialized view.

If the tablespace contains a materialized view log, then the database drops the log and any other direct-path INSERT refresh information associated with the table.

AND DATAFILES

When you specify INCLUDING CONTENTS, the AND DATAFILES clause lets you instruct the database to delete the associated operating system files as well. Oracle Database writes a message to the alert log for each operating system file deleted. This clause is not needed for Oracle Managed Files, because they are removed from the system even if you do not specify AND DATAFILES.

KEEP DATAFILES

When you specify INCLUDING CONTENTS, the KEEP DATAFILES clause lets you instruct the database to leave untouched the associated operating system files, including Oracle Managed Files. You must specify this clause if you are using Oracle Managed Files and you do not want the associated operating system files removed by the INCLUDING CONTENTS clause.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints from tables outside *tablespace* that refer to primary and unique keys of tables inside *tablespace*. If you omit this clause and such referential integrity constraints exist, then Oracle Database returns an error and does not drop the tablespace.

Examples

Dropping a Tablespace: Example

The following statement drops the `tbs_01` tablespace and drops all referential integrity constraints that refer to primary and unique keys inside `tbs_01`:

```
DROP TABLESPACE tbs_01
    INCLUDING CONTENTS
        CASCADE CONSTRAINTS;
```

Deleting Operating System Files: Example

The following example drops the `tbs_02` tablespace and deletes all associated operating system data files:

```
DROP TABLESPACE tbs_02
   INCLUDING CONTENTS AND DATAFILES;
```

DROP TABLESPACE SET

> **Note:**
>
> This SQL statement is valid only if you are using Oracle Sharding. For more information on Oracle Sharding, refer to *Oracle Database Administrator's Guide*.

Purpose

Use the `DROP TABLESPACE SET` statement to drop a tablespace set from a shardgroup.

When you drop a tablespace set, Oracle Database does not place it in the recycle bin. Therefore, you cannot subsequently either purge or undrop the tablespace set.

> **See Also:**
>
> CREATE TABLESPACE SET and ALTER TABLESPACE SET

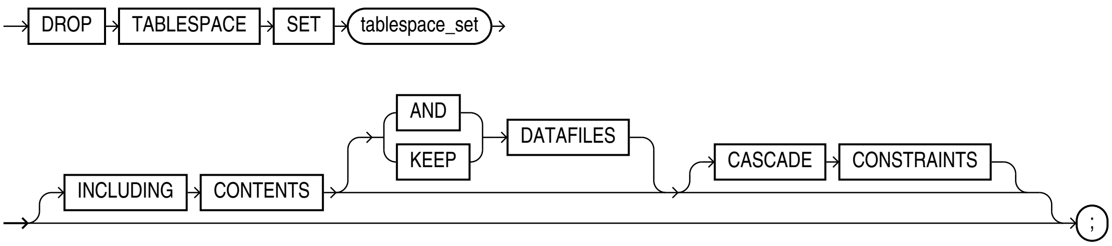
Prerequisites

You must be connected to a shard catalog database as an SDB user.

You must have the `DROP TABLESPACE` system privilege. You cannot drop a tablespace set if its tablespaces contain any rollback segments holding active transactions.

Syntax

***drop_tablespace_set*::=**



Semantics

tablespace_set

Specify the name of the tablespace set to be dropped.

INCLUDING CONTENTS

This clause lets you specify how the database manages objects and datafiles associated with the tablespaces in the tablespace set during the drop operation. The `INCLUDING CONTENTS` clause has the same semantics here as for the `DROP TABLESPACE` statement. See INCLUDING CONTENTS for the full semantics of this clause.

Examples

Dropping a Tablespace Set: Example

The following statement drops the tablespace set `ts1`:

```
DROP TABLESPACE SET ts1;
```

DROP TRIGGER

Purpose

Triggers are defined using PL/SQL. Refer to *Oracle Database PL/SQL Language Reference* for complete information on creating, altering, and dropping triggers.

Use the `DROP TRIGGER` statement to remove a database trigger from the database.

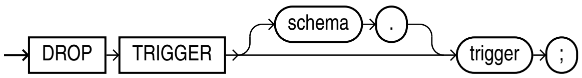
> **See Also:**
>
> CREATE TRIGGER and ALTER TRIGGER

Prerequisites

The trigger must be in your own schema or you must have the `DROP ANY TRIGGER` system privilege. To drop a trigger on `DATABASE` in another user's schema, you must also have the `ADMINISTER DATABASE TRIGGER` system privilege.

Syntax

***drop_trigger*::=**



Semantics

schema

Specify the schema containing the trigger. If you omit *`schema`*, then Oracle Database assumes the trigger is in your own schema.

trigger

Specify the name of the trigger to be dropped. Oracle Database removes it from the database and does not fire it again.

Examples

Dropping a Trigger: Example

The following statement drops the `salary_check` trigger in the schema `hr`:

```
DROP TRIGGER hr.salary_check;
```

DROP TYPE

Purpose

Object types are defined using PL/SQL. Refer to *Oracle Database PL/SQL Language Reference* for complete information on creating, altering, and dropping object types.

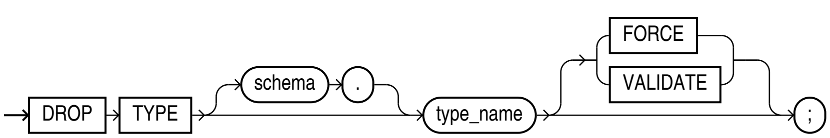
Use the `DROP TYPE` statement to drop the specification and body of an object type, a varray, or a nested table type.

Prerequisites

The object type, varray, or nested table type must be in your own schema or you must have the `DROP ANY TYPE` system privilege.

Syntax

***drop_type*::=**



Semantics

schema

Specify the schema containing the type. If you omit *schema*, then Oracle Database assumes the type is in your own schema.

type_name

Specify the name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If *type_name* is a supertype, then this statement will fail unless you also specify FORCE. If you specify FORCE, then the database invalidates all subtypes depending on this supertype.

If *type_name* is a statistics type, then this statement will fail unless you also specify FORCE. If you specify FORCE, then the database first disassociates all objects that are associated with *type_name* and then drops *type_name*.

> **See Also:**
>
> ASSOCIATE STATISTICS and DISASSOCIATE STATISTICS for more information on statistics types

If *type_name* is an object type that has been associated with a statistics type, then the database first attempts to disassociate *type_name* from the statistics type and then drops *type_name*. However, if statistics have been collected using the statistics type, then the database will be unable to disassociate *type_name* from the statistics type, and this statement will fail.

If *type_name* is an implementation type for an indextype, then the indextype will be marked INVALID.

If *type_name* has a public synonym defined on it, then the database will also drop the synonym.

Unless you specify FORCE, you can drop only object types, nested tables, or varray types that are standalone schema objects with no dependencies. This is the default behavior.

> **See Also:**
>
> CREATE INDEXTYPE

FORCE

Specify FORCE to drop the type even if it has dependent database objects. Oracle Database marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible.

> **Note:**
>
> Oracle does not recommend that you specify `FORCE` to drop object types with dependencies. This operation is not recoverable and could cause the data in the dependent tables or columns to become inaccessible.

VALIDATE

If you specify `VALIDATE` when dropping a type, then Oracle Database checks for stored instances of this type within substitutable columns of any of its supertypes. If no such instances are found, then the database completes the drop operation.

This clause is meaningful only for subtypes. Oracle recommends the use of this option to safely drop subtypes that do not have any explicit type or table dependencies.

Examples

Dropping an Object Type: Example

The following statement removes object type `person_t`. See *Oracle Database PL/SQL Language Reference* for the example that creates this object type. Any columns that are dependent on person_t are marked `UNUSED` and become inaccessible.

```
DROP TYPE person_t FORCE;
```

DROP TYPE BODY

Purpose

Object types are defined using PL/SQL. Refer to *Oracle Database PL/SQL Language Reference* for complete information on creating, altering, and dropping object types.

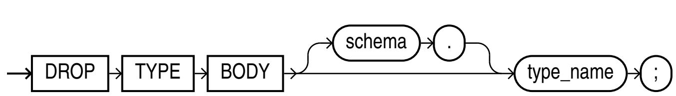
Use the `DROP TYPE BODY` statement to drop the body of an object type, varray, or nested table type. When you drop a type body, the object type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the object type, although you cannot call the member functions.

Prerequisites

The object type body must be in your own schema or you must have the `DROP ANY TYPE` system privilege.

Syntax

***drop_type_body*::=**



Semantics

schema

Specify the schema containing the object type. If you omit `schema`, then Oracle Database assumes the object type is in your own schema.

type_name

Specify the name of the object type body to be dropped.

Restriction on Dropping Type Bodies

You can drop a type body only if it has no type or table dependencies.

Examples

Dropping an Object Type Body: Example

The following statement removes object type body `data_typ1`. See *Oracle Database PL/SQL Language Reference* for the example that creates this object type.

```
DROP TYPE BODY data_typ1;
```

DROP USER

Purpose

Use the `DROP USER` statement to remove a database user and optionally remove the user's objects.

In an Oracle Automatic Storage Management (Oracle ASM) cluster, a user authenticated `AS SYSASM` can use this clause to remove a user from the password file that is local to the Oracle ASM instance of the current node.

When you drop a user, Oracle Database also purges all of that user's schema objects from the recycle bin.

> **Note:**
>
> Do not attempt to drop the users `SYS` or `SYSTEM`. Doing so will corrupt your database.

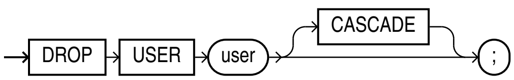
> **See Also:**
>
> CREATE USER and ALTER USER for information on creating and modifying a user

Prerequisites

You must have the `DROP USER` system privilege. In an Oracle ASM cluster, you must be authenticated `AS SYSASM`.

Syntax

***drop_user*::=**



Semantics

user

Specify the user to be dropped. Oracle Database does not drop users whose schemas contain objects unless you specify `CASCADE` or unless you first explicitly drop the user's objects.

Restriction on Dropping Users

You cannot drop a user whose schema contains a table that uses a flashback data archive for historical tracking. You must first disable the table's use of the flashback data archive.

CASCADE

Specify `CASCADE` to drop all objects in the user's schema before dropping the user. You must specify this clause to drop a user whose schema contains any objects.

- If the user's schema contains tables, then Oracle Database drops the tables and automatically drops any referential integrity constraints on tables in other schemas that refer to primary and unique keys on these tables.
- If this clause results in tables being dropped, then the database also drops all domain indexes created on columns of those tables and invokes appropriate drop routines.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on these routines

- Oracle Database invalidates, but does not drop, the following objects in other schemas:
  - Views or synonyms for objects in the dropped user's schema
  - Stored procedures, functions, or packages that query objects in the dropped user's schema
- Oracle Database does not drop materialized views in other schemas that are based on tables in the dropped user's schema. However, because the base tables no longer exist, the materialized views in the other schemas can no longer be refreshed.
- Oracle Database drops all triggers in the user's schema.

- Oracle Database does not drop roles created by the user.

> **Note:**
>
> Oracle Database also drops with `FORCE` all types owned by the user. See the FORCE keyword of DROP TYPE .

Examples

Dropping a Database User: Example

If user Sidney's schema contains no objects, then you can drop `sidney` by issuing the statement:

```
DROP USER sidney;
```

If Sidney's schema contains objects, then you must use the `CASCADE` clause to drop `sidney` and the objects:

```
DROP USER sidney CASCADE;
```

DROP VIEW

Purpose

Use the `DROP VIEW` statement to remove a view or an object view from the database. You can change the definition of a view by dropping and re-creating it.

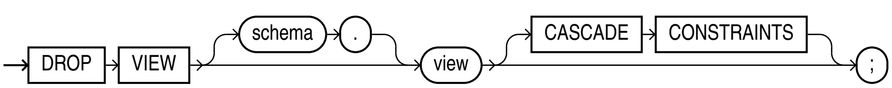
> **See Also:**
>
> CREATE VIEW and ALTER VIEW for information on creating and modifying a view

Prerequisites

The view must be in your own schema or you must have the `DROP ANY VIEW` system privilege.

Syntax

***drop_view*::=**



Semantics

schema

Specify the schema containing the view. If you omit *`schema`*, then Oracle Database assumes the view is in your own schema.

view

Specify the name of the view to be dropped.

Oracle Database does not drop views, materialized views, and synonyms that are dependent on the view but marks them `INVALID`. You can drop them or redefine views and synonyms, or you can define other views in such a way that the invalid views and synonyms become valid again.

If any subviews have been defined on *`view`*, then the database invalidates the subviews as well. To determine whether the view has any subviews, query the `SUPERVIEW_NAME` column of the `USER_`, `ALL_`, or `DBA_VIEWS` data dictionary views.

> **See Also:**
>
> - CREATE TABLE and CREATE SYNONYM
> - ALTER MATERIALIZED VIEW for information on revalidating invalid materialized views

CASCADE CONSTRAINTS

Specify `CASCADE CONSTRAINTS` to drop all referential integrity constraints that refer to primary and unique keys in the view to be dropped. If you omit this clause, and such constraints exist, then the `DROP` statement fails.

Examples

Dropping a View: Example

The following statement drops the `emp_view` view, which was created in "Creating a View: Example":

```
DROP VIEW emp_view;
```

EXPLAIN PLAN

Purpose

Use the `EXPLAIN PLAN` statement to determine the execution plan Oracle Database follows to execute a specified SQL statement. This statement inserts a row describing each step of the execution plan into a specified table. You can also issue the `EXPLAIN PLAN` statement as part of the SQL trace facility.

This statement also determines the cost of executing the statement. If any domain indexes are defined on the table, then user-defined CPU and I/O costs will also be inserted.

The definition of a sample output table `PLAN_TABLE` is available in a SQL script on your distribution media. Your output table must have the same column names and data types as this table. The common name of this script is `UTLXPLAN.SQL`. The exact name and location depend on your operating system.

Oracle Database provides information on cached cursors through several dynamic performance views:

- For information on the work areas used by SQL cursors, query `V$SQL_WORKAREA`.
- For information on the execution plan for a cached cursor, query `V$SQL_PLAN`.
- For execution statistics at each step or operation of an execution plan of cached cursors (for example, number of produced rows, number of blocks read), query `V$SQL_PLAN_STATISTICS`.
- For a selective precomputed join of the preceding three views, query `V$SQL_PLAN_STATISTICS_ALL`.
- Execution statistics at each step or operation of an execution plan of cached cursors are displayed in `V$SQL_PLAN_MONITOR` if the statement execution is monitored. You can force monitoring using the `MONITOR` hint.

> **See Also:**
>
> - *Oracle Database SQL Tuning Guide* for information on the output of `EXPLAIN PLAN`, how to use the SQL trace facility, and how to generate and interpret execution plans
> - *Oracle Database Reference* for information on dynamic performance views

Prerequisites

To issue an `EXPLAIN PLAN` statement, you must have the privileges necessary to insert rows into an existing output table that you specify to hold the execution plan.

You must also have the privileges necessary to execute the SQL statement for which you are determining the execution plan. If the SQL statement accesses a view, then you must have privileges to access any tables and views on which the view is based. If the view is based on another view that is based on a table, then you must have privileges to access both the other view and its underlying table.

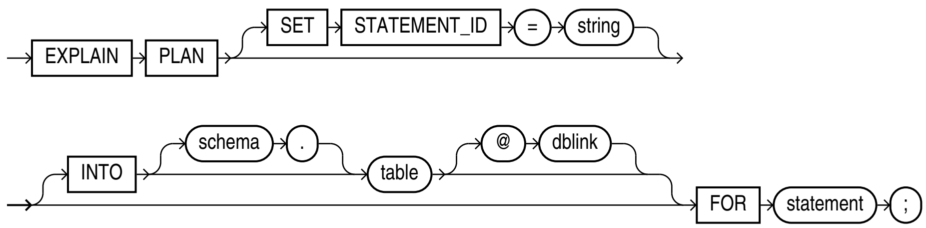
To examine the execution plan produced by an `EXPLAIN PLAN` statement, you must have the privileges necessary to query the output table.

The `EXPLAIN PLAN` statement is a data manipulation language (DML) statement, rather than a data definition language (DDL) statement. Therefore, Oracle Database does not implicitly commit the changes made by an `EXPLAIN PLAN` statement. If you want to keep the rows generated by an `EXPLAIN PLAN` statement in the output table, then you must commit the transaction containing the statement.

> **See Also:**
>
> INSERT and SELECT for information on the privileges you need to populate and query the plan table

Syntax

***explain_plan*::=**



Semantics

SET STATEMENT_ID Clause

Specify a value for the `STATEMENT_ID` column for the rows of the execution plan in the output table. You can then use this value to identify these rows among others in the output table. Be sure to specify a `STATEMENT_ID` value if your output table contains rows from many execution plans. If you omit this clause, then the `STATEMENT_ID` value defaults to null.

INTO *table* Clause

Specify the name of the output table, and optionally its schema and database. This table must exist before you use the `EXPLAIN PLAN` statement.

If you omit *`schema`*, then the database assumes the table is in your own schema.

The *`dblink`* can be a complete or partial name of a database link to a remote Oracle Database where the output table is located. You can specify a remote output table only if you are using Oracle Database distributed functionality. If you omit *`dblink`*, then the database assumes the table is on your local database. See "References to Objects in Remote Databases " for information on referring to database links.

If you omit `INTO` altogether, then the database assumes an output table named `PLAN_TABLE` in your own schema on your local database.

FOR *statement* Clause

Specify a `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `CREATE TABLE`, `CREATE INDEX`, or `ALTER INDEX` ... `REBUILD` statement for which the execution plan is generated.

Notes on EXPLAIN PLAN

The following notes apply to `EXPLAIN PLAN`:

- If *`statement`* includes the *`parallel_clause`*, then the resulting execution plan will indicate parallel execution. However, `EXPLAIN PLAN` actually inserts the statement into the plan table, so that the parallel DML statement you submit is no longer the first DML statement in the transaction. This violates the Oracle Database restriction of one parallel DML statement in a single transaction, and the statement will be executed serially. To maintain parallel execution of the statements, you must commit or roll back the `EXPLAIN PLAN` statement, and then submit the parallel DML statement.

- To determine the execution plan for an operation on a temporary table, EXPLAIN PLAN must be run from the same session, because the data in temporary tables is session specific.

Examples

EXPLAIN PLAN Examples

The following statement determines the execution plan and cost for an UPDATE statement and inserts rows describing the execution plan into the specified plan_table table with the STATEMENT_ID value of 'Raise in Tokyo':

```
EXPLAIN PLAN
    SET STATEMENT_ID = 'Raise in Tokyo'
    INTO plan_table
    FOR UPDATE employees
        SET salary = salary * 1.10
        WHERE department_id =
           (SELECT department_id FROM departments
               WHERE location_id = 1700);
```

The following SELECT statement queries the plan_table table and returns the execution plan and the cost:

```
SELECT id, LPAD(' ',2*(LEVEL-1))||operation operation, options,
       object_name, object_alias, position
    FROM plan_table
    START WITH id = 0 AND statement_id = 'Raise in Tokyo'
    CONNECT BY PRIOR id = parent_id AND statement_id = 'Raise in Tokyo'
    ORDER BY id;
```

The query returns this execution plan:

```
 ID OPERATION            OPTIONS              OBJECT_NAME          OBJECT_ALIAS         POSITION
--- -------------------- -------------------- -------------------- -------------------- --------
  0 UPDATE STATEMENT                                                                           4
  1   UPDATE                                  EMPLOYEES                                        1
  2     INDEX            RANGE SCAN           EMP_DEPARTMENT_IX    EMPLOYEES@UPD$1             1
  3       TABLE ACCESS   BY INDEX ROWID       DEPARTMENTS          DEPARTMENTS@SEL$1           1
  4         INDEX        RANGE SCAN           DEPT_LOCATION_IX     DEPARTMENTS@SEL$1           1
```

The value in the POSITION column of the first row shows that the statement has a cost of 4.

EXPLAIN PLAN: Partitioned Example

The sample table sh.sales is partitioned on the time_id column. Partition sales_q3_2000 contains time values less than Oct. 1, 2000, and there is a local index sales_time_bix on the time_id column.

Consider the query:

```
EXPLAIN PLAN FOR
  SELECT * FROM sales
     WHERE time_id BETWEEN :h AND '01-OCT-2000';
```

where :h represents an already declared bind variable. EXPLAIN PLAN executes this query with PLAN_TABLE as the output table. The basic execution plan, including partitioning information, is obtained with the following query:

```
SELECT operation, options, partition_start, partition_stop,
       partition_id
  FROM plan_table;
```

FLASHBACK DATABASE

Purpose

Use the `FLASHBACK DATABASE` statement to return the database to a past time or system change number (SCN). This statement provides a fast alternative to performing incomplete database recovery.

Following a `FLASHBACK DATABASE` operation, in order to have write access to the flashed back database, you must reopen it with an `ALTER DATABASE OPEN RESETLOGS` statement.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information on `FLASHBACK DATABASE`

Prerequisites

You must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege.

If you are connected to a multitenant container database (CDB):

- To flash back a CDB, you must be connected to the root and you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege granted commonly.
- To flash back a PDB you must be connected to the root and you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege granted commonly, or you must be connected to the PDB you want to flash back and you must have the `SYSDBA`, `SYSBACKUP`, or `SYSDG` system privilege, granted commonly or granted locally in that PDB.

A fast recovery area must have been prepared for the database. The database must have been put in `FLASHBACK` mode with an `ALTER DATABASE FLASHBACK ON` statement unless you are flashing the database back to a guaranteed restore point. The database must be mounted but not open.

In addition:

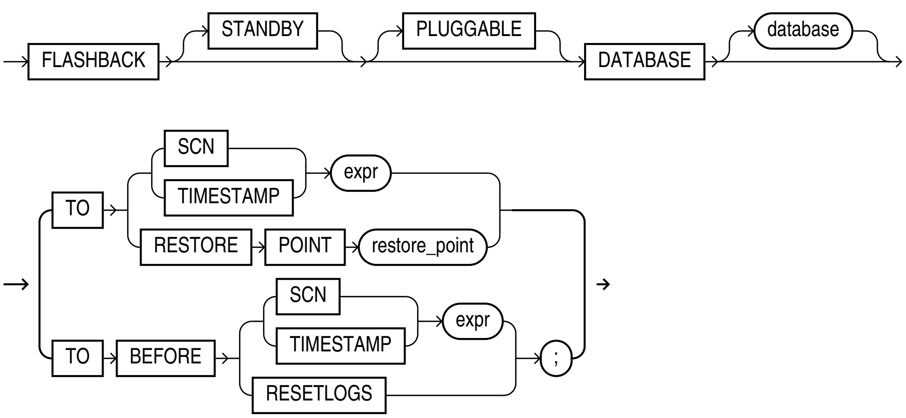
- The database must run in `ARCHIVELOG` mode.
- The database must be mounted, but not open, with a current control file. The control file cannot be a backup or re-created. When the database control file is restored from backup or re-created, all existing flashback log information is discarded.
- The database must contain no online tablespaces for which flashback functionality was disabled with the SQL statement `ALTER TABLESPACE` ... `FLASHBACK OFF`.

See Also:

- *Oracle Database Backup and Recovery User's Guide* and the `ALTER DATABASE` ... *flashback_mode_clause* for information on putting the database in `FLASHBACK` mode
- CREATE RESTORE POINT for information on restore points and guaranteed restore points

Syntax

***flashback_database*::=**



Semantics

When you issue a `FLASHBACK DATABASE` statement, Oracle Database first verifies that all required archived and online redo logs are available. If they are available, then it reverts all currently online data files in the database to the SCN or time specified in this statement.

- The amount of Flashback data retained in the database is controlled by the `DB_FLASHBACK_RETENTION_TARGET` initialization parameter and the size of the fast recovery area. You can determine how far back you can flash back the database by querying the `V$FLASHBACK_DATABASE_LOG` view.
- If insufficient data remains in the database to perform the flashback, then you can use standard recovery procedures to recover the database to a past point in time.
- If insufficient data remains for a set of data files, then the database returns an error. In this case, you can take those data files offline and reissue the statement to revert the remainder of the database. You can then attempt to recover the offline data files using standard recovery procedures.

See Also:

Oracle Database Backup and Recovery User's Guide for more information on recovering data files

STANDBY

Specify STANDBY to revert the standby database to an earlier SCN or time. If the database is not a standby database, then the database returns an error. If you omit this clause, then *database* can be either a primary or a standby database.

> **See Also:**
>
> *Oracle Data Guard Concepts and Administration* for information on how you can use FLASHBACK DATABASE on a standby database to achieve different delays

PLUGGABLE

Specify PLUGGABLE to flash back a PDB. You must specify this clause whether the current container is the root or the PDB you want to flash back.

Restrictions on Flashing Back a PDB

- You cannot flash back a proxy PDB.
- If the CDB is in shared undo mode, then you can only flash back a PDB to a clean PDB restore point. Refer to the CLEAN clause of CREATE RESTORE POINT for more information.

database

If you are flashing back a non-CDB or a CDB, then you can optionally specify the name of the database to be flashed back. If you omit *database*, then Oracle Database flashes back the database identified by the value of the initialization parameter DB_NAME.

If you are flashing back a PDB and the current container is the root, then use *database* to specify the name of the PDB to be flashed back. If you are flashing back a PDB and the current container is that PDB, then you can optionally use *database* to specify the PDB name.

TO SCN Clause

Specify a system change number (SCN):

- TO SCN reverts the database back to its state at the specified SCN.
- TO BEFORE SCN reverts the database back to its state at the system change number just preceding the specified SCN.

You can determine the current SCN by querying the CURRENT_SCN column of the V$DATABASE view. This in turn lets you save the SCN to a spool file, for example, before running a high-risk batch job.

TO TIMESTAMP Clause

Specify a valid datetime expression.

- TO TIMESTAMP reverts the database back to its state at the specified timestamp.

- `TO BEFORE TIMESTAMP` reverts the database back to its state one second before the specified timestamp.

You can represent the timestamp as an offset from a determinate value, such as `SYSDATE`, or as an absolute system timestamp.

TO RESTORE POINT Clause

Specify this clause to flash back the database to the specified restore point. If you have not enabled flashback database, then this is the only clause you can specify in this `FLASHBACK DATABASE` statement. If the database is not in `FLASHBACK` mode, as described in the "Prerequisites" section above, then this is the only clause you can specify for this statement.

RESETLOGS

Specify `TO BEFORE RESETLOGS` to flash the database back to just before the last resetlogs operation (`ALTER DATABASE OPEN RESETLOGS`).

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information about this clause

Examples

Assuming that you have prepared a fast recovery area for the database and enabled media recovery, enable database `FLASHBACK` mode and open the database with the following statements:

```
STARTUP MOUNT
ALTER DATABASE FLASHBACK ON;
ALTER DATABASE OPEN;
```

With your database open for at least a day, you can flash back the database one day with the following statements:

```
SHUTDOWN DATABASE
STARTUP MOUNT
FLASHBACK DATABASE TO TIMESTAMP SYSDATE-1;
```

FLASHBACK TABLE

Purpose

Use the `FLASHBACK TABLE` statement to restore an earlier state of a table in the event of human or application error. The time in the past to which the table can be flashed back is dependent on the amount of undo data in the system. Also, Oracle Database cannot restore a table to an earlier state across any DDL operations that change the structure of the table.

> **Note:**
>
> Oracle strongly recommends that you run your database in automatic undo mode by leaving the UNDO_MANAGEMENT initialization parameter set to AUTO, which is the default. In addition, set the UNDO_RETENTION initialization parameter to an interval large enough to include the oldest data you anticipate needing. For more information refer to the documentation on the UNDO_MANAGEMENT and UNDO_RETENTION initialization parameters.

You cannot roll back a FLASHBACK TABLE statement. However, you can issue another FLASHBACK TABLE statement and specify a time just prior to the current time. Therefore, it is advisable to record the current SCN before issuing a FLASHBACK TABLE clause.

> **See Also:**
>
> - FLASHBACK DATABASE for information on reverting the entire database to an earlier version
> - the *flashback_query_clause* of SELECT for information on retrieving past data from a table
> - *Oracle Database Backup and Recovery User's Guide* for additional information on using the FLASHBACK TABLE statement

Prerequisites

To flash back a table to an earlier SCN or timestamp, you must have either the FLASHBACK object privilege on the table or the FLASHBACK ANY TABLE system privilege. In addition, you must have the READ or SELECT object privilege on the table, and you must have the INSERT, DELETE, and ALTER object privileges on the table.

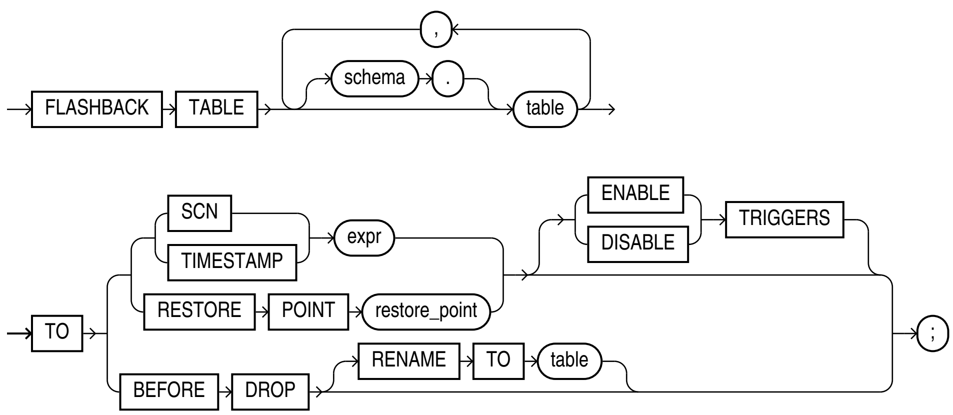
Row movement must be enabled for all tables in the Flashback list unless you are flashing back the table TO BEFORE DROP. That operation is called a **flashback drop** operation, and it uses dropped data in the recycle bin rather than undo data. Refer to *row_movement_clause* for information on enabling row movement.

To flash back a table to a restore point, you must have the SELECT ANY DICTIONARY or FLASHBACK ANY TABLE system privilege or the SELECT_CATALOG_ROLE role.

To flash back a table to before a DROP TABLE operation, you need only the privileges necessary to drop the table.

Syntax

***flashback_table*::=**



Semantics

During an Oracle Flashback Table operation, Oracle Database acquires exclusive DML locks on all the tables specified in the Flashback list. These locks prevent any operations on the tables while they are reverting to their earlier state.

The Flashback Table operation is executed in a single transaction, regardless of the number of tables specified in the Flashback list. Either all of the tables revert to the earlier state or none of them do. If the Flashback Table operation fails on any table, then the entire statement fails.

At the completion of the Flashback Table operation, the data in *`table`* is consistent with *`table`* at the earlier time. However, `FLASHBACK TABLE TO SCN` or `TIMESTAMP` does not preserve rowids, and `FLASHBACK TABLE TO BEFORE DROP` does not recover referential constraints.

Oracle Database does not revert statistics associated with *`table`* to their earlier form. Indexes on *`table`* that exist currently are reverted and reflect the state of the table at the Flashback point. If the index exists now but did not yet exist at the Flashback point, then the database updates the index to reflect the state of the table at the Flashback point. However, indexes that were dropped during the interval between the Flashback point and the current time are not restored.

schema

Specify the schema containing the table. If you omit *`schema`*, then the database assumes the table is in your own schema.

table

Specify the name of one or more tables containing data you want to revert to an earlier version.

Restrictions on Flashing Back Tables

This statement is subject to the following restrictions:

- Flashback Table operations are not valid for the following type objects: tables that are part of a cluster, materialized views, Advanced Queuing (AQ) tables, static data dictionary tables, system tables, remote tables, object tables, nested tables, or individual table partitions or subpartitions.

- The following DDL operations change the structure of a table, so that you cannot subsequently use the `TO SCN` or `TO TIMESTAMP` clause to flash the table back to a time preceding the operation: upgrading, moving, or truncating a table; adding a constraint to a table, adding a table to a cluster; modifying or dropping a column; changing a column encryption key; adding, dropping, merging, splitting, coalescing, or truncating a partition or subpartition (with the exception of adding a range partition).

TO SCN Clause

Specify the system change number (SCN) corresponding to the point in time to which you want to return the table. The *`expr`* must evaluate to a number representing a valid SCN.

TO TIMESTAMP Clause

Specify a timestamp value corresponding to the point in time to which you want to return the table. The *`expr`* must evaluate to a valid timestamp in the past. The table will be flashed back to a time within approximately 3 seconds of the specified timestamp.

TO RESTORE POINT Clause

Specify a restore point to which you want to flash back the table. The restore point must already have been created.

> **See Also:**
>
> CREATE RESTORE POINT for information on creating restore points

ENABLE | DISABLE TRIGGERS

By default, Oracle Database disables all enabled triggers defined on *`table`* during the Flashback Table operation and then reenables them after the Flashback Table operation is complete. Specify `ENABLE TRIGGERS` if you want to override this default behavior and keep the triggers enabled during the Flashback process.

This clause affects only those database triggers defined on *`table`* that are already enabled. To enable currently disabled triggers selectively, use the `ALTER TABLE` ... *`enable_disable_clause`* before you issue the `FLASHBACK TABLE` statement with the `ENABLE TRIGGERS` clause.

TO BEFORE DROP Clause

Use this clause to retrieve from the recycle bin a table that has been dropped, along with all possible dependent objects. The table must have resided in a locally managed tablespace other than the `SYSTEM` tablespace.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on the recycle bin and naming conventions for objects in the recycle bin
> - PURGE for information on removing objects permanently from the recycle bin

You can specify either the original user-specified name of the table or the system-generated name Oracle Database assigned to the object when it was dropped.

- System-generated recycle bin object names are unique. Therefore, if you specify the system-generated name, then the database retrieves that specified object.

  To see the contents of your recycle bin, query the `USER_RECYCLEBIN` data dictionary view. You can use the `RECYCLEBIN` synonym instead. The following two statements return the same rows:

  ```
  SELECT * FROM RECYCLEBIN;
  SELECT * FROM USER_RECYCLEBIN;
  ```

- If you specify the user-specified name, and if the recycle bin contains more than one object of that name, then the database retrieves the object that was moved to the recycle bin most recently. If you want to retrieve an older version of the table, then do one of these things:
  - Specify the system-generated recycle bin name of the table you want to retrieve.
  - Issue additional `FLASHBACK TABLE` ... `TO BEFORE DROP` statements until you retrieve the table you want.

Oracle Database attempts to preserve the original table name. If a new table of the same name has been created in the same schema since the original table was dropped, then the database returns an error unless you also specify the `RENAME TO` clause.

RENAME TO Clause

Use this clause to specify a new name for the table being retrieved from the recycle bin.

Notes on Flashing Back Dropped Tables

The following notes apply to flashing back dropped tables:

- Oracle Database retrieves all indexes defined on the table retrieved from the recycle bin except for bitmap join indexes and domain indexes. (Bitmap join indexes and domain indexes are not put in the recycle bin during a `DROP TABLE` operation, so cannot be retrieved.)
- The database also retrieves all triggers and constraints defined on the table except for referential integrity constraints that reference other tables.

  The retrieved indexes, triggers, and constraints have recycle bin names. Therefore it is advisable to query the `USER_RECYCLEBIN` view before issuing a `FLASHBACK TABLE` ... `TO BEFORE DROP` statement so that you can rename the retrieved triggers and constraints to more usable names.
- When you drop a table, all materialized view logs defined on the table are also dropped but are not placed in the recycle bin. Therefore, the materialized view logs cannot be flashed back along with the table.

- When you drop a table, any indexes on the table are dropped and put into the recycle bin along with the table. If subsequent space pressures arise, then the database reclaims space from the recycle bin by first purging indexes. In this case, when you flash back the table, you may not get back all of the indexes that were defined on the table.
- You cannot flash back a table if it has been purged, either by a user or by Oracle Database as a result of some space reclamation operation.

Examples

Restoring a Table to an Earlier State: Examples

The examples below create a new table, `employees_test`, with row movement enabled, update values within the new table, and issue the `FLASHBACK TABLE` statement.

Create table `employees_test`, with row movement enabled, from table `employees` of the sample `hr` schema:

```
CREATE TABLE employees_test
  AS SELECT * FROM employees;
```

As a benchmark, list those salaries less than 2500:

```
SELECT salary
  FROM employees_test
  WHERE salary < 2500;

    SALARY
----------
      2400
      2200
      2100
      2400
      2200
```

> **Note:**
>
> To allow time for the SCN to propagate to the mapping table used by the `FLASHBACK TABLE` statement, wait a minimum of 5 minutes prior to issuing the following statement. This wait would not be necessary if a previously existing table were being used in this example.

Enable row movement for the table:

```
ALTER TABLE employees_test
   ENABLE ROW MOVEMENT;
```

Issue a 10% salary increase to those employees earning less than 2500:

```
UPDATE employees_test
  SET salary = salary * 1.1
  WHERE salary < 2500;

5 rows updated.
COMMIT;
```

As a second benchmark, list those salaries that remain less than 2500 following the 10% increase:

```
SELECT salary
  FROM employees_test
  WHERE salary < 2500;

    SALARY
----------
      2420
      2310
      2420
```

Restore the table `employees_test` to its state prior to the current system time. The unrealistic duration of 1 minute is used so that you can test this series of examples quickly. Under normal circumstances a much greater interval would have elapsed.

```
FLASHBACK TABLE employees_test
  TO TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' minute);
```

List those salaries less than 2500. After the `FLASHBACK TABLE` statement issued above, this list should match the list in the first benchmark.

```
SELECT salary
  FROM employees_test
  WHERE salary < 2500;

    SALARY
----------
      2400
      2200
      2100
      2400
      2200
```

Retrieving a Dropped Table: Example

If you accidentally drop the `pm.print_media` table and want to retrieve it, then issue the following statement:

```
FLASHBACK TABLE print_media TO BEFORE DROP;
```

If another `print_media` table has been created in the `pm` schema, then use the `RENAME TO` clause to rename the retrieved table:

```
FLASHBACK TABLE print_media TO BEFORE DROP RENAME TO print_media_old;
```

If you know that the employees table has been dropped multiple times, and you want to retrieve the oldest version, then query the `USER_RECYLEBIN` table to determine the system-generated name, and then use that name in the `FLASHBACK TABLE` statement. (System-generated names in your database will differ from those shown here.)

```
SELECT object_name, droptime FROM user_recyclebin
   WHERE original_name = 'PRINT_MEDIA';

OBJECT_NAME                    DROPTIME
------------------------------ -------------------
RB$$45703$TABLE$0              2003-06-03:15:26:39
RB$$45704$TABLE$0              2003-06-12:12:27:27
RB$$45705$TABLE$0              2003-07-08:09:28:01
```

GRANT

Purpose

Use the `GRANT` statement to grant:

- System privileges to users and roles. Table 18-1 lists the system privileges (organized by the database object operated upon).
- Roles to users, roles, and program units. The granted roles can be either user-defined (local or external) or predefined. For a list of predefined roles, refer to *Oracle Database Security Guide*.
- Object privileges for a particular object to users and roles. Table 18-2 lists the object privileges (organized by the database object operated upon).

> **Note:**
>
> Global roles (created with `IDENTIFIED GLOBALLY`) are granted through enterprise roles and cannot be granted using the `GRANT` statement.

Notes on Authorizing Database Users

You can authorize database users through means other than the database and the `GRANT` statement.

- Many Oracle Database privileges are granted through supplied PL/SQL and Java packages. For information on those privileges, refer to the documentation for the appropriate package.
- Some operating systems have facilities that let you grant roles to Oracle Database users with the initialization parameter `OS_ROLES`. If you choose to grant roles to users through operating system facilities, then you cannot also grant roles to users with the `GRANT` statement, although you can use the `GRANT` statement to grant system privileges to users and system privileges and roles to other roles.

Note on Oracle Automatic Storage Management

A user authenticated `AS SYSASM` can use this statement to grant the system privileges `SYSASM`, `SYSOPER`, and `SYSDBA` to a user in the Oracle ASM password file of the current node.

Note on Editionable Objects

A `GRANT` operation to grant object privileges on an editionable object actualizes the object in the current edition. See *Oracle Database Development Guide* for more information about editions and editionable objects.

See Also:

- CREATE USER and CREATE ROLE for definitions of local, global, and external privileges
- *Oracle Database Security Guide* for information about other authorization methods and for information about privileges
- REVOKE for information on revoking grants

Prerequisites

To grant a **system privilege**, one of the following conditions must be met:

- You must have been granted the GRANT ANY PRIVILEGE system privilege. In this case, if you grant the system privilege to a role, then a user to whom the role has been granted does not have the privilege unless the role is enabled in user's session.
- You must have been granted the system privilege with the ADMIN OPTION. In this case, if you grant the system privilege to a role, then a user to whom the role has been granted has the privilege regardless whether the role is enabled in the user's session.

To grant a **role to a user or another role**, you must have been directly granted the role with the ADMIN OPTION, or you must have been granted the GRANT ANY ROLE system privilege, or you must have created the role.

To grant a **role to a program unit in your own schema**, you must have been directly granted the role with either the ADMIN OPTION or the DELEGATE OPTION, or you must have been granted the GRANT ANY ROLE system privilege, or you must have created the role.

To grant a **role to a program unit in another user's schema**, you must be the user SYS and the role must have been created by the schema owner or directly granted to the schema owner.

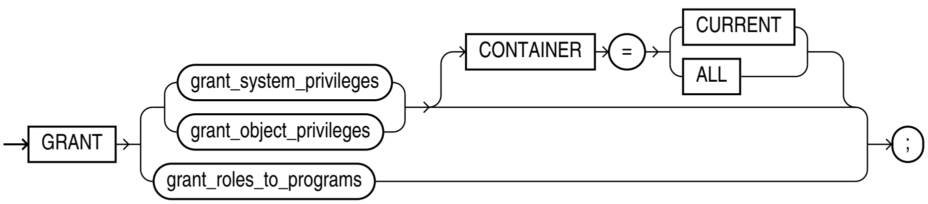
To grant an **object privilege on a user**, by specifying the ON USER clause of the *on_object_clause*, you must be the user on whom the privilege is granted, or you must have been granted the object privilege on that user with the WITH GRANT OPTION, or you must have been granted the GRANT ANY OBJECT PRIVILEGE system privilege. If you can grant an object privilege on a user only because you have the GRANT ANY OBJECT PRIVILEGE, then the GRANTOR column of the *_TAB_PRIVS views displays the user on whom the privilege is granted rather than the user who issued the GRANT statement.

To grant an **object privilege on all other types of objects**, you must own the object, or the owner of the object must have granted you the object privileges with the WITH GRANT OPTION, or you must have been granted the GRANT ANY OBJECT PRIVILEGE system privilege. If you have the GRANT ANY OBJECT PRIVILEGE, then you can grant the object privilege only if the object owner could have granted the same object privilege. In this case, the GRANTOR column of the *_TAB_PRIVS views displays the object owner rather than the user who issued the GRANT statement.

To specify the CONTAINER clause, you must be connected to a multitenant container database (CDB). To specify CONTAINER = ALL, the current container must be the root.

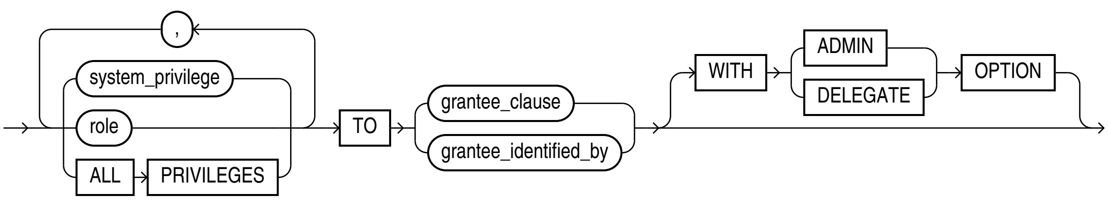
Syntax

***grant*::=**



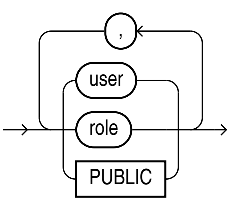
(*grant_system_privileges*::=, *grant_object_privileges*::=, *grant_roles_to_programs*::=)

***grant_system_privileges*::=**

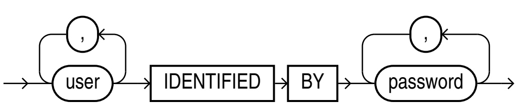


(*grantee_clause*::=, *grantee_identified_by*::=)

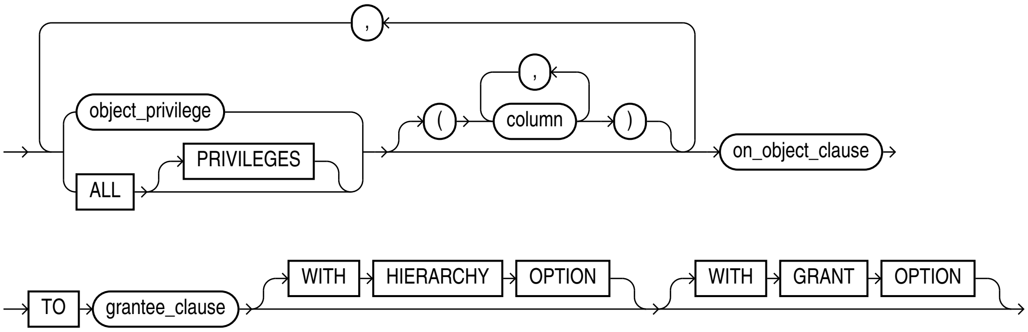
***grantee_clause*::=**



***grantee_identified_by*::=**

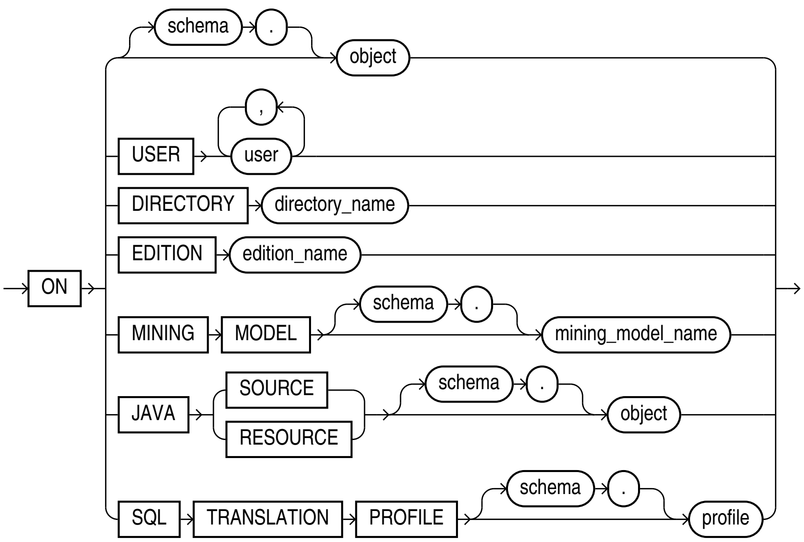


grant_object_privileges::=

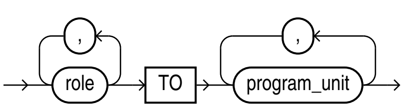


(*on_object_clause*::=, *grantee_clause*::=)

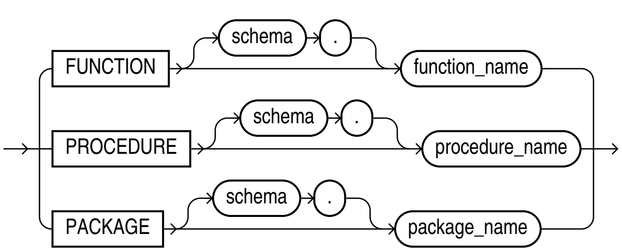
on_object_clause::=



grant_roles_to_programs::=



program_unit::=



Semantics

grant_system_privileges

Use these clauses to grant system privileges.

system_privilege

Specify the system privilege you want to grant. Table 18-1 lists the system privileges, organized by the database object operated upon.

- If you grant a privilege to a **user**, then the database adds the privilege to the user's privilege domain. The user can immediately exercise the privilege. Oracle recommends that you only grant the `ANY` privileges to trusted users.
- If you grant a privilege to a **role**, then the database adds the privilege to the privilege domain of the role. Users who have been granted and have enabled the role can immediately exercise the privilege. Other users who have been granted the role can enable the role and exercise the privilege.

> **See Also:**
>
> Granting a System Privilege to a User: Example and "Granting System Privileges to a Role: Example"

- If you grant a privilege to **`PUBLIC`**, then the database adds the privilege to the privilege domains of each user. All users can immediately perform operations authorized by the privilege. Oracle recommends against granting system privileges to `PUBLIC`.

role

Specify the role you want to grant. You can grant an Oracle Database predefined role or a user-defined role.

- If you grant a role to a **user**, then the database makes the role available to the user. The user can immediately enable the role and exercise the privileges in the privilege domain of the role.

  In the case of a secure application role, you need not grant such a role directly to the user. You can let the associated PL/SQL package do this, assuming the user passes appropriate security policies. For more information, see the `CREATE ROLE` semantics for USING *package* and *Oracle Database Security Guide*
- If you grant a role to another **role**, then the database adds the privilege domain of the granted role to the privilege domain of the grantee role. Users who have been granted the grantee role can enable it and exercise the privileges in the granted role's privilege domain.
- If you grant a role to **`PUBLIC`**, then the database makes the role available to all users. All users can immediately enable the role and exercise the privileges in the privilege domain of the role.

ALL PRIVILEGES

Specify `ALL PRIVILEGES` to grant all of the system privileges listed in Table 18-1, except the `SELECT ANY DICTIONARY`, `ALTER DATABASE LINK`, and `ALTER PUBLIC DATABASE LINK` privileges.

> **See Also:**
>
> - *Oracle Database Security Guide* for information on the Oracle predefined roles
> - "Granting a Role to a Role: Example"
> - CREATE ROLE for information on creating a user-defined role

grantee_clause

Use the *`grantee_clause`* to specify the users or roles to which the system privilege, role, or object privilege is granted.

PUBLIC

Specify `PUBLIC` to grant the privileges to all users. Oracle recommends against granting system privileges to `PUBLIC`.

Restriction on Grantees

A user, role, or `PUBLIC` cannot appear more than once in the *`grantee_clause`*.

grantee_identified_by

The *`grantee_identified_by`* clause lets you assign passwords to users when granting them system privileges and roles. You must specify an equal number of users and passwords. The first password is assigned to the first user, the second password is assigned to the second user, and so on. If a specified user exists, then the database resets the user's password. If a specified user does not exist, then the database creates the user with the password.

> **See Also:**
>
> CREATE USER for restrictions on usernames and passwords and "Assigning User Passwords When Granting a System Privilege: Example"

WITH ADMIN OPTION

Specify `WITH ADMIN OPTION` to enable the grantee to:

- Grant the privilege or role to another user or role, unless the role is a `GLOBAL` role
- Revoke the privilege or role from another user or role
- Alter the privilege or role to change the authorization needed to access it
- Drop the privilege or role
- Grant the role to a program unit in the grantee's schema.
- Revoke the role from a program unit in the grantee's schema.

If you grant a system privilege or role to a user without specifying WITH ADMIN OPTION, and then subsequently grant the privilege or role to the user WITH ADMIN OPTION, then the user has the ADMIN OPTION on the privilege or role.

To revoke the ADMIN OPTION on a system privilege or role from a user, you must revoke the privilege or role from the user altogether and then grant the privilege or role to the user without the ADMIN OPTION.

> **See Also:**
>
> "Granting a Role with the ADMIN OPTION: Example"

WITH DELEGATE OPTION

You can specify this clause only when granting a role to a user.

Specify WITH DELEGATE OPTION to enable the grantee to:

- Grant the role to a program unit in the grantee's schema
- Revoke the role from a program unit in the grantee's schema

If you grant a role to a user without specifying WITH DELEGATE OPTION, and then subsequently grant the role to the user WITH DELEGATE OPTION, then the user has the DELEGATE OPTION on the role.

To revoke the DELEGATE OPTION on a role from a user, you must revoke the role from the user altogether and then grant the role to the user without the DELEGATE OPTION.

> **See Also:**
>
> - "Granting a Role with the DELEGATE OPTION: Example"
> - The *grant_roles_to_programs* clause for more information on granting roles to program units

Restrictions on Granting System Privileges and Roles

Privileges and roles are subject to the following restrictions:

- A privilege or role cannot appear more than once in the list of privileges and roles to be granted.
- You cannot grant a role to itself.
- You cannot grant a role IDENTIFIED GLOBALLY to anything.
- You cannot grant a role IDENTIFIED EXTERNALLY to a global user or global role.
- You cannot grant roles circularly. For example, if you grant the role banker to the role teller, then you cannot subsequently grant teller to banker.
- You cannot grant an IDENTIFIED BY role, IDENTIFIED USING role, or IDENTIFIED EXTERNALLY role to another role.

grant_object_privileges

Use these clauses to grant object privileges.

object_privilege

Specify the object privilege you want to grant. Table 18-2 lists the object privileges, organized by the type of object on which they can be granted. When you grant an object privilege on a editionable object, either to a user or to a role, the object is actualized in the edition in which the grant is made. Refer to CREATE EDITION for information on editionable object types and editions.

> **Note:**
>
> To grant `SELECT` on a view to another user, either you must own all of the objects underlying the view or you must have been granted the `SELECT` object privilege `WITH GRANT OPTION` on all of those underlying objects. This is true even if the grantee already has `SELECT` privileges on those underlying objects.
>
> To grant `READ` on a view to another user, either you must own all of the objects underlying the view or you must have been granted the `READ` or `SELECT` object privilege `WITH GRANT OPTION` on all of those underlying objects. This is true even if the grantee already has the `READ` or `SELECT` privilege on those underlying objects.

Restriction on Object Privileges

A privilege cannot appear more than once in the list of privileges to be granted.

ALL [PRIVILEGES]

Specify `ALL` to grant all the privileges for the object that you have been granted with the `GRANT OPTION`. The user who owns the schema containing an object automatically has all privileges on the object with the `GRANT OPTION`. The keyword `PRIVILEGES` is provided for semantic clarity and is optional.

column

Specify the table or view column on which privileges are to be granted. You can specify columns only when granting the `INSERT`, `REFERENCES`, or `UPDATE` privilege. If you do not list columns, then the grantee has the specified privilege on all columns in the table or view.

For information on existing column object grants, query the `USER_`, `ALL_`, or `DBA_COL_PRIVS` data dictionary view.

> **See Also:**
>
> *Oracle Database Reference* for information on the data dictionary views and "Granting Multiple Object Privileges on Individual Columns: Example"

on_object_clause

The `on_object_clause` identifies the object on which the privileges are granted. Users, directory objects, editions, data mining models, Java source and resource schema objects, and SQL translation profiles are identified separately because they reside in separate namespaces.

> **See Also:**
>
> "Granting Object Privileges to a Role: Example"

object

Specify the schema object on which the privileges are to be granted. If you do not qualify *`object`* with *`schema`*, then the database assumes the object is in your own schema. The object can be one of the following types:

- Table, view, or materialized view
- Sequence
- Procedure, function, or package
- User-defined type
- Synonym for any of the preceding items
- Directory, library, operator, or indextype
- Java source, class, or resource

You cannot grant privileges directly to a single partition of a partitioned table.

> **See Also:**
>
> "Granting Object Privileges on a Table to a User: Example", "Granting Object Privileges on a View: Example", and "Granting Object Privileges to a Sequence in Another Schema: Example"

ON USER

Specify the database user you want to grant privileges to.

Restriction on Granting Privileges on Users

You cannot grant privileges on user `PUBLIC`.

> **See Also:**
>
> "Granting an Object Privilege on a User: Example"

ON DIRECTORY

Specify the name of the directory object on which privileges are to be granted. You cannot qualify *directory_name* with a schema name.

> **See Also:**
>
> CREATE DIRECTORY and "Granting an Object Privilege on a Directory: Example"

ON EDITION

Specify the name of the edition on which the `USE` object privilege is to be granted. You cannot qualify *edition_name* with a schema name.

ON MINING MODEL

Specify the name of the mining model on which privileges are to be granted. If you do not qualify *mining_model_name* with *schema*, then the database assumes that the mining model is in your own schema.

ON JAVA SOURCE | RESOURCE

Specify the name of the Java source or resource schema object on which privileges are to be granted. If you do not qualify *object* with *schema*, then the database assumes that the object is in your own schema.

> **See Also:**
>
> CREATE JAVA

ON SQL TRANSLATION PROFILE

Specify the name of the SQL translation profile on which privileges are to be granted. If you do not qualify *profile* with *schema*, then the database assumes that the profile is in your own schema.

WITH HIERARCHY OPTION

Specify `WITH HIERARCHY OPTION` to grant the specified object privilege on all subobjects of *object*, such as subviews created under a view, including subobjects created subsequent to this statement.

This clause is meaningful only in combination with the `READ` or `SELECT` object privilege.

WITH GRANT OPTION

Specify `WITH GRANT OPTION` to enable the grantee to grant the object privileges to other users and roles.

If you grant an object privilege to a user without specifying `WITH GRANT OPTION`, and then subsequently grant the privilege to the user `WITH GRANT OPTION`, then the user has the `GRANT OPTION` on the privilege.

To revoke the `GRANT OPTION` on an object privilege from a user, you must revoke the privilege from the user altogether and then grant the privilege to the user without the `GRANT OPTION`.

Restriction on Granting WITH GRANT OPTION

You can specify `WITH GRANT OPTION` only when granting to a user or to `PUBLIC`, not when granting to a role.

grant_roles_to_programs

Use this clause to grant roles to program units. Such roles are called code based access control (CBAC) roles.

role

Specify the role you want to grant. You can grant an Oracle Database predefined role or a user-defined role. The role must have been created by or directly granted to the schema owner of the program unit.

program_unit

Specify the program unit to which the role is to be granted. You can specify a PL/SQL function, procedure, or package. If you do not specify *`schema`*, then Oracle Database assumes the function, procedure, or package is in your own schema.

> **See Also:**
>
> *Oracle Database Security Guide* for more information on granting code based access control roles to program units

CONTAINER Clause

If the current container is a pluggable database (PDB):

- Specify `CONTAINER = CURRENT` to locally grant a system privilege, object privilege, or role to a user or role. The privilege or role is granted to the user or role only in the current PDB.

If the current container is the root:

- Specify `CONTAINER = CURRENT` to locally grant a system privilege, object privilege, or role to a common user or common role. The privilege or role is granted to the user or role only in the root.
- Specify `CONTAINER = ALL` to commonly grant a system privilege, object privilege on a common object, or role, to a common user or common role.

If you omit this clause, then `CONTAINER = CURRENT` is the default.

> **Note:**
>
> If you specify the `CONTAINER` clause when granting a privilege or role, then the current container must be the same container and you must specify the same `CONTAINER` clause when you revoke the privilege or role. Refer to the CONTAINER Clause of the `REVOKE` statement for more information.

Listings of System and Object Privileges

> **Note:**
>
> When you grant a privilege on ANY object, such as CREATE ANY CLUSTER, the result is determined by the value of the O7_DICTIONARY_ACCESSIBILITY initialization parameter. By default, this parameter is set to FALSE, so that ANY privileges give the grantee access to that type of object in all schemas except the SYS schema. If you set O7_DICTIONARY_ACCESSIBILITY to TRUE, then the ANY privileges also give the grantee access, in the SYS schema, to all objects except Oracle Scheduler objects. For security reasons, Oracle recommends that you use this setting only with great caution.

Table 18-1 System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
Advisor Framework Privileges: All of the advisor framework privileges are part of the DBA role.	—
ADVISOR	Access the advisor framework through PL/SQL packages such as DBMS_ADVISOR and DBMS_SQLTUNE.
ADMINISTER SQL TUNING SET	Create, drop, select (read), load (write), and delete SQL tuning sets owned by the grantee through the DBMS_SQLTUNE package.
ADMINISTER ANY SQL TUNING SET	Create, drop, select (read), load (write), and delete SQL tuning sets owned by any user through the DBMS_SQLTUNE package.
CREATE ANY SQL PROFILE	Accept a SQL Profile recommended by the SQL Tuning Advisor, which is accessed through Enterprise Manager or by the DBMS_SQLTUNE package. **Note**: This privilege has been deprecated in favor of ADMINISTER SQL MANAGEMENT OBJECT.
ALTER ANY SQL PROFILE	Alter the attributes of an existing SQL Profile. **Note**: This privilege has been deprecated in favor of ADMINISTER SQL MANAGEMENT OBJECT.
DROP ANY SQL PROFILE	Drop existing SQL Profiles. **Note**: This privilege has been deprecated in favor of ADMINISTER SQL MANAGEMENT OBJECT.
ADMINISTER SQL MANAGEMENT OBJECT	Create, alter, and drop SQL Profiles owned by any user through the DBMS_SQLTUNE package.
ANALYTIC VIEWS	—
CREATE ANALYTIC VIEW	Create analytic views in the grantee's schema.
CREATE ANY ANALYTIC VIEW	Create analytic views in any schema except SYS, AUDSYS.
ALTER ANY ANALYTIC VIEW	Rename analytic views in any schema except SYS, AUDSYS.
DROP ANY ANALYTIC VIEW	Drop analytic views in any schema except SYS, AUDSYS .
ATTRIBUTE DIMENSIONS	—

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
`CREATE ATTRIBUTE DIMENSION`	Create attribute dimensions in the grantee's schema.
`CREATE ANY ATTRIBUTE DIMENSION`	Create attribute dimensions in any schema except `SYS,AUDSYS`.
`ALTER ANY ATTRIBUTE DIMENSION`	Rename attribute dimensions in any schema except `SYS,AUDSYS`.
`DROP ANY ATTRIBUTE DIMENSION`	Drop attribute dimensions in any schema except `SYS,AUDSYS`.
CLUSTERS:	—
`CREATE CLUSTER`	Create clusters in the grantee's schema.
`CREATE ANY CLUSTER`	Create clusters in any schema except `SYS,AUDSYS`. Behaves similarly to `CREATE ANY TABLE`.
`ALTER ANY CLUSTER`	Alter clusters in any schema except `SYS`, `AUDSYS`.
`DROP ANY CLUSTER`	Drop clusters in any schema except `SYS,AUDSYS`.
CONTEXTS:	—
`CREATE ANY CONTEXT`	Create any context namespace.
`DROP ANY CONTEXT`	Drop any context namespace.
DATA REDACTION:	—
`EXEMPT REDACTION POLICY`	Bypass any existing Oracle Data Redaction policies and view actual data from tables or views on which Data Redaction policies are defined.
DATABASE:	—
`ALTER DATABASE`	Alter the database.
`ALTER SYSTEM`	Issue `ALTER SYSTEM` statements.
`AUDIT SYSTEM`	Issue `AUDIT` statements.
DATABASE LINKS:	—
`CREATE DATABASE LINK`	Create private database links in the grantee's schema.
`CREATE PUBLIC DATABASE LINK`	Create public database links.
`ALTER DATABASE LINK`	Modify a fixed-user database link when the password of the connection or authentication user changes.
`ALTER PUBLIC DATABASE LINK`	Modify a public fixed-user database link when the password of the connection or authentication user changes.
`DROP PUBLIC DATABASE LINK`	Drop public database links.
DEBUGGING:	—
`DEBUG CONNECT SESSION`	Connect the current session to a debugger.
`DEBUG ANY PROCEDURE`	Debug all PL/SQL and Java code in any database object. Display information on all SQL statements executed by the application. **Note:** Granting this privilege is equivalent to granting the `DEBUG` object privilege on all applicable objects in the database.
DICTIONARIES:	—
`ANALYZE ANY DICTIONARY`	Analyze any data dictionary object.
DIMENSIONS:	—

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
CREATE DIMENSION	Create dimensions in the grantee's schema.
CREATE ANY DIMENSION	Create dimensions in any schema except SYS,AUDSYS.
ALTER ANY DIMENSION	Alter dimensions in any schema except SYS,AUDSYS.
DROP ANY DIMENSION	Drop dimensions in any schema except SYS,AUDSYS.
DIRECTORIES:	—
CREATE ANY DIRECTORY	Create directory database objects.
DROP ANY DIRECTORY	Drop directory database objects.
EDITIONS:	—
CREATE ANY EDITION	Create editions.
DROP ANY EDITION	Drop editions.
FLASHBACK DATA ARCHIVES:	—
FLASHBACK ARCHIVE ADMINISTER	Create, alter, or drop any flashback data archive.
HIERARCHIES	—
CREATE HIERARCHY	Create hierarchies in the grantee's schema.
CREATE ANY HIERARCHY	Create hierarchies in any schema except SYS,AUDSYS.
ALTER ANY HIERARCHY	Rename hierarchies in any schema except SYS,AUDSYS.
DROP ANY HIERARCHY	Drop hierarchies in any schema except SYS, AUDSYS.
INDEXES:	—
CREATE ANY INDEX	Create in any schema, except SYS, AUDSYS, a domain index or an index on any table in any schema except SYS,AUDSYS.
ALTER ANY INDEX	Alter indexes in any schema except SYS,AUDSYS.
DROP ANY INDEX	Drop indexes in any schema except SYS,AUDSYS.
INDEXTYPES:	—
CREATE INDEXTYPE	Create indextypes in the grantee's schema.
CREATE ANY INDEXTYPE	Create indextypes in any schema except SYS and create comments on indextypes in any schema except SYS.
ALTER ANY INDEXTYPE	Modify indextypes in any schema except SYS,AUDSYS.
DROP ANY INDEXTYPE	Drop indextypes in any schema except SYS,AUDSYS.
EXECUTE ANY INDEXTYPE	Reference indextypes in any schema except SYS,AUDSYS.
JOB SCHEDULER OBJECTS:	The following privileges are needed to execute procedures in the DBMS_SCHEDULER package. This privileges do not apply to lightweight jobs, which are not database objects. Refer to *Oracle Database Administrator's Guide* for more information about lightweight jobs.
CREATE JOB	Create, alter, or drop jobs, chains, schedules, programs, credentials, resource objects, or incompatibility resource objects in the grantee's schema.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
CREATE ANY JOB	Create, alter, or drop jobs, chains, schedules, programs, credentials, resource objects, or incompatibility resource objects in any schema except SYS,AUDSYS. **Note:** This extremely powerful privilege allows the grantee to execute code as any other user. It should be granted with caution.
CREATE EXTERNAL JOB	Create in the grantee's schema an executable scheduler job that runs on the operating system.
EXECUTE ANY CLASS	Specify any job class in a job in the grantee's schema.
EXECUTE ANY PROGRAM	Use any program in a job in the grantee's schema.
MANAGE SCHEDULER	Create, alter, or drop any job class, window, or window group.
USE ANY JOB RESOURCE	Associate any schedule resource object with any program or job in the grantee's schema.
KEY MANAGEMENT FRAMEWORK:	—
ADMINISTER KEY MANAGEMENT	Manage keys and keystores.
LIBRARIES:	**Caution:** CREATE LIBARARY, CREATE ANY LIBRARY, ALTER ANY LIBRARY, and EXECUTE ANY LIBRARY are extremely powerful privileges that should be granted only to trusted users. Refer to *Oracle Database Security Guide* before granting these privileges.
CREATE LIBRARY	Create external procedure or function libraries in the grantee's schema.
CREATE ANY LIBRARY	Create external procedure or function libraries in any schema except SYS,AUDSYS.
ALTER ANY LIBRARY	Alter external procedure or function libraries in any schema except SYS,AUDSYS.
DROP ANY LIBRARY	Drop external procedure or function libraries in any schema except SYS,AUDSYS.
EXECUTE ANY LIBRARY	Use external procedure or function libraries in any schema except SYS,AUDSYS.
LOGMINER:	—
LOGMINING	Execute procedures in the DBMS_LOGMNR package in a CDB. Query the contents of the V$LOGMNR_CONTENTS view.
MATERIALIZED VIEWS:	—
CREATE MATERIALIZED VIEW	Create materialized views in the grantee's schema.
CREATE ANY MATERIALIZED VIEW	Create materialized views in any schema except SYS,AUDSYS.
ALTER ANY MATERIALIZED VIEW	Alter materialized views in any schema except SYS,AUDSYS.
DROP ANY MATERIALIZED VIEW	Drop materialized views in any schema except SYS,AUDSYS.
QUERY REWRITE	This privilege has been deprecated. No privileges are needed for a user to enable rewrite for a materialized view that references tables or views in the user's own schema.
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view when that materialized view references tables or views in any schema except SYS.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
ON COMMIT REFRESH	Create a refresh-on-commit materialized view on any table in the database. Alter a refresh-on-demand materialized view on any table in the database to refresh-on-commit.
FLASHBACK ANY TABLE	Issue a SQL Flashback Query on any table, view, or materialized view in any schema except SYS. This privilege is not needed to execute the DBMS_FLASHBACK procedures.
MINING MODELS:	—
CREATE MINING MODEL	Create mining models in the grantee's schema using the DBMS_DATA_MINING.CREATE_MODEL procedure.
CREATE ANY MINING MODEL	Create mining models in any schema, except SYS, AUDSYS, using the DBMS_DATA_MINING.CREATE_MODEL procedure.
ALTER ANY MINING MODEL	Change the mining model name or the associated cost matrix of a model in any schema, except SYS, AUDSYS, using the applicable DBMS_DATA_MINING procedures.
DROP ANY MINING MODEL	Drop mining models in any schema, except SYS,AUDSYS, using the DBMS_DATA_MINING.DROP_MODEL procedure.
SELECT ANY MINING MODEL	Score or view mining models in any schema except SYS, AUDSYS. Scoring is done either with the PREDICTION family of SQL functions or with the DBMS_DATA_MINING.APPLY procedure. Viewing the model is done with the DBMS_DATA_MINING.GET_MODEL_DETAILS_* procedures.
COMMENT ANY MINING MODEL	Create comments on mining models in any schema, except SYS, AUDSYS, using the SQL COMMENT statement.
OLAP CUBES:	The following privileges are valid when you are using Oracle Database with the OLAP option.
CREATE CUBE	Create OLAP cubes in the grantee's schema.
CREATE ANY CUBE	Create OLAP cubes in any schema except SYS,AUDSYS.
ALTER ANY CUBE	Alter OLAP cubes in any schema except SYS,AUDSYS.
DROP ANY CUBE	Drop OLAP cubes in any schema except SYS,AUDSYS.
SELECT ANY CUBE	Query or view OLAP cubes in any schema except SYS,AUDSYS.
UPDATE ANY CUBE	Update OLAP cubes in any schema except SYS,AUDSYS.
OLAP CUBE MEASURE FOLDERS:	The following privileges are valid when you are using Oracle Database with the OLAP option.
CREATE MEASURE FOLDER	Create OLAP measure folders in the grantee's schema.
CREATE ANY MEASURE FOLDER	Create OLAP measure folders in any schema except SYS,AUDSYS.
DELETE ANY MEASURE FOLDER	Delete a measure from an OLAP measure folder in any schema except SYS,AUDSYS.
DROP ANY MEASURE FOLDER	Drop OLAP measure folders in any schema except SYS,AUDSYS.
INSERT ANY MEASURE FOLDER	Insert a measure into an OLAP measure folder in any schema except SYS,AUDSYS.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
OLAP CUBE DIMENSIONS:	The following privileges are valid when you are using Oracle Database with the OLAP option.
`CREATE CUBE DIMENSION`	Create OLAP cube dimension in the grantee's schema.
`CREATE ANY CUBE DIMENSION`	Create OLAP cube dimensions in any schema except `SYS,AUDSYS`.
`ALTER ANY CUBE DIMENSION`	Alter OLAP cube dimensions in any schema except `SYS,AUDSYS`.
`DELETE ANY CUBE DIMENSION`	Delete from OLAP cube dimensions in any schema except `SYS`, `AUDSYS`.
`DROP ANY CUBE DIMENSION`	Drop OLAP cube dimensions in any schema except `SYS,AUDSYS`.
`INSERT ANY CUBE DIMENSION`	Insert into OLAP cube dimensions in any schema except `SYS,AUDSYS`.
`SELECT ANY CUBE DIMENSION`	View or query OLAP cube dimensions in any schema except `SYS,AUDSYS`.
`UPDATE ANY CUBE DIMENSION`	Update OLAP cube dimensions in any schema except `SYS,AUDSYS`.
OLAP CUBE BUILD PROCESSES:	—
`CREATE CUBE BUILD PROCESS`	Create OLAP cube build processes in the grantee's schema.
`CREATE ANY CUBE BUILD PROCESS`	Create OLAP cube build processes in any schema except `SYS,AUDSYS`.
`DROP ANY CUBE BUILD PROCESS`	Drop OLAP cube build processes in any schema except `SYS,AUDSYS`.
`UPDATE ANY CUBE BUILD PROCESS`	Update OLAP cube build processes in any schema except `SYS,AUDSYS`.
OPERATORS:	—
`CREATE OPERATOR`	Create an operator and its bindings in the grantee's schema.
`CREATE ANY OPERATOR`	Create an operator and its bindings in any schema and create a comment on an operator in any schema.
`ALTER ANY OPERATOR`	Modify operators in any schema.
`DROP ANY OPERATOR`	Drop operators in any schema.
`EXECUTE ANY OPERATOR`	Reference operators in any schema.
OUTLINES:	—
`CREATE ANY OUTLINE`	Create public outlines that can be used in any schema that uses outlines.
`ALTER ANY OUTLINE`	Modify outlines.
`DROP ANY OUTLINE`	Drop outlines.
PDB LOCKDOWN PROFILES:	—
`CREATE LOCKDOWN PROFILE`	Create PDB lockdown profiles.
`ALTER LOCKDOWN PROFILE`	Alter PDB lockdown profiles.
`DROP LOCKDOWN PROFILE`	Drop PDB lockdown profiles.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
PLAN MANAGEMENT:	—
ADMINISTER SQL MANAGEMENT OBJECT	Perform controlled manipulation of plan history and SQL plan baselines maintained for various SQL statements.
PLUGGABLE DATABASES:	—
CREATE PLUGGABLE DATABASE	Create a PDB. Plug in a PDB that was previously unplugged from a CDB. Clone a PDB.
SET CONTAINER	Allow a common user to switch into the container for which this privilege was granted. This privilege can be granted only to a common user or common role.
PROCEDURES:	—
CREATE PROCEDURE	Create stored procedures, functions, or packages in the grantee's schema.
CREATE ANY PROCEDURE	Create stored procedures, functions, or packages in any schema except SYS,AUDSYS.
ALTER ANY PROCEDURE	Alter stored procedures, functions, or packages in any schema except SYS,AUDSYS.
DROP ANY PROCEDURE	Drop stored procedures, functions, or packages in any schema except SYS,AUDSYS.
EXECUTE ANY PROCEDURE	Execute procedures or functions, either standalone or packaged. Reference public package variables in any schema except SYS,AUDSYS.
INHERIT ANY REMOTE PRIVILEGES	Execute definer's rights procedures or functions that contain current user database links.
PROFILES:	—
CREATE PROFILE	Create profiles.
ALTER PROFILE	Alter profiles.
DROP PROFILE	Drop profiles.
ROLES:	—
CREATE ROLE	Create roles.
ALTER ANY ROLE	Alter any role in the database.
DROP ANY ROLE	Drop roles.
GRANT ANY ROLE	Grant any role in the database.
ROLLBACK SEGMENTS:	—
CREATE ROLLBACK SEGMENT	Create rollback segments.
ALTER ROLLBACK SEGMENT	Alter rollback segments.
DROP ROLLBACK SEGMENT	Drop rollback segments.
SEQUENCES:	—
CREATE SEQUENCE	Create sequences in the grantee's schema.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
`CREATE ANY SEQUENCE`	Create sequences in any schema except `SYS,AUDSYS`.
`ALTER ANY SEQUENCE`	Alter sequences in any schema except `SYS,AUDSYS`.
`DROP ANY SEQUENCE`	Drop sequences in any schema except `SYS,AUDSYS`.
`SELECT ANY SEQUENCE`	Reference sequences in any schema except `SYS,AUDSYS`.
SESSIONS:	—
`CREATE SESSION`	Connect to the database.
`ALTER RESOURCE COST`	Set costs for session resources.
`ALTER SESSION`	Enable and disable the SQL trace facility.
`RESTRICTED SESSION`	Logon after the instance is started using the SQL*Plus `STARTUP RESTRICT` statement.
SNAPSHOTS:	See `MATERIALIZED VIEWS`
SQL TRANSLATION PROFILES:	—
`CREATE SQL TRANSLATION PROFILE`	Create SQL translation profiles in the grantee's schema.
`CREATE ANY SQL TRANSLATION PROFILE`	Create SQL translation profiles in any schema except `SYS,AUDSYS`.
`ALTER ANY SQL TRANSLATION PROFILE`	Alter the translator, custom SQL statement translations, or custom error translations of a SQL translation profile in any schema except `SYS,AUDSYS`.
`USE ANY SQL TRANSLATION PROFILE`	Use SQL translation profiles in any schema except `SYS,AUDSYS`.
`DROP ANY SQL TRANSLATION PROFILE`	Drop SQL translation profiles in any schema except `SYS,AUDSYS`.
`TRANSLATE ANY SQL`	Translate SQL through the grantee's SQL translation profile for any user.
SYNONYMS:	**Caution:** `CREATE PUBLIC SYNONYM` and `DROP PUBLIC SYNONYM` are extremely powerful privileges that should be granted only to trusted users. Refer to *Oracle Database Security Guide* before granting these privileges.
`CREATE SYNONYM`	Create synonyms in the grantee's schema.
`CREATE ANY SYNONYM`	Create private synonyms in any schema except `SYS,AUDSYS`.
`CREATE PUBLIC SYNONYM`	Create public synonyms.
`DROP ANY SYNONYM`	Drop private synonyms in any schema except `SYS,AUDSYS`.
`DROP PUBLIC SYNONYM`	Drop public synonyms.
TABLES:	**Note:** For external tables, the only valid privileges are `CREATE ANY TABLE`, `ALTER ANY TABLE`, `DROP ANY TABLE`, `READ ANY TABLE`, and `SELECT ANY TABLE`.
`CREATE TABLE`	Create tables in the grantee's schema.
`CREATE ANY TABLE`	Create a table in any schema except `SYS,AUDSYS`. The owner of the schema containing the table must have space quota on the tablespace to contain the table.
`ALTER ANY TABLE`	Alter a table or view in any schema except `SYS, AUDSYS`.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
`BACKUP ANY TABLE`	Use the Export utility to incrementally export objects from the schema of other users except `SYS,AUDSYS`.
`DELETE ANY TABLE`	Delete rows from tables, table partitions, or views in any schema except `SYS,AUDSYS`.
`DROP ANY TABLE`	Drop or truncate tables or table partitions in any schema except `SYS,AUDSYS`.
`INSERT ANY TABLE`	Insert rows into tables and views in any schema except `SYS,AUDSYS`.
`LOCK ANY TABLE`	Lock tables and views in any schema except `SYS,AUDSYS`.
`READ ANY TABLE`	Query tables, views, or materialized views in any schema except `SYS,AUDSYS`.
`SELECT ANY TABLE`	Query tables, views, or materialized views in any schema except `SYS,AUDSYS`. Obtain row locks using a `SELECT` ... `FOR UPDATE`.
`FLASHBACK ANY TABLE`	Issue a SQL Flashback Query on any table, view, or materialized view in any schema except `SYS,AUDSYS`. This privilege is not needed to execute the `DBMS_FLASHBACK` procedures.
`UPDATE ANY TABLE`	Update rows in tables and views in any schema except `SYS,AUDSYS`.
`REDEFINE ANY TABLE`	Perform online redefinition without granting any of the privileges in `USER` or `FULL` mode.
TABLESPACES:	—
`CREATE TABLESPACE`	Create tablespaces.
`ALTER TABLESPACE`	Alter tablespaces.
`DROP TABLESPACE`	Drop tablespaces.
`MANAGE TABLESPACE`	Take tablespaces offline and online and begin and end tablespace backups.
`UNLIMITED TABLESPACE`	Use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, then the user's schema objects remain but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
TRIGGERS:	—
`CREATE TRIGGER`	Create database triggers in the grantee's schema.
`CREATE ANY TRIGGER`	Create database triggers in any schema except `SYS, AUDSYS`.
`ALTER ANY TRIGGER`	Enable, disable, or compile database triggers in any schema except `SYS,AUDSYS`.
`DROP ANY TRIGGER`	Drop database triggers in any schema except `SYS,AUDSYS`.
`ADMINISTER DATABASE TRIGGER`	Create a trigger on `DATABASE`. You must also have the `CREATE TRIGGER` or `CREATE ANY TRIGGER` system privilege.
TYPES:	—
`CREATE TYPE`	Create object types and object type bodies in the grantee's schema.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
CREATE ANY TYPE	Create object types and object type bodies in any schema except SYS,AUDSYS.
ALTER ANY TYPE	Alter object types in any schema except SYS,AUDSYS.
DROP ANY TYPE	Drop object types and object type bodies in any schema except SYS,AUDSYS.
EXECUTE ANY TYPE	Use and reference object types and collection types in any schema except SYS,AUDSYS, and invoke methods of an object type in any schema, except SYS,AUDSYS, if you make the grant to a specific user. If you grant EXECUTE ANY TYPE to a role, then users holding the enabled role will not be able to invoke methods of an object type in any schema.
UNDER ANY TYPE	Create subtypes under any nonfinal object types.
USERS:	—
CREATE USER	Create users. This privilege also allows the creator to: • Assign quotas on any tablespace. • Set default and temporary tablespaces. • Assign a profile as part of a CREATE USER statement.
ALTER USER	Alter any user except SYS. This privilege authorizes the grantee to: • Change another user's password or authentication method. • Assign quotas on any tablespace. • Set default and temporary tablespaces. • Assign a profile and default roles.
DROP USER	Drop users
VIEWS:	—
CREATE VIEW	Create views in the grantee's schema.
CREATE ANY VIEW	Create views in any schema except SYS,AUDSYS.
DROP ANY VIEW	Drop views in any schema except SYS,AUDSYS.
UNDER ANY VIEW	Create subviews under any object views.
FLASHBACK ANY TABLE	Issue a SQL Flashback Query on any table, view, or materialized view in any schema except SYS,AUDSYS. This privilege is not needed to execute the DBMS_FLASHBACK procedures.
MERGE ANY VIEW	If a user has been granted the MERGE ANY VIEW privilege, then for any query issued by that user, the optimizer can use view merging to improve query performance without performing the checks that would otherwise be performed to ensure that view merging does not violate any security intentions of the view creator. See also *Oracle Database Reference* for information on the OPTIMIZER_SECURE_VIEW_MERGING parameter and *Oracle Database SQL Tuning Guide* for information on view merging.
MISCELLANEOUS:	—
ANALYZE ANY	Analyze a table, cluster, or index in any schema except SYS.
AUDIT ANY	Audit an object in any schema, except SYS,AUDSYS, using AUDIT *schema_objects* statements.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
BECOME USER	Allow users of the Data Pump Import utility (impdp) and the original Import utility (imp) to assume the identity of another user in order to perform operations that cannot be directly performed by a third party (for example, loading objects such as object privilege grants). Allow Streams administrators to create or alter capture users and apply users in a Streams environment. By default this privilege is part of the DBA role. Database Vault removes this privileges from the DBA role. Therefore, this privilege is needed by Streams only in an environment where Database Vault is installed.
CHANGE NOTIFICATION	Create a registration on queries and receive database change notifications in response to DML or DDL changes to the objects associated with the registered queries. Refer to *Oracle Database Development Guide* for more information on database change notification.
COMMENT ANY TABLE	Comment on a table, view, or column in any schema except SYS,AUDSYS.
EXEMPT ACCESS POLICY	Bypass fine-grained access control. **Caution:** This is a very powerful system privilege, as it lets the grantee bypass application-driven security policies. Database administrators should use caution when granting this privilege.
FORCE ANY TRANSACTION	Force the commit or rollback of any in-doubt distributed transaction in the local database. Induce the failure of a distributed transaction.
FORCE TRANSACTION	Force the commit or rollback of the grantee's in-doubt distributed transactions in the local database.
GRANT ANY OBJECT PRIVILEGE	Grant any object privilege that the object owner is permitted to grant. Revoke any object privilege that was granted by the object owner or by some other user with the GRANT ANY OBJECT PRIVILEGE privilege.
GRANT ANY PRIVILEGE	Grant any system privilege.
INHERIT ANY PRIVILEGES	Execute invoker's rights procedures owned by the grantee with the privileges of the invoker.
KEEP DATE TIME	The SYSDATE and SYSTIMESTAMP functions return their original values during replay for Application Continuity when the grantee is running the application. This privilege is useful for providing bind variable consistency after recoverable errors. **Note:** If this privilege is granted or revoked between runtime and failover of a request, then the original values are not returned during replay for Application Continuity for that request.
KEEP SYSGUID	The SYS_GUID function returns its original value during replay for Application Continuity when the grantee is running the application. This privilege is useful for providing bind variable consistency after recoverable errors. **Note:** If this privilege is granted or revoked between runtime and failover of a request, then the original value is not returned during replay for Application Continuity for that request.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
PURGE DBA_RECYCLEBIN	Remove all objects from the system-wide recycle bin.
RESUMABLE	Enable resumable space allocation.
SELECT ANY DICTIONARY	Query any data dictionary object in the SYS schema, with the exception of the following objects: DEFAULT_PWD$, ENC$, LINK$, USER$, USER_HISTORY$, and XS$VERIFIERS. This privilege lets you selectively override the default FALSE setting of the O7_DICTIONARY_ACCESSIBILITY initialization parameter.
SELECT ANY TRANSACTION	Query the contents of the FLASHBACK_TRANSACTION_QUERY view. **Caution:** This is a very powerful system privilege, as it lets the grantee view all data in the database, including past data. This privilege should be granted only to users who need to use the Oracle Flashback Transaction Query feature.
SYSBACKUP	Perform the following backup and recovery operations: STARTUP and SHUTDOWN. CREATE CONTROLFILE. CREATE PFILE and CREATE SPFILE. FLASHBACK DATABASE. Create, use, view, and drop restore points (including guaranteed restore points). Execute procedures in the DBMS_DATAPUMP, DBMS_PIPE, DBMS_TDB, and DBMS_TTS packages. SELECT on X$ tables, V$ views, and GV$ views. Includes the ALTER DATABASE, ALTER SESSION, ALTER SYSTEM, ALTER TABLESPACE, CREATE ANY CLUSTER, CREATE ANY DIRECTORY, CREATE ANY TABLE, CREATE SESSION, DROP DATABASE, DROP TABLESPACE, RESUMABLE, SELECT ANY DICTIONARY, SELECT ANY TRANSACTION, UNLIMITED TABLESPACE privileges and the SELECT_CATALOG_ROLE role.
SYSDBA	STARTUP and SHUTDOWN. ALTER DATABASE: open, mount, back up, or change character set. CREATE DATABASE. DROP DATABASE. ARCHIVELOG and RECOVERY. CREATE SPFILE. Includes the RESTRICTED SESSION privilege.

Table 18-1 (Cont.) System Privileges (Organized by the Database Object Operated Upon)

System Privilege Name	Operations Authorized
SYSDG	Perform the following Oracle Data Guard operations: STARTUP and SHUTDOWN. FLASHBACK DATABASE. Create, use, view, and drop restore points (including guaranteed restore points). SELECT on X$ tables, V$ views, and GV$ views. Includes the ALTER DATABASE, ALTER SESSION, ALTER SYSTEM, CREATE SESSION, and SELECT ANY DICTIONARY privileges.
SYSKM	Perform the following encryption key management operations: Connect to the database even if the database is not open. SELECT on the following views when the database is open: V$CLIENT_SECRETS, V$ENCRYPTED_TABLESPACES, V$ENCRYPTION_KEYS, V$ENCRYPTION_WALLET and V$WALLET. Includes the ADMINISTER KEY MANAGEMENT and CREATE SESSION privileges.
SYSOPER	STARTUP and SHUTDOWN operations. ALTER DATABASE: open, mount, or back up. ARCHIVELOG and RECOVERY. CREATE SPFILE. Includes the RESTRICTED SESSION privilege.

Table 18-2 Object Privileges (Organized by the Database Object Operated Upon)

Object Privilege Name	Operations Authorized
ANALYTIC VIEW PRIVILEGES	The following **analytic view privileges** authorize operations on analytic views.
ALTER	Rename the analytic view.
READ	Query the object with the SELECT statement.
SELECT	Query the object with the SELECT statement.
ATTRIBUTE DIMENSION PRIVILEGES	The following **attribute dimension privileges** authorize operations on attribute dimensions..
ALTER	Rename the attribute dimension.
DIRECTORY PRIVILEGES	The following **directory privileges** provide secured access to the files stored in the operating system directory to which the directory object serves as a pointer. The directory object contains the full path name of the operating system directory where the files reside. Because the files are actually stored outside the database, Oracle Database server processes also need to have appropriate file permissions on the file system server. Granting object privileges on the directory database object to individual database users, rather than on the operating system, allows the database to enforce security during file operations.
READ	Read files in the directory.

Table 18-2 (Cont.) Object Privileges (Organized by the Database Object Operated Upon)

Object Privilege Name	Operations Authorized
`WRITE`	Write files in the directory. This privilege is useful only in connection with external tables. It allows the grantee to determine whether the external table agent can write a log file or a bad file to the directory. **Restriction:** This privilege does not allow the grantee to write to a `BFILE`.
`EXECUTE`	Execute a preprocessor program that resides in the directory. A preprocessor program converts data to a supported format when loading data records from an external table with the `ORACLE_LOADER` access driver. Refer to *Oracle Database Utilities* for more information. This privilege does not implicitly allow READ access on the external table data.
EDITION PRIVILEGE	The following **edition privilege** authorizes the use of an edition.
`USE`	Use an edition.
FLASHBACK DATA ARCHIVE PRIVILEGE	The following **flashback data archive privilege** authorizes operations on flashback data archives.
`FLASHBACK ARCHIVE`	Enable or disable historical tracking for a table.
HIERARCHY PRIVILEGES	The following **hierarchy privileges** authorize operations on hierarchies.
`ALTER`	Rename the hierarchy.
`READ`	Query the object with the `SELECT` statement.
`SELECT`	Query the object with the `SELECT` statement.
INDEXTYPE PRIVILEGE	The following **indextype privilege** authorizes operations on indextypes.
`EXECUTE`	Reference an indextype.
LIBRARY PRIVILEGE	The following **library privilege** authorizes operations on a library.
`EXECUTE`	Use and reference the specified object and invoke its methods. **Caution:** This extremely powerful privilege should be granted only to trusted users. Refer to *Oracle Database Security Guide* before granting this privilege.
MATERIALIZED VIEW PRIVILEGES	The following **materialized view privileges** authorize operations on a materialized view. The `DELETE`, `INSERT`, and `UPDATE` privileges can be granted only to updatable materialized views.
`ON COMMIT REFRESH`	Create a refresh-on-commit materialized view on the specified table.
`QUERY REWRITE`	Create a materialized view for query rewrite using the specified table.
`READ`	Query the materialized view.
`SELECT`	Query the materialized view. Obtain row locks with the `SELECT` ... `FOR UPDATE` or `LOCK TABLE` statement.
MINING MODEL PRIVILEGES	The following mining model privileges authorize operations on a mining model. These privileges are not required for models within the users own schema.
`ALTER`	Change the mining model name or the associated cost matrix using the applicable `DBMS_DATA_MINING` procedures.
`SELECT`	Score or view the mining model. Scoring is done with the `PREDICTION` family of SQL functions or with the `DBMS_DATA_MINING.APPLY` procedure. Viewing the model is done with the `DBMS_DATA_MINING.GET_MODEL_DETAILS_*` procedures.
OBJECT TYPE PRIVILEGES	The following **object type privileges** authorize operations on a database object type.

Table 18-2 (Cont.) Object Privileges (Organized by the Database Object Operated Upon)

Object Privilege Name	Operations Authorized
`DEBUG`	Access, through a debugger, all public and nonpublic variables, methods, and types defined on the object type. Place a breakpoint or stop at a line or instruction boundary within the type body.
`EXECUTE`	Use and reference the specified object and invoke its methods. Access, through a debugger, public variables, types, and methods defined on the object type.
`UNDER`	Create a subtype under this type. You can grant this object privilege only if you have the `UNDER ANY TYPE` privilege `WITH GRANT OPTION` on the immediate supertype of this type.
OLAP PRIVILEGES	The following object privileges are valid if you are using Oracle Database with the OLAP option.
`INSERT`	Insert members into the OLAP cube dimension or measures into the measures folder.
`ALTER`	Change the definition of the OLAP cube dimension or cube.
`DELETE`	Delete members from the OLAP cube dimension or measures from the measures folder.
`SELECT`	View or query the OLAP cube or cube dimension.
`UPDATE`	Update measure values of the OLAP cube or attribute values of the cube dimension.
OPERATOR PRIVILEGE	The following **operator privilege** authorizes operations on user-defined operators.
`EXECUTE`	Reference an operator.
PROCEDURE, FUNCTION, PACKAGE PRIVILEGES	The following **procedure, function, and package privileges** authorize operations on procedures, functions, and packages. These privileges also apply to **Java sources, classes, and resources**, which Oracle Database treats as though they were procedures for purposes of granting object privileges.
`DEBUG`	Access, through a debugger, all public and nonpublic variables, methods, and types defined on the object. Place a breakpoint or stop at a line or instruction boundary within the procedure, function, or package. This privilege grants access to the declarations in the method or package specification and body.

Table 18-2 (Cont.) Object Privileges (Organized by the Database Object Operated Upon)

Object Privilege Name	Operations Authorized
`EXECUTE`	Execute the procedure or function directly, or access any program object declared in the specification of a package, or compile the object implicitly during a call to a currently invalid or uncompiled function or procedure. This privilege does not allow the grantee to explicitly compile using `ALTER PROCEDURE` or `ALTER FUNCTION`. For explicit compilation you need the appropriate `ALTER` system privilege. Access, through a debugger, public variables, types, and methods defined on the procedure, function, or package. This privilege grants access to the declarations in the method or package specification only. Job scheduler objects are created using the `DBMS_SCHEDULER` package. After these objects are created, you can grant the `EXECUTE` object privilege on job scheduler classes and programs. You can also grant `ALTER` privilege on job scheduler jobs, programs, and schedules. **Note:** Users do not need this privilege to execute a procedure, function, or package indirectly.
SCHEDULER PRIVILEGES	Job scheduler objects are created using the `DBMS_SCHEDULER` package. After these objects are created, you can grant the following privileges.
`EXECUTE`	Operations on job classes, programs, chains, and credentials.
`ALTER`	Modifications to jobs, programs, chains, credentials, and schedules.
`USE`	Associate the specified scheduler resource object with programs and jobs.
SEQUENCE PRIVILEGES	The following **sequence privileges** authorize operations on a sequence.
`ALTER`	Change the sequence definition with the `ALTER SEQUENCE` statement.
`KEEP SEQUENCE`	The sequence pseudocolumn `NEXTVAL` retains its original value during replay for Application Continuity when the grantee is running the application. This privilege is useful for providing bind variable consistency when replaying after recoverable errors. If this privilege is granted or revoked between runtime and failover of a request, then the original value of `NEXTVAL` is not retained during replay for Application Continuity for that request. **Note:** This privilege is not granted by the `GRANT ALL PRIVILEGES ON` *`sequence`* statement. You must explicitly grant this privilege. **Note:** This privilege is part of the DBA role.
`SELECT`	Examine and increment values of the sequence with the `CURRVAL` and `NEXTVAL` pseudocolumns.
SQL TRANSLATION PROFILE PRIVILEGES	The following **SQL translation profile privileges** authorize operations on a SQL translation profile.
`ALTER`	Alter the translator, custom SQL statement translations, or custom error translations of a SQL translation profile.
`USE`	Use a SQL translation profile.
SYNONYM PRIVILEGES	**Synonym privileges** are the same as the privileges for the target object. Granting a privilege on a synonym is equivalent to granting the privilege on the base object. Similarly, granting a privilege on a base object is equivalent to granting the privilege on all synonyms for the object. If you grant to a user a privilege on a synonym, then the user can use either the synonym name or the base object name in the SQL statement that exercises the privilege.

Table 18-2 (Cont.) Object Privileges (Organized by the Database Object Operated Upon)

Object Privilege Name	Operations Authorized
TABLE PRIVILEGES	The following **table privileges** authorize operations on a table. Any one of following object privileges, except the `READ` privilege, allows the grantee to lock the table in any lock mode with the `LOCK TABLE` statement. **Note:** For external tables, the only valid object privileges are `ALTER`, `READ`, and `SELECT`.
`ALTER`	Change the table definition with the `ALTER TABLE` statement.
`DEBUG`	Access, through a debugger: • PL/SQL code in the body of any triggers defined on the table • Information on SQL statements that reference the table directly
`DELETE`	Remove rows from the table with the `DELETE` statement. **Note:** You must grant the `SELECT` privilege on the table along with the `DELETE` privilege if the table is on a remote database.
`INDEX`	Create an index on the table with the `CREATE INDEX` statement.
`INSERT`	Add new rows to the table with the `INSERT` statement. **Note:** You must grant the `SELECT` privilege on the table along with the `INSERT` privilege if the table is on a remote database.
`READ`	Query the table with the `SELECT` statement. Does not allow `SELECT` ... `FOR UPDATE`.
`REFERENCES`	Create a constraint that refers to the table. You cannot grant this privilege to a role.
`SELECT`	Query the table with the `SELECT` statement, including `SELECT` ... `FOR UPDATE`.
`UPDATE`	Change data in the table with the `UPDATE` statement. **Note:** You must grant the `SELECT` privilege on the table along with the `UPDATE` privilege if the table is on a remote database.
`FLASHBACK`	To allow access to a specific table during queries, grant the `FLASHBACK` privileges on the table. Issue a SQL Flashback Query on the table.
USER PRIVILEGES	The following privileges authorize operations on a user.
`INHERIT PRIVILEGES`	Execute invoker's rights procedures or functions owned by the grantee with the privileges of the invoker when the invoker is the user on whom this privilege is granted.
`INHERIT REMOTE PRIVILEGES`	Allow the user on whom this privilege is granted to execute definer's rights procedures or functions that contain current user database links and are owned by the grantee.
`TRANSLATE SQL`	Translate SQL through the grantee's SQL translation profile for the user on whom this privilege is granted.
VIEW PRIVILEGES	The following **view privileges** authorize operations on a view. Any one of the following object privileges, except the `READ` privilege, allows the grantee to lock the view in any lock mode with the `LOCK TABLE` statement. To grant a privilege on a view, you must have that privilege with the `GRANT OPTION` on all of the base tables of the view.

Table 18-2 (Cont.) Object Privileges (Organized by the Database Object Operated Upon)

Object Privilege Name	Operations Authorized
DEBUG	Access, through a debugger: • PL/SQL code in the body of any triggers defined on the view • Information on SQL statements that reference the view directly
DELETE	Remove rows from the view with the DELETE statement.
INSERT	Add new rows to the view with the INSERT statement.
MERGE VIEW	This object privilege has the same behavior as the system privilege MERGE ANY VIEW, except that the privilege is limited to the views specified in the ON clause. For any query issued by the grantee on the specified views, the optimizer can use view merging to improve query performance without performing the checks that would otherwise be performed to ensure that view merging does not violate any security intentions of the view creator.
READ	Query the view with the SELECT statement. Does not allow SELECT ... FOR UPDATE.
REFERENCES	Define foreign key constraints on the view.
SELECT	Query the view with the SELECT statement, including SELECT ... FOR UPDATE. **See Also:** object_privilege for additional information on granting this object privilege on a view
UNDER	Create a subview under this view. You can grant this object privilege only if you have the UNDER ANY VIEW privilege WITH GRANT OPTION on the immediate superview of this view.
UPDATE	Change data in the view with the UPDATE statement.
UPDATE	To allow access to a specific view during queries, grant the FLASHBACK privileges on the view. Issue a SQL Flashback Query on the view.

Examples

Granting a System Privilege to a User: Example

To grant the CREATE SESSION system privilege to the sample user hr, allowing hr to log on to Oracle Database, issue the following statement:

```
GRANT CREATE SESSION
   TO hr;
```

Assigning User Passwords When Granting a System Privilege: Example

Assume that user hr exists and user newuser does not exist. The following statement resets the user hr password to *password1*, creates user newuser with *password2*, and grants both users the CREATE SESSION system privilege:

```
GRANT CREATE SESSION
  TO hr, newuser IDENTIFIED BY password1, password2;
```

Granting System Privileges to a Role: Example

The following statement grants appropriate system privileges to a data warehouse manager role, which was created in the "Creating a Role: Example":

```
GRANT
     CREATE ANY MATERIALIZED VIEW
   , ALTER ANY MATERIALIZED VIEW
   , DROP ANY MATERIALIZED VIEW
   , QUERY REWRITE
   , GLOBAL QUERY REWRITE
   TO dw_manager
   WITH ADMIN OPTION;
```

The `dw_manager` privilege domain now contains the system privileges related to materialized views.

Granting a Role with the ADMIN OPTION: Example

To grant the `dw_manager` role with the `ADMIN OPTION` to the sample user `sh`, issue the following statement:

```
GRANT dw_manager
   TO sh
   WITH ADMIN OPTION;
```

User `sh` can now perform the following operations with the `dw_manager` role:

- Enable the role and exercise any privileges in the privilege domain of the role, including the `CREATE MATERIALIZED VIEW` system privilege
- Grant and revoke the role to and from other users
- Drop the role
- Grant and revoke the `dw_manager` role to and from program units in the `sh` schema

Granting a Role with the DELEGATE OPTION: Example

To grant the `dw_manager` role with the `DELEGATE OPTION` to the sample user `sh`, issue the following statement:

```
GRANT dw_manager
   TO sh
   WITH DELEGATE OPTION;
```

User `sh` can now grant and revoke the `dw_manager` role to and from program units in the `sh` schema.

Granting Object Privileges to a Role: Example

The following example grants the `SELECT` object privileges to a data warehouse user role, which was created in the "Creating a Role: Example":

```
GRANT SELECT ON sh.sales TO warehouse_user;
```

Granting a Role to a Role: Example

The following statement grants the `warehouse_user` role to the `dw_manager` role. Both roles were created in the "Creating a Role: Example":

```
GRANT warehouse_user TO dw_manager;
```

The `dw_manager` role now contains all of the privileges in the domain of the `warehouse_user` role.

Granting an Object Privilege on a User: Example

To grant the INHERIT PRIVILEGES object privilege on user sh to user hr, issue the following statement:

```
GRANT INHERIT PRIVILEGES ON USER sh TO hr;
```

Granting an Object Privilege on a Directory: Example

To grant READ on directory bfile_dir to user hr, with the GRANT OPTION, issue the following statement:

```
GRANT READ ON DIRECTORY bfile_dir TO hr
   WITH GRANT OPTION;
```

Granting Object Privileges on a Table to a User: Example

To grant all privileges on the table oe.bonuses, which was created in "Merging into a Table: Example", to the user hr with the GRANT OPTION, issue the following statement:

```
GRANT ALL ON bonuses TO hr
   WITH GRANT OPTION;
```

The user hr can subsequently perform the following operations:

- Exercise any privilege on the bonuses table
- Grant any privilege on the bonuses table to another user or role

Granting Object Privileges on a View: Example

To grant SELECT and UPDATE privileges on the view emp_view, which was created in "Creating a View: Example", to all users, issue the following statement:

```
GRANT SELECT, UPDATE
   ON emp_view TO PUBLIC;
```

All users can subsequently query and update the view of employee details.

Granting Object Privileges to a Sequence in Another Schema: Example

To grant SELECT privilege on the customers_seq sequence in the schema oe to the user hr, issue the following statement:

```
GRANT SELECT
   ON oe.customers_seq TO hr;
```

The user hr can subsequently generate the next value of the sequence with the following statement:

```
SELECT oe.customers_seq.NEXTVAL
   FROM DUAL;
```

Granting Multiple Object Privileges on Individual Columns: Example

To grant to user oe the REFERENCES privilege on the employee_id column and the UPDATE privilege on the employee_id, salary, and commission_pct columns of the employees table in the schema hr, issue the following statement:

```
GRANT REFERENCES (employee_id),
      UPDATE (employee_id, salary, commission_pct)
   ON hr.employees
   TO oe;
```

The user oe can subsequently update values of the employee_id, salary, and commission_pct columns. User oe can also define referential integrity constraints that refer to the employee_id column. However, because the GRANT statement lists only these columns, oe cannot perform operations on any of the other columns of the employees table.

For example, oe can create a table with a constraint:

```
CREATE TABLE dependent
   (dependno   NUMBER,
    dependname VARCHAR2(10),
    employee   NUMBER
   CONSTRAINT in_emp REFERENCES hr.employees(employee_id) );
```

The constraint in_emp ensures that all dependents in the dependent table correspond to an employee in the employees table in the schema hr.

INSERT

Purpose

Use the INSERT statement to add rows to a table, the base table of a view, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or the base table of an object view.

Prerequisites

For you to insert rows into a table, the table must be in your own schema or you must have the INSERT object privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have the INSERT object privilege on the base table. Also, if the view is in a schema other than your own, then you must have the INSERT object privilege on the view.

If you have the INSERT ANY TABLE system privilege, then you can also insert rows into any table or the base table of any view.

You must also have the READ or SELECT object privilege on the table into which you want to insert rows if the table is on a remote database.

Conventional and Direct-Path INSERT

You can use the INSERT statement to insert data into a table, partition, or view in two ways: conventional INSERT and direct-path INSERT. When you issue a conventional INSERT statement, Oracle Database reuses free space in the table into which you are inserting and maintains referential integrity constraints. With direct-path INSERT, the database appends the inserted data after existing data in the table. Data is written directly into data files, bypassing the buffer cache. Free space in the existing data is not reused. This alternative enhances performance during insert operations and is similar to the functionality of the Oracle direct-path loader utility, SQL*Loader. When you insert into a table that has been created in parallel mode, direct-path INSERT is the default.

The manner in which the database generates redo and undo data depends in part on whether you are using conventional or direct-path INSERT:

- Conventional INSERT always generates maximal redo and undo for changes to both data and metadata, regardless of the logging setting of the table and the archivelog and force logging settings of the database.

- Direct-path `INSERT` generates both redo and undo for *metadata* changes, because these are needed for operation recovery. For *data* changes, undo and redo are generated as follows:
  - Direct-path `INSERT` always bypasses undo generation for data changes.
  - If the database is not in `ARCHIVELOG` or `FORCE LOGGING` mode, then no redo is generated for data changes, regardless of the logging setting of the table.
  - If the database is in `ARCHIVELOG` mode (but not in `FORCE LOGGING` mode), then direct-path `INSERT` generates data redo for `LOGGING` tables but not for `NOLOGGING` tables.
  - If the database is in `ARCHIVELOG` **and** `FORCE LOGGING` mode, then direct-path SQL generate data redo for both `LOGGING` and `NOLOGGING` tables.

Direct-path `INSERT` is subject to a number of restrictions. If any of these restrictions is violated, then Oracle Database executes conventional `INSERT` serially without returning any message, unless otherwise noted:

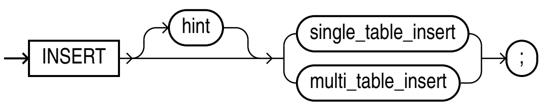
- You can have multiple direct-path `INSERT` statements in a single transaction, with or without other DML statements. However, after one DML statement alters a particular table, partition, or index, no other DML statement in the transaction can access that table, partition, or index.
- Queries that access the same table, partition, or index are allowed before the direct-path `INSERT` statement, but not after it.
- If any serial or parallel statement attempts to access a table that has already been modified by a direct-path `INSERT` in the same transaction, then the database returns an error and rejects the statement.
- The target table cannot be of a cluster.
- The target table cannot contain object type columns.
- Direct-path `INSERT` is not supported for an index-organized table (IOT) if it has a mapping table, or if it is reference by a materialized view.
- Direct-path `INSERT` into a single partition of an index-organized table (IOT), into a partitioned IOT with only one partition, or into an IOT that is not partitioned, will be done serially, even if the IOT was created in parallel mode or you specify the `APPEND` or `APPEND_VALUES` hint. However, direct-path `INSERT` operations into a partitioned IOT will honor parallel mode as long as the partition-extended name is not used and the IOT has more than one partition.
- The target table cannot have any triggers or referential integrity constraints defined on it.
- The target table cannot be replicated.
- A transaction containing a direct-path `INSERT` statement cannot be or become distributed.

You cannot query or modify direct-path inserted data immediately after the insert is complete. If you attempt to do so, an `ORA-12838` error is generated. You must first issue a `COMMIT` statement before attempting to read or modify the newly-inserted data.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for a more complete description of direct-path `INSERT`
> - *Oracle Database Utilities* for information on SQL*Loader
> - *Oracle Database SQL Tuning Guide* for information on statistics gathering when inserting into an empty table using direct-path `INSERT`

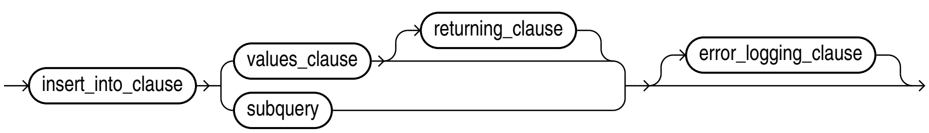
Syntax

***insert*::=**



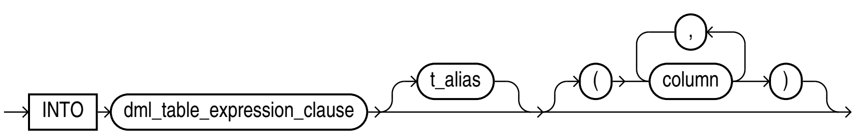
(*single_table_insert*::=, *multi_table_insert*::=)

***single_table_insert*::=**



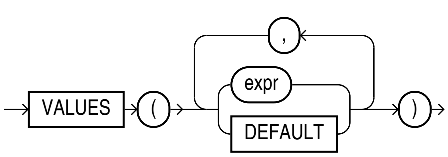
(*insert_into_clause*::=, *values_clause*::=, *returning_clause*::=, *subquery*::=, *error_logging_clause*::=)

***insert_into_clause*::=**

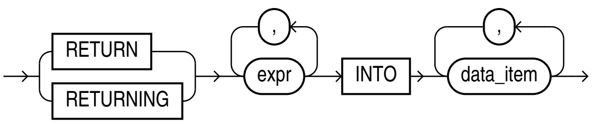


(*DML_table_expression_clause*::=)

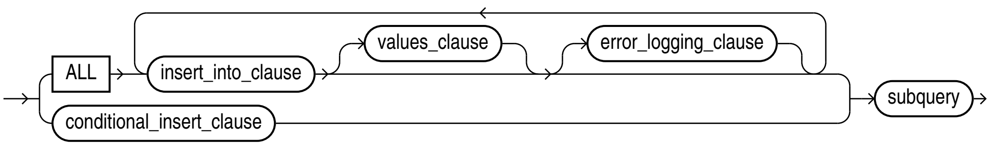
***values_clause*::=**



***returning_clause*::=**

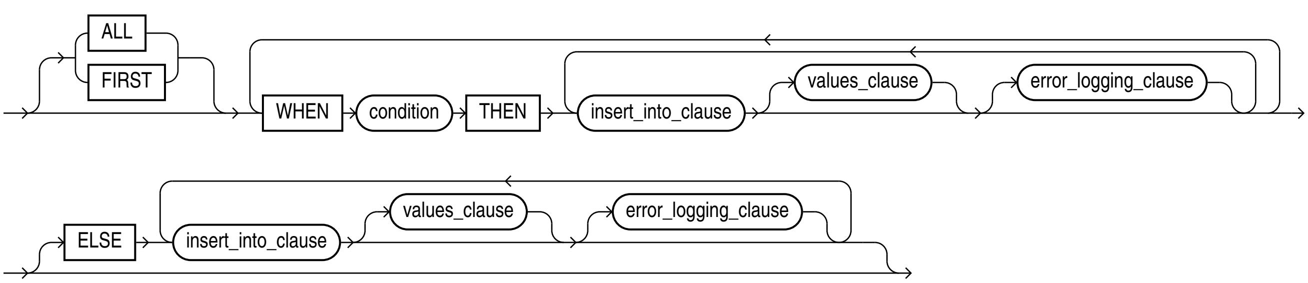


***multi_table_insert*::=**



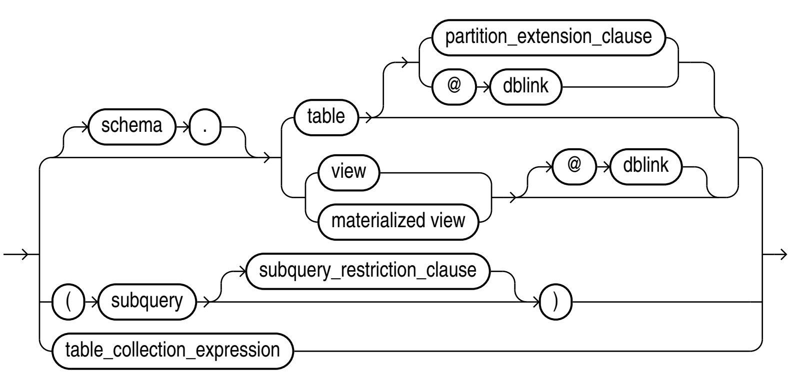
(*insert_into_clause::=*, *values_clause::=*, *conditional_insert_clause::=*, *subquery::=*, *error_logging_clause::=*)

***conditional_insert_clause*::=**



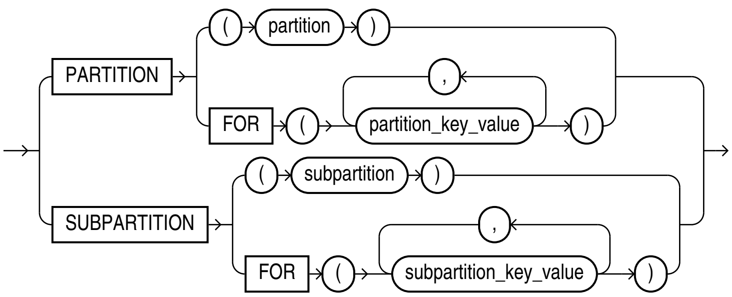
(*insert_into_clause::=*, *values_clause::=*)

***DML_table_expression_clause*::=**

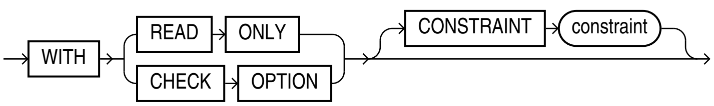


(*partition_extension_clause*::=, *subquery*::=—part of `SELECT`, *subquery_restriction_clause*::=, *table_collection_expression*::=)

***partition_extension_clause*::=**

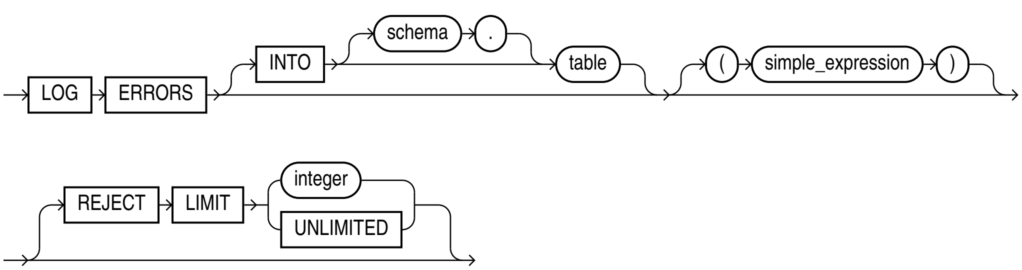


***subquery_restriction_clause*::=**



***table_collection_expression*::=**

***error_logging_clause*::=**



Semantics

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

For a multitable insert, if you specify the `PARALLEL` hint for any target table, then the entire multitable insert statement is parallelized even if the target tables have not been created or altered with `PARALLEL` specified. If you do not specify the `PARALLEL` hint, then the insert operation will not be parallelized unless all target tables were created or altered with `PARALLEL` specified.

> **See Also:**
>
> - "Hints " for the syntax and description of hints
> - "Restrictions on Multitable Inserts"

single_table_insert

In a **single-table insert**, you insert values into one row of a table, view, or materialized view by specifying values explicitly or by retrieving the values through a subquery.

You can use the *`flashback_query_clause`* in *`subquery`* to insert past data into *`table`*. Refer to the *flashback_query_clause* of `SELECT` for more information on this clause.

Restriction on Single-table Inserts

If you retrieve values through a subquery, then the select list of the subquery must have the same number of columns as the column list of the `INSERT` statement. If you omit the column list, then the subquery must provide values for every column in the table.

> **See Also:**
>
> "Inserting Values into Tables: Examples"

insert_into_clause

Use the `INSERT INTO` clause to specify the target object or objects into which the database is to insert data.

DML_table_expression_clause

Use the `INTO` *`DML_table_expression_clause`* to specify the objects into which data is being inserted.

schema

Specify the schema containing the table, view, or materialized view. If you omit *`schema`*, then the database assumes the object is in your own schema.

table* | *view* | *materialized_view* | *subquery

Specify the name of the table or object table, view or object view, materialized view, or the column or columns returned by a subquery, into which rows are to be inserted. If you specify a view or object view, then the database inserts rows into the base table of the view.

You cannot insert rows into a read-only materialized view. If you insert rows into a writable materialized view, then the database inserts the rows into the underlying container table. However, the insertions are overwritten at the next refresh operation. If you insert rows into an updatable materialized view that is part of a materialized view group, then the database also inserts the corresponding rows into the master table.

If any value to be inserted is a REF to an object table, and if the object table has a primary key object identifier, then the column into which you insert the REF must be a REF column with a referential integrity or SCOPE constraint to the object table.

If *table*, or the base table of *view*, contains one or more domain index columns, then this statement executes the appropriate indextype insert routine.

Issuing an INSERT statement against a table fires any INSERT triggers defined on the table.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on these routines

Restrictions on the *DML_table_expression_clause*

This clause is subject to the following restrictions:

- You cannot execute this statement if *table* or the base table of *view* contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- With regard to the ORDER BY clause of the *subquery* in the *DML_table_expression_clause*, ordering is guaranteed only for the rows being inserted, and only within each extent of the table. Ordering of new rows with respect to existing rows is not guaranteed.
- If a view was created using the WITH CHECK OPTION, then you can insert into the view only rows that satisfy the defining query of the view.
- If a view was created using a single base table, then you can insert rows into the view and then retrieve those values using the *returning_clause*.
- You cannot insert rows into a view except with INSTEAD OF triggers if the defining query of the view contains one of the following constructs:

  A set operator
  A DISTINCT operator
  An aggregate or analytic function
  A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
  A collection expression in a SELECT list
  A subquery in a SELECT list
  A subquery designated WITH READ ONLY
  Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the INSERT statement will fail unless the SKIP_UNUSABLE_INDEXES session parameter has been set to TRUE. Refer to ALTER SESSION for information on the SKIP_UNUSABLE_INDEXES session parameter.

partition_extension_clause

Specify the name or partition key value of the partition or subpartition within *`table`*, or the base table of *`view`*, targeted for inserts.

If a row to be inserted does not map into a specified partition or subpartition, then the database returns an error.

Restriction on Target Partitions and Subpartitions

This clause is not valid for object tables or object views.

> **See Also:**
>
> "References to Partitioned Tables and Indexes "

dblink

Specify a complete or partial name of a database link to a remote database where the table or view is located. You can insert rows into a remote table or view only if you are using Oracle Database distributed functionality.

If you omit *`dblink`*, then Oracle Database assumes that the table or view is on the local database.

> **Note:**
>
> Starting with Oracle Database 12*c* Release 2 (12.2), the `INSERT` statement accepts remote LOB locators as bind variables. Refer to the “Distributed LOBs” chapter in *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

> **See Also:**
>
> - "Syntax for Schema Objects and Parts in SQL Statements" and "References to Objects in Remote Databases " for information on referring to database links
> - "Inserting into a Remote Database: Example"

subquery_restriction_clause

Use the *`subquery_restriction_clause`* to restrict the subquery in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_Cn`, where n is an integer that makes the constraint name unique within the database.

> **See Also:**
>
> "Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *`table_collection_expression`* lets you inform Oracle that the value of *`collection_expression`* should be treated as a table for purposes of query and DML operations. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the `TABLE` collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

> **Note:**
>
> In earlier releases of Oracle, when *`collection_expression`* was a subquery, *`table_collection_expression`* was expressed as `THE` *`subquery`*. That usage is now deprecated.

> **See Also:**
>
> "Table Collections: Examples"

t_alias

Specify a **correlation name**, which is an alias for the table, view, materialized view, or subquery to be referenced elsewhere in the statement.

Restriction on Table Aliases

You cannot specify *`t_alias`* during a multitable insert.

column

Specify a column of the table, view, or materialized view. In the inserted row, each column in this list is assigned a value from the *values_clause* or the subquery. If you want to assign a value to an INVISIBLE column, then you must include the column in this list.

If you omit one or more of the table's columns from this list, then the column value of that column for the inserted row is the column default value as specified when the table was created or last altered. If any omitted column has a NOT NULL constraint and no default value, then the database returns an error indicating that the constraint has been violated and rolls back the INSERT statement. Refer to CREATE TABLE for more information on default column values.

If you omit the column list altogether, then the *values_clause* or query must specify values for all columns in the table.

values_clause

For a **single-table insert** operation, specify a row of values to be inserted into the table or view. You must specify a value in the *values_clause* for each column in the column list. If you omit the column list, then the *values_clause* must provide values for every column in the table.

For a **multitable insert** operation, each expression in the *values_clause* must refer to columns returned by the select list of the subquery. If you omit the *values_clause*, then the select list of the subquery determines the values to be inserted, so it must have the same number of columns as the column list of the corresponding *insert_into_clause*. If you do not specify a column list in the *insert_into_clause*, then the computed row must provide values for all columns in the target table.

For both types of insert operations, if you specify a column list in the *insert_into_clause*, then the database assigns to each column in the list a corresponding value from the values clause or the subquery. You can specify DEFAULT for any value in the *values_clause*. If you have specified a default value for the corresponding column of the table or view, then that value is inserted. If no default value for the corresponding column has been specified, then the database inserts null. Refer to "About SQL Expressions " and SELECT for syntax of valid expressions.

Restrictions on Inserted Values

The value are subject to the following restrictions:

- You cannot insert a BFILE value until you have initialized the BFILE locator to null or to a directory name and filename.

  See Also:

  – BFILENAME for information on initializing BFILE values and for an example of inserting into a BFILE

  – *Oracle Database SecureFiles and Large Objects Developer's Guide* for information on initializing BFILE locators

- When inserting into a list-partitioned table, you cannot insert a value into the partitioning key column that does not already exist in the *partition_key_value* list of one of the partitions.

- You cannot specify DEFAULT when inserting into a view.
- If you insert string literals into a RAW column, then during subsequent queries Oracle Database will perform a full table scan rather than using any index that might exist on the RAW column.

> **See Also:**
>
> – "Using XML in SQL Statements " for information on inserting values into an XMLType table
>
> – "Inserting into a Substitutable Tables and Columns: Examples", "Inserting Using the TO_LOB Function: Example", "Inserting Sequence Values: Example", and "Inserting Using Bind Variables: Example"

returning_clause

The returning clause retrieves the rows affected by a DML statement. You can specify this clause for tables and materialized views and for views with a single base table.

When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and REFs to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the affected rows in bind arrays.

expr

Each item in the *expr* list must be a valid expression syntax.

INTO

The INTO clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.

data_item

Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the RETURNING list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the INTO list.

Restrictions

The following restrictions apply to the RETURNING clause:

- The *expr* is restricted as follows:
  – For UPDATE and DELETE statements each *expr* must be a simple expression or a single-set aggregate function expression. You cannot combine simple expressions and single-set aggregate function expressions in the same *returning_clause*. For INSERT statements, each *expr* must be a simple expression. Aggregate functions are not supported in an INSERT statement RETURNING clause.
  – Single-set aggregate function expressions cannot include the DISTINCT keyword.

- If the *expr* list contains a primary key column or other NOT NULL column, then the update statement fails if the table has a BEFORE UPDATE trigger defined on it.
- You cannot specify the *returning_clause* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

multi_table_insert

In a **multitable insert**, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Table aliases are not defined by the select list of the subquery. Therefore, they are not visible in the clauses dependent on the select list. For example, this can happen when trying to refer to an object column in an expression. To use an expression with a table alias, you must put the expression into the select list with a column alias, and then refer to the column alias in the VALUES clause or WHEN condition of the multitable insert.

ALL *into_clause*

Specify ALL followed by multiple *insert_into_clauses* to perform an **unconditional multitable insert**. Oracle Database executes each *insert_into_clause* once for each row returned by the subquery.

conditional_insert_clause

Specify the *conditional_insert_clause* to perform a **conditional multitable insert**. Oracle Database filters each *insert_into_clause* through the corresponding WHEN condition, which determines whether that *insert_into_clause* is executed. Each expression in the WHEN condition must refer to columns returned by the select list of the subquery. A single multitable insert statement can contain up to 127 WHEN clauses.

ALL

If you specify ALL, the default value, then the database evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the database executes the corresponding INTO clause list.

FIRST

If you specify FIRST, then the database evaluates each WHEN clause in the order in which it appears in the statement. For the first WHEN clause that evaluates to true, the database executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

ELSE clause

For a given row, if no `WHEN` clause evaluates to true, then:

- If you have specified an `ELSE` clause, then the database executes the `INTO` clause list associated with the `ELSE` clause.
- If you did not specify an else clause, then the database takes no action for that row.

> **See Also:**
>
> "Multitable Inserts: Examples"

Restrictions on Multitable Inserts

Multitable inserts are subject to the following restrictions:

- You can perform multitable inserts only on tables, not on views or materialized views.
- You cannot perform a multitable insert into a remote table.
- You cannot specify a `TABLE` collection expression when performing a multitable insert.
- Multitable inserts are not parallelized if any target table is index organized or if any target table has a bitmap index defined on it.
- Plan stability is not supported for multitable insert statements.
- You cannot specify a sequence in any part of a multitable insert statement. A multitable insert is considered a single SQL statement. Therefore, the first reference to `NEXTVAL` generates the next number, and all subsequent references in the statement return the same number.

subquery

Specify a subquery that returns rows that are inserted into the table. The subquery can refer to any table, view, or materialized view, including the target tables of the `INSERT` statement. If the subquery selects no rows, then the database inserts no rows into the table.

You can use *`subquery`* in combination with the `TO_LOB` function to convert the values in a `LONG` column to LOB values in another column in the same or another table.

- To migrate `LONG` values to LOB values in another column in a view, you must perform the migration on the base table and then add the LOB column to the view.
- To migrate `LONG` values on a remote table to LOB values in a local table, you must perform the migration on the remote table using the `TO_LOB` function, and then perform an `INSERT` ... *`subquery`* operation to copy the LOB values from the remote table into the local table.

Notes on Inserting with a Subquery

The following notes apply when inserting with a subquery:

- If *`subquery`* returns the partial or total equivalent of a materialized view, then the database may use the materialized view for query rewrite in place of one or more tables specified in *`subquery`*.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on materialized views and query rewrite

- If `subquery` refers to remote objects, then the `INSERT` operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the `subquery` in the `DML_table_expression_clause` refers to any remote objects, then the `INSERT` operation will run serially without notification. See *parallel_clause* for more information.
- If `subquery` includes an `ORDER BY` clause, then it will override row ordering specified using attribute clustering table properties.

> **See Also:**
>
> - "Inserting Values with a Subquery: Example"
> - BFILENAME for an example of inserting into a `BFILE`
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for information on initializing `BFILE`s
> - "About SQL Expressions " and SELECT for syntax of valid expressions

error_logging_clause

The `error_logging_clause` lets you capture DML errors and the log column values of the affected rows and save them in an error logging table.

INTO *table*

Specify the name of the error logging table. If you omit this clause, then the database assigns the default name generated by the `DBMS_ERRLOG` package. The default error log table name is `ERR$_` followed by the first 25 characters of the name of the table upon which the DML operation is being executed.

simple_expression

Specify the value to be used as a statement tag, so that you can identify the errors from this statement in the error logging table. The expression can be either a text literal, a number literal, or a general SQL expression such as a bind variable. You can also use a function expression if you convert it to a text literal — for example, `TO_CHAR(SYSDATE)`.

REJECT LIMIT

This clause lets you specify an integer as an upper limit for the number of errors to be logged before the statement terminates and rolls back any changes made by the statement. The default rejection limit is zero. For parallel DML operations, the reject limit is applied to each parallel server.

Restrictions on DML Error Logging

- The following conditions cause the statement to fail and roll back without invoking the error logging capability:
  - Violated deferred constraints.
  - Any direct-path `INSERT` or `MERGE` operation that raises a unique constraint or index violation.
  - Any update operation `UPDATE` or `MERGE` that raises a unique constraint or index violation.
- You cannot track errors in the error logging table for `LONG`, LOB, or object type columns. However, the table that is the target of the DML operation can contain these types of columns.
  - If you create or modify the corresponding error logging table so that it contains a column of an unsupported type, and if the name of that column corresponds to an unsupported column in the target DML table, then the DML statement fails at parse time.
  - If the error logging table does not contain any unsupported column types, then all DML errors are logged until the reject limit of errors is reached. For rows on which errors occur, column values with corresponding columns in the error logging table are logged along with the control information.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for information on using the `create_error_log` procedure of the `DBMS_ERRLOG` package and *Oracle Database Administrator's Guide* for general information on DML error logging.
> - "Inserting Into a Table with Error Logging: Example"

Examples

Inserting Values into Tables: Examples

The following statement inserts a row into the sample table `departments`:

```
INSERT INTO departments
   VALUES (280, 'Recreation', 121, 1700);
```

If the `departments` table had been created with a default value of 121 for the `manager_id` column, then you could issue the same statement as follows:

```
INSERT INTO departments
   VALUES (280, 'Recreation', DEFAULT, 1700);
```

The following statement inserts a row with six columns into the `employees` table. One of these columns is assigned `NULL` and another is assigned a number in scientific notation:

```
INSERT INTO employees (employee_id, last_name, email,
      hire_date, job_id, salary, commission_pct)
   VALUES (207, 'Gregory', 'pgregory@example.com',
      sysdate, 'PU_CLERK', 1.2E3, NULL);
```

The following statement has the same effect as the preceding example, but uses a subquery in the *`DML_table_expression_clause`*:

```
INSERT INTO
   (SELECT employee_id, last_name, email, hire_date, job_id,
      salary, commission_pct FROM employees)
   VALUES (207, 'Gregory', 'pgregory@example.com',
      sysdate, 'PU_CLERK', 1.2E3, NULL);
```

Inserting Values with a Subquery: Example

The following statement copies employees whose commission exceeds 25% of their salary into the `bonuses` table, which was created in "Merging into a Table: Example":

```
INSERT INTO bonuses
   SELECT employee_id, salary*1.1
   FROM employees
   WHERE commission_pct > 0.25;
```

Inserting Into a Table with Error Logging: Example

The following statements create a `raises` table in the sample schema `hr`, create an error logging table using the `DBMS_ERRLOG` package, and populate the `raises` table with data from the `employees` table. One of the inserts violates the check constraint on `raises`, and that row can be seen in `errlog`. If more than ten errors had occurred, then the statement would have aborted, rolling back any insertions made:

```
CREATE TABLE raises (emp_id NUMBER, sal NUMBER
   CONSTRAINT check_sal CHECK(sal > 8000));

EXECUTE DBMS_ERRLOG.CREATE_ERROR_LOG('raises', 'errlog');

INSERT INTO raises
   SELECT employee_id, salary*1.1 FROM employees
   WHERE commission_pct > .2
   LOG ERRORS INTO errlog ('my_bad') REJECT LIMIT 10;

SELECT ORA_ERR_MESG$, ORA_ERR_TAG$, emp_id, sal FROM errlog;

ORA_ERR_MESG$               ORA_ERR_TAG$         EMP_ID SAL
--------------------------- -------------------- ------ -------
ORA-02290: check constraint my_bad               161    7700
 (HR.SYS_C004266) violated
```

Inserting into a Remote Database: Example

The following statement inserts a row into the `employees` table owned by the user `hr` on the database accessible by the database link `remote`:

```
INSERT INTO employees@remote
   VALUES (8002, 'Juan', 'Fernandez', 'juanf@example.com', NULL,
   TO_DATE('04-OCT-1992', 'DD-MON-YYYY'), 'SH_CLERK', 3000,
   NULL, 121, 20);
```

Inserting Sequence Values: Example

The following statement inserts a new row containing the next value of the `departments_seq` sequence into the `departments` table:

```
INSERT INTO departments
   VALUES  (departments_seq.nextval, 'Entertainment', 162, 1400);
```

Inserting Using Bind Variables: Example

The following example returns the values of the inserted rows into output bind variables :bnd1 and :bnd2. The bind variables must first be declared.

```
INSERT INTO employees
      (employee_id, last_name, email, hire_date, job_id, salary)
   VALUES
   (employees_seq.nextval, 'Doe', 'john.doe@example.com',
       SYSDATE, 'SH_CLERK', 2400)
   RETURNING salary*12, job_id INTO :bnd1, :bnd2;
```

Inserting into a Substitutable Tables and Columns: Examples

The following example inserts into the persons table, which is created in "Substitutable Table and Column Examples". The first statement uses the root type person_t. The second insert uses the employee_t subtype of person_t, and the third insert uses the part_time_emp_t subtype of employee_t:

```
INSERT INTO persons VALUES (person_t('Bob', 1234));
INSERT INTO persons VALUES (employee_t('Joe', 32456, 12, 100000));
INSERT INTO persons VALUES (
   part_time_emp_t('Tim', 5678, 13, 1000, 20));
```

The following example inserts into the books table, which was created in "Substitutable Table and Column Examples". Notice that specification of the attribute values is identical to that for the substitutable table example:

```
INSERT INTO books VALUES (
   'An Autobiography', person_t('Bob', 1234));
INSERT INTO books VALUES (
   'Business Rules', employee_t('Joe', 3456, 12, 10000));
INSERT INTO books VALUES (
   'Mixing School and Work',
   part_time_emp_t('Tim', 5678, 13, 1000, 20));
```

You can extract data from substitutable tables and columns using built-in functions and conditions. For examples, see the functions TREAT and SYS_TYPEID , and "IS OF *type* Condition ".

Inserting Using the TO_LOB Function: Example

The following example copies LONG data to a LOB column in the following long_tab table:

```
CREATE TABLE long_tab (pic_id NUMBER, long_pics LONG RAW);
```

First you must create a table with a LOB.

```
CREATE TABLE lob_tab (pic_id NUMBER, lob_pics BLOB);
```

Next, use an INSERT ... SELECT statement to copy the data in all rows for the LONG column into the newly created LOB column:

```
INSERT INTO lob_tab
   SELECT pic_id, TO_LOB(long_pics) FROM long_tab;
```

When you are confident that the migration has been successful, you can drop the long_pics table. Alternatively, if the table contains other columns, then you can simply drop the LONG column from the table as follows:

```
ALTER TABLE long_tab DROP COLUMN long_pics;
```

Multitable Inserts: Examples

The following example uses the multitable insert syntax to insert into the sample table `sh.sales` some data from an input table with a different structure.

> **Note:**
>
> A number of `NOT NULL` constraints on the `sales` table have been disabled for purposes of this example, because the example ignores a number of table columns for the sake of brevity.

The input table looks like this:

```
SELECT * FROM sales_input_table;

PRODUCT_ID CUSTOMER_ID WEEKLY_ST  SALES_SUN  SALES_MON  SALES_TUE  SALES_WED SALES_THU  SALES_FRI  SALES_SAT
---------- ----------- --------- ---------- ---------- ---------- -------------------- ---------- ----------
       111         222 01-OCT-00        100        200        300        400       500        600        700
       222         333 08-OCT-00        200        300        400        500       600        700        800
       333         444 15-OCT-00        300        400        500        600       700        800        900
```

The multitable insert statement looks like this:

```
INSERT ALL
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date, sales_sun)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+1, sales_mon)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+2, sales_tue)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+3, sales_wed)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+4, sales_thu)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+5, sales_fri)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+6, sales_sat)
   SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
      FROM sales_input_table;
```

Assuming these are the only rows in the `sales` table, the contents now look like this:

```
SELECT * FROM sales
   ORDER BY prod_id, cust_id, time_id;

   PROD_ID    CUST_ID TIME_ID   C   PROMO_ID QUANTITY_SOLD     AMOUNT       COST
---------- ---------- --------- - ---------- ------------- ---------- ----------
       111        222 01-OCT-00                                   100
       111        222 02-OCT-00                                   200
       111        222 03-OCT-00                                   300
       111        222 04-OCT-00                                   400
       111        222 05-OCT-00                                   500
       111        222 06-OCT-00                                   600
       111        222 07-OCT-00                                   700
       222        333 08-OCT-00                                   200
       222        333 09-OCT-00                                   300
       222        333 10-OCT-00                                   400
```

```
222        333 11-OCT-00                                500
222        333 12-OCT-00                                600
222        333 13-OCT-00                                700
222        333 14-OCT-00                                800
333        444 15-OCT-00                                300
333        444 16-OCT-00                                400
333        444 17-OCT-00                                500
333        444 18-OCT-00                                600
333        444 19-OCT-00                                700
333        444 20-OCT-00                                800
333        444 21-OCT-00                                900
```

The next examples insert into multiple tables. Suppose you want to provide to sales representatives some information on orders of various sizes. The following example creates tables for small, medium, large, and special orders and populates those tables with data from the sample table `oe.orders`:

```
CREATE TABLE small_orders
   (order_id       NUMBER(12)   NOT NULL,
    customer_id    NUMBER(6)    NOT NULL,
    order_total    NUMBER(8,2),
    sales_rep_id   NUMBER(6)
   );

CREATE TABLE medium_orders AS SELECT * FROM small_orders;

CREATE TABLE large_orders AS SELECT * FROM small_orders;

CREATE TABLE special_orders
   (order_id       NUMBER(12)    NOT NULL,
    customer_id    NUMBER(6)     NOT NULL,
    order_total    NUMBER(8,2),
    sales_rep_id   NUMBER(6),
    credit_limit   NUMBER(9,2),
    cust_email     VARCHAR2(40)
   );
```

The first multitable insert populates only the tables for small, medium, and large orders:

```
INSERT ALL
   WHEN order_total <= 100000 THEN
      INTO small_orders
   WHEN order_total > 1000000 AND order_total <= 200000 THEN
      INTO medium_orders
   WHEN order_total > 200000 THEN
      INTO large_orders
   SELECT order_id, order_total, sales_rep_id, customer_id
      FROM orders;
```

You can accomplish the same thing using the `ELSE` clause in place of the insert into the `large_orders` table:

```
INSERT ALL
   WHEN order_total <= 100000 THEN
      INTO small_orders
   WHEN order_total > 100000 AND order_total <= 200000 THEN
      INTO medium_orders
   ELSE
      INTO large_orders
   SELECT order_id, order_total, sales_rep_id, customer_id
      FROM orders;
```

The next example inserts into the small, medium, and large tables, as in the preceding example, and also puts orders greater than 290,000 into the `special_orders` table. This table also shows how to use column aliases to simplify the statement:

```
INSERT ALL
   WHEN ottl <= 100000 THEN
      INTO small_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 100000 and ottl <= 200000 THEN
      INTO medium_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 200000 THEN
      into large_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 290000 THEN
      INTO special_orders
   SELECT o.order_id oid, o.customer_id cid, o.order_total ottl,
      o.sales_rep_id sid, c.credit_limit cl, c.cust_email cem
      FROM orders o, customers c
      WHERE o.customer_id = c.customer_id;
```

Finally, the next example uses the `FIRST` clause to put orders greater than 290,000 into the `special_orders` table and exclude those orders from the `large_orders` table:

```
INSERT FIRST
   WHEN ottl <= 100000 THEN
      INTO small_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 100000 and ottl <= 200000 THEN
      INTO medium_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 290000 THEN
      INTO special_orders
   WHEN ottl > 200000 THEN
      INTO large_orders
         VALUES(oid, ottl, sid, cid)
   SELECT o.order_id oid, o.customer_id cid, o.order_total ottl,
      o.sales_rep_id sid, c.credit_limit cl, c.cust_email cem
      FROM orders o, customers c
      WHERE o.customer_id = c.customer_id;
```

LOCK TABLE

Purpose

Use the `LOCK TABLE` statement to lock one or more tables, table partitions, or table subpartitions in a specified mode. This lock manually overrides automatic locking and permits or denies access to a table or view by other users for the duration of your operation.

Some forms of locks can be placed on the same table at the same time. Other locks allow only one lock for a table.

A locked table remains locked until you either commit your transaction or roll it back, either entirely or to a savepoint before you locked the table.

A lock never prevents other users from querying the table. A query never places a lock on a table. Readers never block writers and writers never block readers.

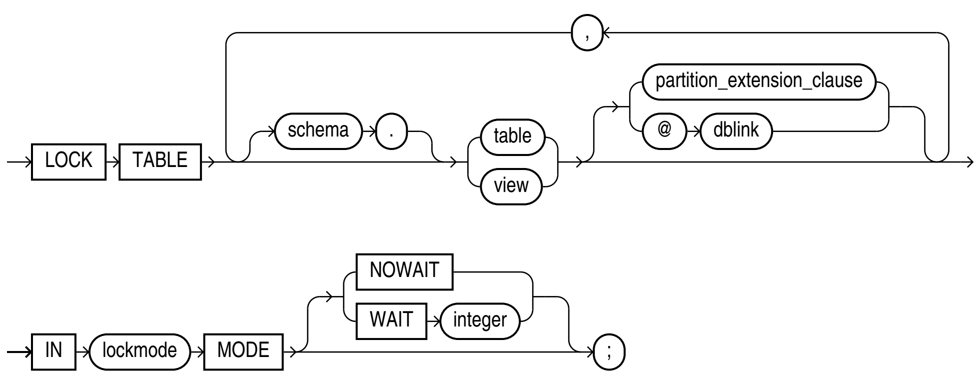
> **See Also:**
>
> - *Oracle Database Concepts* for a complete description of the interaction of lock modes
> - COMMIT
> - ROLLBACK
> - SAVEPOINT

Prerequisites

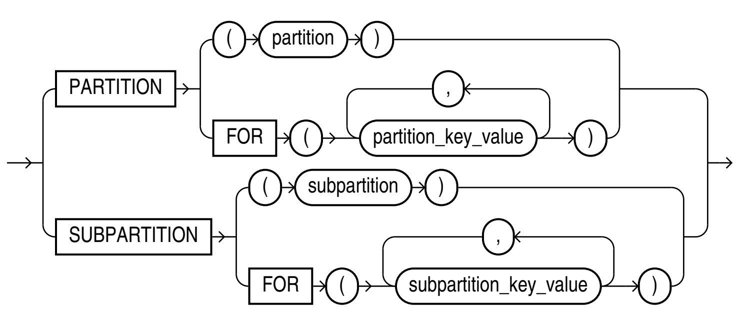
The table or view must be in your own schema, or you must have the `LOCK ANY TABLE` system privilege, or you must have any object privilege (except the `READ` object privilege) on the table or view.

Syntax

***lock_table*::=**



***partition_extension_clause*::=**



Semantics

schema

Specify the schema containing the table or view. If you omit *schema*, then Oracle Database assumes the table or view is in your own schema.

table | *view*

Specify the name of the table or view to be locked.

If you specify *view*, then Oracle Database locks the base tables of the view.

If you specify the *partition_extension_clause*, then Oracle Database first acquires an implicit lock on the table. The table lock is the same as the lock you specify for the partition or subpartition, with two exceptions:

- If you specify a `SHARE` lock for the subpartition, then the database acquires an implicit `ROW SHARE` lock on the table.
- If you specify an `EXCLUSIVE` lock for the subpartition, then the database acquires an implicit `ROW EXCLUSIVE` lock on the table.

If you specify `PARTITION` and *table* is composite-partitioned, then the database acquires locks on all the subpartitions of the partition.

Restrictions on Locking Tables

The following restrictions apply to locking tables:

- If *view* is part of a hierarchy, then it must be the root of the hierarchy.
- You can acquire locks on only the existing partitions in an automatic list-partitioned table. That is, when you specify the following statement, the partition key value must correspond to a partition that already exists in the table; it cannot correspond to a partition that might be created on-demand at a later time:

```
LOCK TABLE ... PARTITION FOR (partition_key_value) ...
```

dblink

Specify a database link to a remote Oracle Database where the table or view is located. You can lock tables and views on a remote database only if you are using Oracle distributed functionality. All tables locked by a `LOCK TABLE` statement must be on the same database.

If you omit *dblink*, then Oracle Database assumes the table or view is on the local database.

> **See Also:**
>
> "References to Objects in Remote Databases " for information on specifying database links

lockmode Clause

Specify one of the following modes:

ROW SHARE

`ROW SHARE` permits concurrent access to the locked table but prohibits users from locking the entire table for exclusive access. `ROW SHARE` is synonymous with `SHARE UPDATE`, which is included for compatibility with earlier versions of Oracle Database.

ROW EXCLUSIVE

`ROW EXCLUSIVE` is the same as `ROW SHARE`, but it also prohibits locking in `SHARE` mode. `ROW EXCLUSIVE` locks are automatically obtained when updating, inserting, or deleting.

SHARE UPDATE

See ROW SHARE.

SHARE

`SHARE` permits concurrent queries but prohibits updates to the locked table.

SHARE ROW EXCLUSIVE

`SHARE ROW EXCLUSIVE` is used to look at a whole table and to allow others to look at rows in the table but to prohibit others from locking the table in `SHARE` mode or from updating rows.

EXCLUSIVE

`EXCLUSIVE` permits queries on the locked table but prohibits any other activity on it.

NOWAIT

Specify `NOWAIT` if you want the database to return control to you immediately if the specified table, partition, or table subpartition is already locked by another user. In this case, the database returns a message indicating that the table, partition, or subpartition is already locked by another user.

WAIT

Use the `WAIT` clause to indicate that the `LOCK TABLE` statement should wait up to the specified number of seconds to acquire a DML lock. There is no limit on the value of *`integer`*.

If you specify neither `NOWAIT` nor `WAIT`, then the database waits indefinitely until the table is available, locks it, and returns control to you. When the database is executing DDL statements concurrently with DML statements, a timeout or deadlock can sometimes result. The database detects such timeouts and deadlocks and returns an error.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about locking tables

Examples

Locking a Table: Example

The following statement locks the `employees` table in exclusive mode but does not wait if another user already has locked the table:

```
LOCK TABLE employees
   IN EXCLUSIVE MODE
   NOWAIT;
```

The following statement locks the remote `employees` table that is accessible through the database link `remote`:

```
LOCK TABLE employees@remote
   IN SHARE MODE;
```

19
SQL Statements: MERGE to UPDATE

This chapter contains the following SQL statements:

- MERGE
- NOAUDIT (Traditional Auditing)
- NOAUDIT (Unified Auditing)
- PURGE
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SELECT
- SET CONSTRAINT[S]
- SET ROLE
- SET TRANSACTION
- TRUNCATE CLUSTER
- TRUNCATE TABLE
- UPDATE

MERGE

Purpose

Use the `MERGE` statement to select rows from one or more sources for update or insertion into a table or view. You can specify conditions to determine whether to update or insert into the target table or view.

This statement is a convenient way to combine multiple operations. It lets you avoid multiple `INSERT`, `UPDATE`, and `DELETE` DML statements.

`MERGE` is a deterministic statement. You cannot update the same row of the target table multiple times in the same `MERGE` statement.

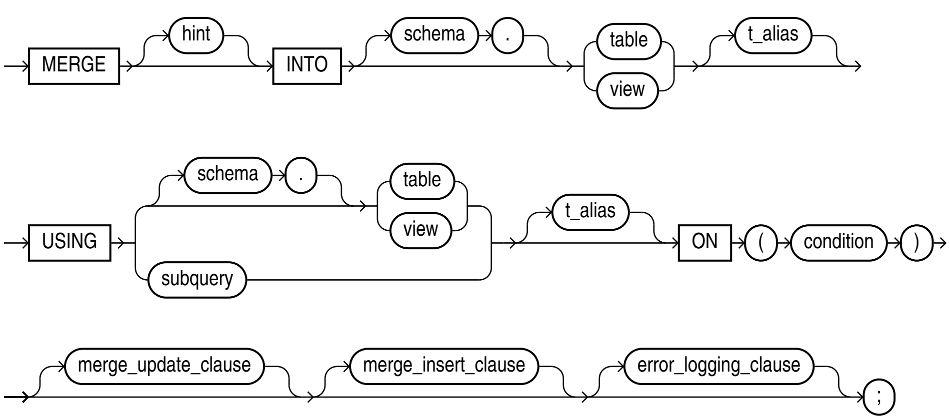
> **Note:**
>
> In previous releases of Oracle Database, when you created an Oracle Virtual Private Database policy on an application that included the `MERGE INTO` statement, the `MERGE INTO` statement would be prevented with an `ORA-28132: Merge into syntax does not support security policies` error, due to the presence of the Virtual Private Database policy. Beginning with Oracle Database 11*g* Release 2 (11.2.0.2), you can create policies on applications that include `MERGE INTO` operations. To do so, in the `DBMS_RLS.ADD_POLICY` `statement_types` parameter, include the `INSERT`, `UPDATE`, and `DELETE` statements, or just omit the `statement_types` parameter altogether. Refer to *Oracle Database Security Guide* for more information on enforcing policies on specific SQL statement types.

Prerequisites

You must have the `INSERT` and `UPDATE` object privileges on the target table and the `SELECT` object privilege on the source objects. To specify the `DELETE` clause of the *`merge_update_clause`*, you must also have the `DELETE` object privilege on the target table or view.

Syntax

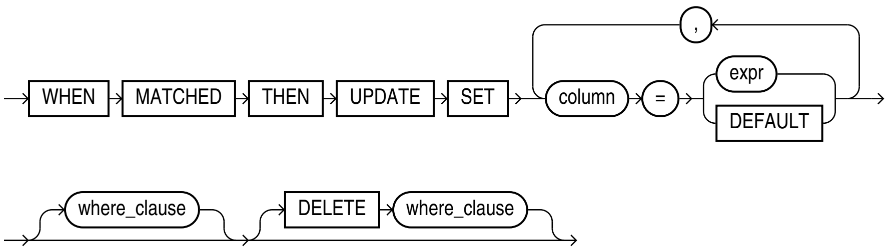
***merge*::=**



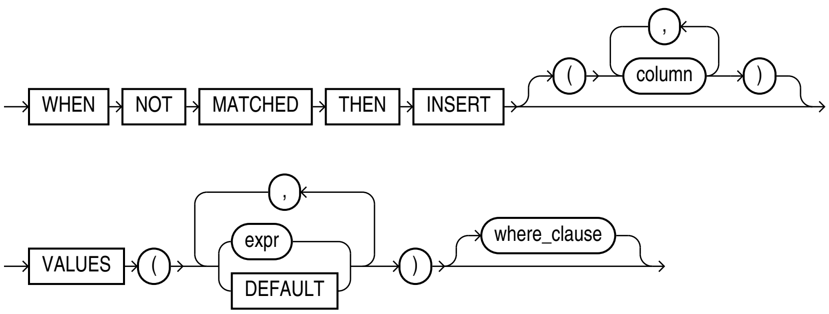
> **Note:**
>
> You must specify at least one of the clauses *`merge_update_clause`* or *`merge_insert_clause`*.

(*merge_update_clause::=*, *merge_insert_clause::=*, *error_logging_clause::=*

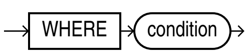
***merge_update_clause*::=**



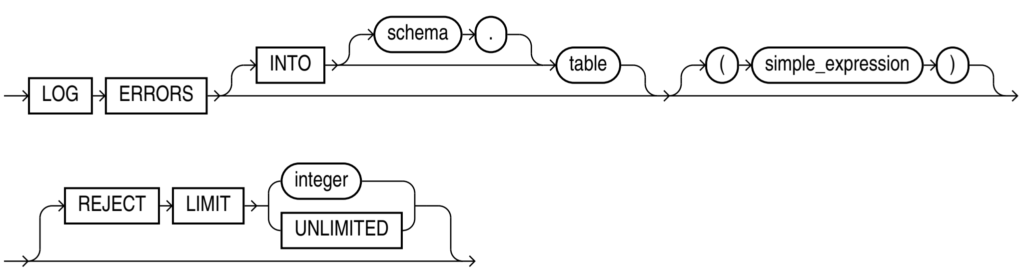
***merge_insert_clause*::=**



***where_clause*::=**



***error_logging_clause*::=**



Semantics

INTO Clause

Use the `INTO` clause to specify the target table or view you are updating or inserting into. In order to merge data into a view, the view must be updatable. Refer to "Notes on Updatable Views" for more information.

Restriction on Target Views

You cannot specify a target view on which an `INSTEAD OF` trigger has been defined.

USING Clause

Use the `USING` clause to specify the source you are updating or inserting from.

ON Clause

Use the `ON` clause to specify the condition upon which the `MERGE` operation either updates or inserts. For each row in the target table for which the search condition is true, Oracle Database updates the row with corresponding data from the source . If the condition is not true for any rows, then the database inserts into the target table based on the corresponding source row.

merge_update_clause

The *`merge_update_clause`* specifies the new column values of the target table or view. Oracle performs this update if the condition of the `ON` clause is true. If the update clause is executed, then all update triggers defined on the target table are activated.

Specify the *`where_clause`* if you want the database to execute the update operation only if the specified condition is true. The condition can refer to either the data source or the target table. If the condition is not true, then the database skips the update operation when merging the row into the table.

Specify the `DELETE` *`where_clause`* to clean up data in a table while populating or updating it. The only rows affected by this clause are those rows in the destination table that are updated by the merge operation. The `DELETE WHERE` condition evaluates the updated value, not the original value that was evaluated by the `UPDATE SET` ... `WHERE` condition. If a row of the destination table meets the `DELETE` condition but is not included in the join defined by the `ON` clause, then it is not deleted. Any delete triggers defined on the target table will be activated for each row deletion.

You can specify this clause by itself or with the *`merge_insert_clause`*. If you specify both, then they can be in either order.

Restrictions on the *merge_update_clause*

This clause is subject to the following restrictions:

- You cannot update a column that is referenced in the `ON` *`condition`* clause.
- You cannot specify `DEFAULT` when updating a view.

merge_insert_clause

The *`merge_insert_clause`* specifies values to insert into the column of the target table if the condition of the `ON` clause is false. If the insert clause is executed, then all insert triggers defined on the target table are activated. If you omit the column list after the `INSERT` keyword, then the number of columns in the target table must match the number of values in the `VALUES` clause.

To insert all of the source rows into the table, you can use a **constant filter predicate** in the `ON` clause condition. An example of a constant filter predicate is `ON` (`0=1`). Oracle Database recognizes such a predicate and makes an unconditional insert of all source rows into the table. This approach is different from omitting the *`merge_update_clause`*. In that case, the database still must perform a join. With constant filter predicate, no join is performed.

Specify the *where_clause* if you want Oracle Database to execute the insert operation only if the specified condition is true. The condition can refer only to the data source columns. Oracle Database skips the insert operation for all rows for which the condition is not true.

You can specify the *merge_insert_clause* by itself or with the *merge_update_clause*. If you specify both, then they can be in either order.

Restriction on the *merge_insert_clause*

You cannot specify DEFAULT when inserting into a view.

error_logging_clause

The *error_logging_clause* has the same behavior in a MERGE statement as in an INSERT statement. Refer to the INSERT statement *error_logging_clause* for more information.

> **See Also:**
>
> "Inserting Into a Table with Error Logging: Example"

Examples

Merging into a Table: Example

The following example uses the bonuses table in the sample schema oe with a default bonus of 100. It then inserts into the bonuses table all employees who made sales, based on the sales_rep_id column of the oe.orders table. Finally, the human resources manager decides that employees with a salary of $8000 or less should receive a bonus. Those who have not made sales get a bonus of 1% of their salary. Those who already made sales get an increase in their bonus equal to 1% of their salary. The MERGE statement implements these changes in one step:

```
CREATE TABLE bonuses (employee_id NUMBER, bonus NUMBER DEFAULT 100);

INSERT INTO bonuses(employee_id)
   (SELECT e.employee_id FROM hr.employees e, oe.orders o
   WHERE e.employee_id = o.sales_rep_id
   GROUP BY e.employee_id);

SELECT * FROM bonuses ORDER BY employee_id;

EMPLOYEE_ID      BONUS
----------- ----------
        153        100
        154        100
        155        100
        156        100
        158        100
        159        100
        160        100
        161        100
        163        100

MERGE INTO bonuses D
   USING (SELECT employee_id, salary, department_id FROM hr.employees
   WHERE department_id = 80) S
   ON (D.employee_id = S.employee_id)
```

```
   WHEN MATCHED THEN UPDATE SET D.bonus = D.bonus + S.salary*.01
     DELETE WHERE (S.salary > 8000)
   WHEN NOT MATCHED THEN INSERT (D.employee_id, D.bonus)
     VALUES (S.employee_id, S.salary*.01)
     WHERE (S.salary <= 8000);

SELECT * FROM bonuses ORDER BY employee_id;

EMPLOYEE_ID      BONUS
----------- ----------
        153        180
        154        175
        155        170
        159        180
        160        175
        161        170
        164         72
        165         68
        166         64
        167         62
        171         74
        172         73
        173         61
        179         62
```

Conditional Insert and Update: Example

The following example conditionally inserts and updates table data by using the `MERGE` statement.

The following statements create two tables named `people_source` and `people_target` and populate them with names:

```
CREATE TABLE people_source (
  person_id  INTEGER NOT NULL PRIMARY KEY,
  first_name VARCHAR2(20) NOT NULL,
  last_name  VARCHAR2(20) NOT NULL,
  title      VARCHAR2(10) NOT NULL
);

CREATE TABLE people_target (
  person_id  INTEGER NOT NULL PRIMARY KEY,
  first_name VARCHAR2(20) NOT NULL,
  last_name  VARCHAR2(20) NOT NULL,
  title      VARCHAR2(10) NOT NULL
);

INSERT INTO people_target VALUES (1, 'John', 'Smith', 'Mr');
INSERT INTO people_target VALUES (2, 'alice', 'jones', 'Mrs');
INSERT INTO people_source VALUES (2, 'Alice', 'Jones', 'Mrs.');
INSERT INTO people_source VALUES (3, 'Jane', 'Doe', 'Miss');
INSERT INTO people_source VALUES (4, 'Dave', 'Brown', 'Mr');

COMMIT;
```

The following statement compares the contents of people_target and people_source by using the person_id column. The values in the people_target table are updated when there is a match in the people_source table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
  SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title;
```

The following statements display the contents of the people_target table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID FIRST_NAME        LAST_NAME          TITLE
---------- -------------------- -------------------- ----------
     1 John         Smith              Mr
     2 Alice        Jones              Mrs.

ROLLBACK;
```

This statement compares the contents of the people_target and people_source tables by using the person_id column. The values in the people_target table are updated only when there is no match in the people_source table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN NOT MATCHED THEN INSERT
  (pt.person_id, pt.first_name, pt.last_name, pt.title)
  VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title);
```

The following statements display the contents of the people_target table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID FIRST_NAME        LAST_NAME          TITLE
---------- -------------------- -------------------- ----------
     1 John         Smith              Mr
     2 alice        jones              Mrs
     3 Jane         Doe              Miss
     4 Dave         Brown              Mr

ROLLBACK;
```

The following statement compares the contents of the people_target and people_source tables by using the person_id column and conditionally inserts and updates data in the people_target table. For each matching row in the people_source table, the values in the people_target table are updated by using the values from the people_source table. Any unmatched rows from the people_source table are added to the people_target table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
  SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title
WHEN NOT MATCHED THEN INSERT
  (pt.person_id, pt.first_name, pt.last_name, pt.title)
  VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title);
```

The following statements display the contents of the people_target table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID FIRST_NAME     LAST_NAME          TITLE
---------- -------------  ------------------ ----------
     1 John          Smith              Mr
     2 Alice         Jones              Mrs.
     3 Jane          Doe                Miss
     4 Dave          Brown              Mr

ROLLBACK;
```

The following statement compares the people_target and people_source tables by using the person_id column. When the person_id matches, the corresponding rows in the people_target table are updated by using values from the people_source table. The DELETE clause removes all the values in people_target where title is 'Mrs.'. When the person_id does not match, the rows from the people_source table are added to the people_target table. The WHERE clause ensures that only values that have title as 'Mr' are added to the people_target table:

```
MERGE INTO people_target pt
USING people_source ps
ON    (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
  SET pt.first_name = ps.first_name,
      pt.last_name = ps.last_name,
      pt.title = ps.title
  DELETE where pt.title  = 'Mrs.'
WHEN NOT MATCHED THEN INSERT
  (pt.person_id, pt.first_name, pt.last_name, pt.title)
  VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title)
  WHERE ps.title = 'Mr';
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID FIRST_NAME         LAST_NAME           TITLE
---------- -------------------- -------------------- ----------
     1 John         Smith              Mr
     4 Dave         Brown              Mr

ROLLBACK;
```

Dealing with Inputs from an Application

Usually applications have to check for the existence of a row first in order to decide whether to `INSERT` a new row, or `UPDATE` an already existing one. The `MERGE` statement eliminates the need for such a check by allowing the use of bind variables inside the `USING` statement as a source.

The following statements demonstrate the use of bind variables to insert a new row into the `people_target`:

```
var person_id  NUMBER;
var first_name VARCHAR2(20);
var last_name  VARCHAR2(20);
var title      VARCHAR2(10);

exec :person_id := 3;
exec :first_name := 'Gerald';
exec :last_name := 'Walker';
exec :title := 'Mr';

MERGE INTO people_target pt
   USING (SELECT :person_id  AS person_id,
                 :first_name AS first_name,
                 :last_name  AS last_name,
                 :title      AS title FROM DUAL) ps
   ON (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
SET pt.first_name = ps.first_name,
    pt.last_name = ps.last_name,
    pt.title = ps.title
WHEN NOT MATCHED THEN INSERT
    (pt.person_id, pt.first_name, pt.last_name, pt.title)
    VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title);
```

The following statements display the contents of the `people_target table` and perform a rollback:

```
SELECT * FROM people_target;

 PERSON_ID FIRST_NAME         LAST_NAME           TITLE
```

```
---------- -------------------- -------------------- ----------
     1    John                 Smith              Mr
     2    alice                jones              Mrs
     3    Gerald               Walker              Mr

ROLLBACK;
```

The following statements demonstrate the use of bind variables to update an already existing row in the `people_target`. Note that the `MERGE` statement is identical to the one just used to insert a new row:

```
var person_id  NUMBER;
var first_name VARCHAR2(20);
var last_name  VARCHAR2(20);
var title      VARCHAR2(10);

exec :person_id := 2;
exec :first_name := 'Alice';
exec :last_name := 'Jones';
exec :title := 'Mrs';

MERGE INTO people_target pt
   USING (SELECT :person_id  AS person_id,
                 :first_name AS first_name,
                 :last_name  AS last_name,
                 :title      AS title FROM DUAL) ps
   ON (pt.person_id = ps.person_id)
WHEN MATCHED THEN UPDATE
SET pt.first_name = ps.first_name,
    pt.last_name = ps.last_name,
    pt.title = ps.title
WHEN NOT MATCHED THEN INSERT
    (pt.person_id, pt.first_name, pt.last_name, pt.title)
    VALUES (ps.person_id, ps.first_name, ps.last_name, ps.title);
```

The following statements display the contents of the `people_target` table and perform a rollback:

```
SELECT * FROM people_target;

PERSON_ID FIRST_NAME        LAST_NAME         TITLE
---------- -------------------- -------------------- ----------
     1    John                  Smith             Mr
     2    Alice                 Jones             Mrs

ROLLBACK;
```

NOAUDIT (Traditional Auditing)

This section describes the `NOAUDIT` statement for **traditional auditing**, which is the same auditing functionality used in releases earlier than Oracle Database 12*c*.

Beginning with Oracle Database 12*c*, Oracle introduces **unified auditing**, which provides a full set of enhanced auditing features. For backward compatibility, traditional auditing is still supported. However, Oracle recommends that you plan the migration of your existing audit settings to the new unified audit policy syntax. For new audit requirements, Oracle recommends that you use the new unified auditing. Traditional auditing may be desupported in a future major release.

> **See Also:**
>
> NOAUDIT (Unified Auditing) for a description of the `NOAUDIT` statement for unified auditing

Purpose

Use the `NOAUDIT` statement to stop auditing operations previously enabled by the `AUDIT` statement.

The `NOAUDIT` statement must have the same syntax as the previous `AUDIT` statement. Further, it reverses the effects only of that particular statement. For example, suppose one `AUDIT` statement A enables auditing for a specific user. A second statement B enables auditing for all users. A `NOAUDIT` statement C to disable auditing for all users reverses statement B. However, statement C leaves statement A in effect and continues to audit the user that statement A specified.

> **See Also:**
>
> AUDIT (Traditional Auditing)

Prerequisites

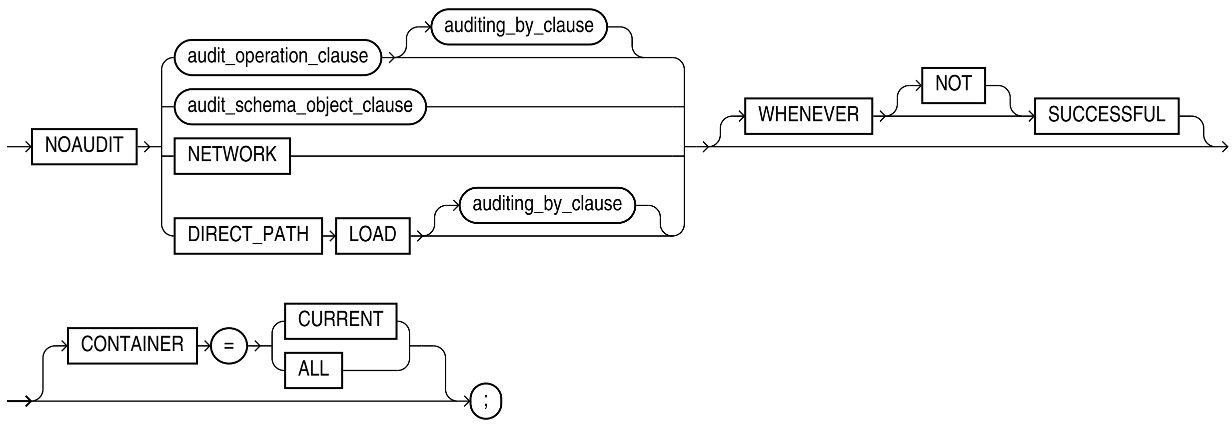
To stop auditing of SQL statements, you must have the `AUDIT SYSTEM` system privilege.

To stop auditing of schema objects, you must be the owner of the object on which you stop auditing or you must have the `AUDIT ANY` system privilege. In addition, if the object you chose for auditing is a directory, then even if you created it, you must have the `AUDIT ANY` system privilege.

To specify the `CONTAINER` clause, you must be connected to a multitenant container database (CDB). To specify `CONTAINER = ALL`, the current container must be the root and you must have the commonly granted `AUDIT SYSTEM` privilege in order to stop auditing for the issuances of a SQL statement, or the commonly granted `AUDIT ANY` privilege in order to stop auditing for the operations on a schema object. To specify `CONTAINER = CURRENT`, the current container must be a pluggable database (PDB) and you must have the locally granted `AUDIT SYSTEM` privilege in order to stop auditing the issuances of a SQL statement, or the locally granted `AUDIT ANY` privilege in order to stop auditing operations on a schema object.

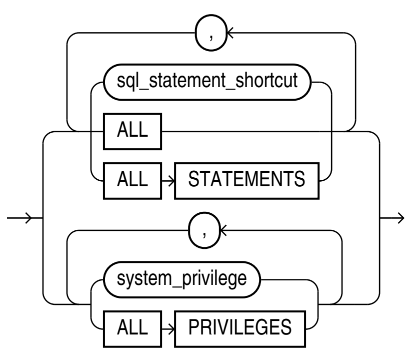
Syntax

***noaudit*::=**

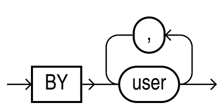


(*audit_operation_clause*::=, *auditing_by_clause*::=, *audit_schema_object_clause*::=)

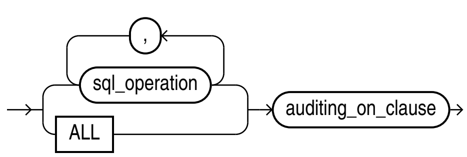
***audit_operation_clause*::=**



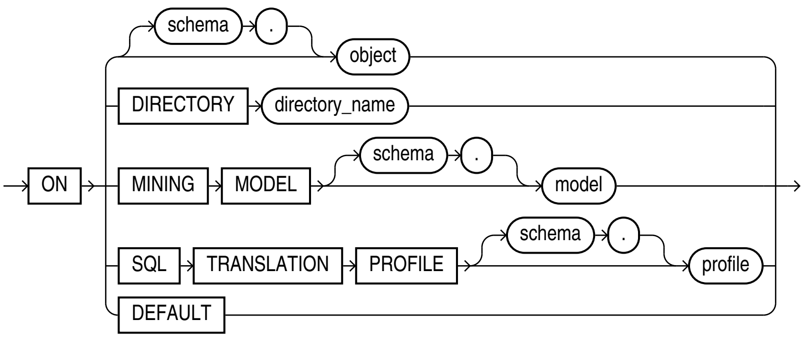
***auditing_by_clause*::=**



***audit_schema_object_clause*::=**



auditing_on_clause::=



Semantics

audit_operation_clause

Use the *audit_operation_clause* to stop auditing of a particular SQL statement.

statement_option

For *sql_statement_shortcut*, specify the shortcut for the SQL statements for which auditing is to be stopped. Refer to Table 12-1 and Table 12-2 for a list of the SQL statement shortcuts and the SQL statements they audit.

ALL

Specify `ALL` to stop auditing of all statement options currently being audited because of an earlier `AUDIT ALL ...` statement. You cannot use this clause to reverse an earlier `AUDIT ALL STATEMENTS ...` statement.

ALL STATEMENTS

Specify `ALL STATEMENTS` to reverse an earlier `AUDIT ALL STATEMENTS ...` statement. You cannot use this clause to reverse an earlier `AUDIT ALL ...` statement.

system_privilege

For *system_privilege*, specify the system privilege for which auditing is to be stopped. Refer to Table 18-1 for a list of the system privileges and the statements they authorize.

ALL PRIVILEGES

Specify `ALL PRIVILEGES` to stop auditing of all system privileges currently being audited.

auditing_by_clause

Use the *auditing_by_clause* to stop auditing only for SQL statements issued by the specified users in their subsequent sessions. If you omit this clause, then Oracle Database stops auditing for all users' statements, except for the situation described for `WHENEVER SUCCESSFUL`.

audit_schema_object_clause

Use the *audit_schema_object_clause* to stop auditing of a particular database object.

sql_operation

For *sql_operation*, specify the type of operation for which auditing is to be stopped on the object specified in the ON clause. Refer to Table 12-3 for a list of these options.

ALL

Specify ALL as a shortcut equivalent to specifying all SQL operations applicable for the type of object.

auditing_on_clause

The *auditing_on_clause* lets you specify the particular schema object for which auditing is to be stopped.

- For object, specify the object name of a table, view, sequence, stored procedure, function, or package, materialized view, or library. If you do not qualify *object* with *schema*, then Oracle Database assumes the object is in your own schema. Refer to AUDIT (Traditional Auditing) for information on auditing specific schema objects.
- The DIRECTORY clause lets you specify the name of the directory on which auditing is to be stopped.
- The SQL TRANSLATION PROFILE clause lets you specify the SQL translation profile on which auditing is to be stopped.
- Specify DEFAULT to remove the specified object options as default object options for subsequently created objects.

NETWORK

Use this clause to discontinue auditing of database link usage and logins.

DIRECT_PATH LOAD

Use this clause to discontinue auditing of SQL*Loader direct path loads.

WHENEVER [NOT] SUCCESSFUL

Specify WHENEVER SUCCESSFUL to stop auditing only for SQL statements and operations on schema objects that complete successfully.

Specify WHENEVER NOT SUCCESSFUL to stop auditing only for SQL statements and operations that result in Oracle Database errors.

If you omit this clause, then the database stops auditing for all statements or operations, regardless of success or failure.

CONTAINER Clause

Use the CONTAINER clause to specify the scope of the NOAUDIT command.

- Specify CONTAINER = CURRENT to stop auditing in the PDB to which you are connected. If you specify the *auditing_by_clause*, then *user* must be a common user or local user in the current PDB. If you specify the *auditing_on_clause*, then the objects must be local objects in the current PDB.
- Specify CONTAINER = ALL to stop auditing across the entire CDB. If you specify the *auditing_by_clause*, then *user* must be a common user. If you do not specify the *auditing_by_clause*, then auditing is stopped for all common users and all local

users in each PDB. If you specify the *auditing_on_clause*, then the objects must be common objects.

If you omit this clause, then CONTAINER = CURRENT is the default.

Examples

Stop Auditing of SQL Statements Related to Roles: Example

If you have chosen auditing for every SQL statement that creates or drops a role, then you can stop auditing of such statements by issuing the following statement:

```
NOAUDIT ROLE;
```

Stop Auditing of Updates or Queries on Objects Owned by a Particular User: Example

If you have chosen auditing for any statement that queries or updates any table issued by the users hr and oe, then you can stop auditing for queries by hr by issuing the following statement:

```
NOAUDIT SELECT TABLE BY hr;
```

The preceding statement stops auditing only queries by hr, so the database continues to audit queries and updates by oe as well as updates by hr.

Stop Auditing of Statements Authorized by a Particular Object Privilege: Example

To stop auditing on all statements that are authorized by DELETE ANY TABLE system privilege, issue the following statement:

```
NOAUDIT DELETE ANY TABLE;
```

Stop Auditing of Queries on a Particular Object: Example

If you have chosen auditing for every SQL statement that queries the employees table in the schema hr, then you can stop auditing for such queries by issuing the following statement:

```
NOAUDIT SELECT
   ON hr.employees;
```

Stop Auditing of Queries that Complete Successfully: Example

You can stop auditing for queries that complete successfully by issuing the following statement:

```
NOAUDIT SELECT
   ON hr.employees
   WHENEVER SUCCESSFUL;
```

This statement stops auditing only for successful queries. Oracle Database continues to audit queries resulting in Oracle Database errors.

NOAUDIT (Unified Auditing)

This section describes the NOAUDIT statement for **unified auditing.** This type of auditing is new beginning with Oracle Database 12*c* and provides a full set of enhanced auditing features. Refer to *Oracle Database Security Guide* for more information on unified auditing.

Purpose

Use the NOAUDIT statement to:

- Disable a unified audit policy for all users or for specified users
- Exclude the values of context attributes from audit records

Operations performed with this statement take effect in subsequent user sessions, not in the current session.

> **See Also:**
>
> - AUDIT (Unified Auditing)
> - CREATE AUDIT POLICY (Unified Auditing)
> - ALTER AUDIT POLICY (Unified Auditing)
> - DROP AUDIT POLICY (Unified Auditing)

Prerequisites

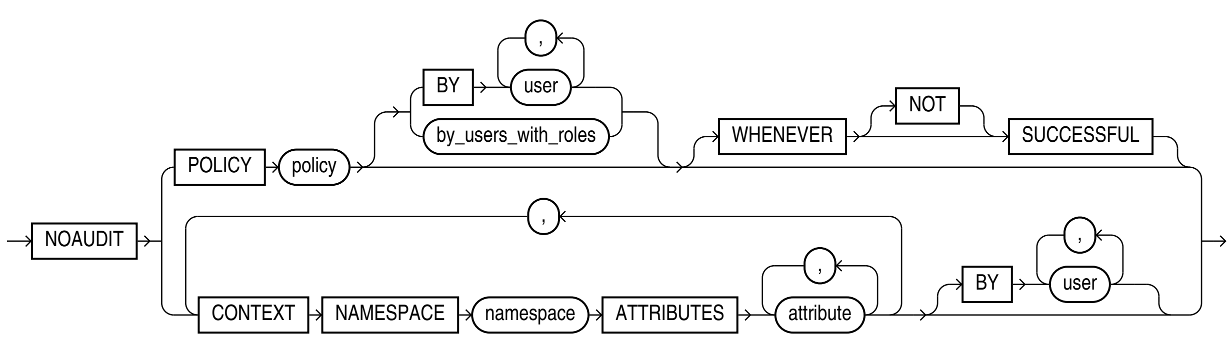
You must have the AUDIT SYSTEM system privilege or the AUDIT_ADMIN role.

If you are connected to a multitenant container database (CDB), then to disable a common unified audit policy, the current container must be the root and you must have the commonly granted AUDIT SYSTEM privilege or the AUDIT_ADMIN common role. To disable a local unified audit policy, the current container must be the container in which the audit policy was created and you must have the commonly granted AUDIT SYSTEM privilege or the AUDIT_ADMIN common role, or you must have the locally granted AUDIT SYSTEM privilege or the AUDIT_ADMIN local role in the container.

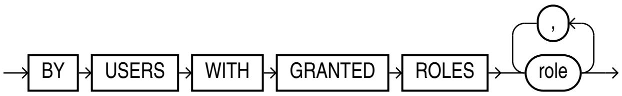
To specify the NOAUDIT CONTEXT ... statement when connected to a CDB, you must have the commonly granted AUDIT SYSTEM privilege or the AUDIT_ADMIN common role, or you must have the locally granted AUDIT SYSTEM privilege or the AUDIT_ADMIN local role in the current session's container.

Syntax

***unified_noaudit*::=**



by_users_with_roles::=



Semantics

policy

Specify the name of the unified audit policy you want to disable.

You can find descriptions of all unified audit policies by querying the `AUDIT_UNIFIED_POLICIES` view and descriptions of all *enabled* unified audit policies by querying the `AUDIT_UNIFIED_ENABLED_POLICIES` view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_POLICIES` and `AUDIT_UNIFIED_ENABLED_POLICIES` views

CONTEXT Clause

Specify the `CONTEXT` clause to exclude the values of context attributes in audit records.

- For *`namespace`*, specify the context namespace.
- For *`attribute`*, specify one or more context attributes whose values you want to exclude from audit records.

If you specify the `CONTEXT` clause when the current container is the root of a CDB, then the values of context attributes will be included in audit records only for events executed in the root. If you specify the optional `BY` clause, then *`user`* must be a common user.

If you specify the `CONTEXT` clause when the current container is a pluggable database (PDB), then the values of context attributes will be included in audit records only for events executed in that PDB. If you specify the optional `BY` clause, then *`user`* must be a common user or a local user in that PDB.

You can find the application context attributes that are configured to be captured in the audit trail by querying the `AUDIT_UNIFIED_CONTEXTS` view.

> **See Also:**
>
> *Oracle Database Reference* for more information on the `AUDIT_UNIFIED_CONTEXTS` view

BY

You can specify the `BY` clause for the `NOAUDIT POLICY` and `NOAUDIT CONTEXT` statements.

NOAUDIT POLICY ... BY

The behavior of the BY clause depends on whether *policy* is enabled for all users or specific users.

- If *policy* is enabled for all users, then you can disable *policy* for all users by omitting the BY clause. If you specify the BY clause, then the NOAUDIT POLICY statement will have no effect.
- If *policy* is enabled for one or more users (using the AUDIT POLICY ... BY ... statement), then you can:
  - Disable *policy* for one or more of those users by specifying the BY clause followed by the users for whom you want *policy* disabled
  - Completely disable *policy* by specifying the BY clause followed by all of the users for whom *policy* is enabled

  If you do not specify the BY clause, then the NOAUDIT POLICY statement will have no effect.
- If *policy* is enabled for all users except specific users (using the AUDIT POLICY ... EXCEPT ... statement), then you can disable *policy* for all users by omitting the BY clause. If you specify the BY clause, then the NOAUDIT POLICY statement will have no effect.

If *policy* is a common unified audit policy, then *user* must be a common user. If *policy* is a local unified audit policy, then *user* must be a common user or a local user in the container to which you are connected.

NOAUDIT CONTEXT ... BY

The behavior of the BY clause depends on whether *attribute* is configured to be included in audit records for all users or specific users.

- If *attribute* is configured to be included in audit records for all users, then you can exclude *attribute* from audit records for all users by omitting the BY clause. If you specify the BY clause, then the NOAUDIT CONTEXT statement will have no effect.
- If *attribute* is configured to be included in audit records for specific users, then you can exclude *attribute* for one or more of those users by specifying the BY clause followed by the users for whom you want *attribute* excluded. If you do not specify the BY clause, then the NOAUDIT CONTEXT statement will have no effect.

by_users_with_roles

Specify this clause to disable *policy* only for users who have been directly granted the specified roles. If you subsequently grant one of the roles to an additional user, then the policy is automatically disabled for that user.

When you are connected to a CDB, if *policy* is a common unified audit policy, then *role* must be a common role. If *policy* is a local unified audit policy, then role must be a common role or a local role in the container to which you are connected.

Examples

The following examples disable unified audit policies that were created in the CREATE AUDIT POLICY "Examples" and enabled in the AUDIT "Examples".

Disabling a Unified Audit Policy for All Users: Example

Assume that unified audit policy `table_pol` is enabled for all users. The following statement disables `table_pol` for all users:

```
NOAUDIT POLICY table_pol;
```

The following statement returns no rows, which verifies that `table_pol` is disabled for all users:

```
SELECT *
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'TABLE_POL';
```

Disabling a Unified Audit Policy for Specific Users: Example

Assume that unified audit policy `dml_pol` is enabled for users `hr` and `sh`, as shown by the following query:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL'
  ORDER BY entity_name;

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
DML_POL      BY           HR
DML_POL      BY           SH
```

The following statement disables `dml_pol` for user `hr`:

```
NOAUDIT POLICY dml_pol BY hr;
```

The following statement verifies that `dml_pol` is now enabled for only user `sh`:

```
SELECT policy_name, enabled_option, entity_name
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL';

POLICY_NAME  ENABLED_OPTION  ENTITY_NAME
-----------  -----------  ---------
DML_POL      BY           SH
```

The following statement disables `dml_pol` for user `sh`:

```
NOAUDIT POLICY dml_pol BY sh;
```

The following statement returns no rows, which verifies that `dml_pol` is disabled for all users:

```
SELECT *
  FROM audit_unified_enabled_policies
  WHERE policy_name = 'DML_POL';
```

Excluding Values of Context Attributes in Audit Records: Example

The following statement instructs the database to exclude the values of namespace `USERENV` attributes `CURRENT_USER` and `DB_NAME` from all audit records for user `hr`:

```
NOAUDIT CONTEXT NAMESPACE userenv
  ATTRIBUTES current_user, db_name
  BY hr;
```

PURGE

Purpose

Use the PURGE statement to:

- Remove a table or index from your recycle bin and release all of the space associated with the object
- Remove part or all of a dropped tablespace or tablespace set from the recycle bin
- Remove the entire recycle bin

> **Note:**
>
> You cannot roll back a PURGE statement, nor can you recover an object after it is purged.

To see the contents of your recycle bin, query the USER_RECYCLEBIN data dictionary view. You can use the RECYCLEBIN synonym instead. The following two statements return the same rows:

```
SELECT * FROM RECYCLEBIN;
SELECT * FROM USER_RECYCLEBIN;
```

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information on the recycle bin and naming conventions for objects in the recycle bin
> - FLASHBACK TABLE for information on retrieving dropped tables from the recycle bin
> - *Oracle Database Reference* for information on using the RECYCLEBIN initialization parameter to control whether dropped tables go into the recycle bin

Prerequisites

To purge a table, the table must reside in your own schema or you must have the DROP ANY TABLE system privilege, or you must have the SYSDBA system privilege.

To purge an index, the index must reside in your own schema or you must have the DROP ANY INDEX system privilege, or you must have the SYSDBA system privilege.

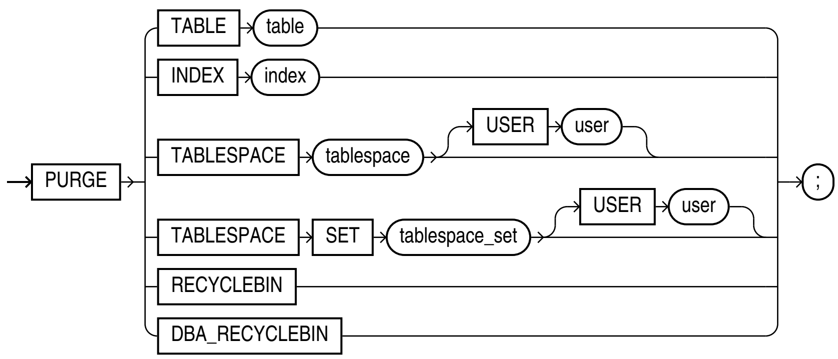
To purge a tablespace or tablespace set, you must have the DROP TABLESPACE system privilege, or you must have the SYSDBA system privilege.

To purge a tablespace set, you must also be connected to a shard catalog database as an SDB user.

To perform the `PURGE DBA_RECYCLEBIN` operation, you must have the `SYSDBA` or `PURGE DBA_RECYCLEBIN` system privilege.

Syntax

***purge*::=**



Semantics

TABLE or INDEX

Specify the name of the table or index in the recycle bin that you want to purge. You can specify either the original user-specified name or the system-generated name Oracle Database assigned to the object when it was dropped.

- If you specify the user-specified name, and if the recycle bin contains more than one object of that name, then the database purges the object that has been in the recycle bin the longest.
- System-generated recycle bin object names are unique. Therefore, if you specify the system-generated name, then the database purges that specified object.

When the database purges a table, all table partitions, LOBs and LOB partitions, indexes, and other dependent objects of that table are also purged.

TABLESPACE or TABLESPACE SET

Use this clause to purge all the objects residing in the specified tablespace or tablespace set from the recycle bin.

USER *user*

Use this clause to reclaim space in a tablespace or tablespace set for a specified user. This operation is useful when a particular user is running low on disk quota for the specified tablespace or tablespace set.

RECYCLEBIN

Use this clause to purge the current user's recycle bin. Oracle Database will remove all objects from the user's recycle bin and release all space associated with objects in the recycle bin.

DBA_RECYCLEBIN

This clause is valid only if you have the SYSDBA or PURGE DBA_RECYCLEBIN system privilege. It lets you remove all objects from the system-wide recycle bin, and is equivalent to purging the recycle bin of every user. This operation is useful, for example, before backward migration.

Examples

Remove a File From Your Recycle Bin: Example

The following statement removes the table test from the recycle bin. If more than one version of test resides in the recycle bin, then Oracle Database removes the version that has been there the longest:

```
PURGE TABLE test;
```

To determine system-generated name of the table you want removed from your recycle bin, issue a SELECT statement on your recycle bin. Using that object name, you can remove the table by issuing a statement similar to the following statement. (The system-generated name will differ from the one shown in the example.)

```
PURGE TABLE RB$$33750$TABLE$0;
```

Remove the Contents of Your Recycle Bin: Example

To remove the entire contents of your recycle bin, issue the following statement:

```
PURGE RECYCLEBIN;
```

RENAME

Purpose

> **Note:**
>
> You cannot roll back a RENAME statement.

Use the RENAME statement to rename a table, view, sequence, or private synonym.

- Oracle Database automatically transfers integrity constraints, indexes, and grants on the old object to the new object.
- Oracle Database invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

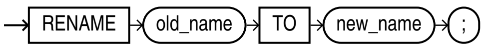
> **See Also:**
>
> CREATE SYNONYM and DROP SYNONYM

Prerequisites

The object must be in your own schema.

Syntax

rename::=



Semantics

old_name

Specify the name of an existing table, view, sequence, or private synonym.

new_name

Specify the new name to be given to the existing object. The new name must not already be used by another schema object in the same namespace and must follow the rules for naming schema objects.

Restrictions on Renaming Objects

Renaming objects is subject to the following restrictions:

- You cannot rename a public synonym. Instead, drop the public synonym and then re-create the public synonym with the new name.
- You cannot rename a type synonym that has any dependent tables or dependent valid user-defined object types.

> **See Also:**
>
> "Database Object Naming Rules "

Examples

Renaming a Database Object: Example

The following example uses a copy of the sample table `hr.departments`. To change the name of table `departments_new` to `emp_departments`, issue the following statement:

```
RENAME departments_new TO emp_departments;
```

You cannot use this statement directly to rename columns. However, you can rename a column using the `ALTER TABLE` ... *`rename_column_clause`*.

> **See Also:**
>
> *rename_column_clause*

Another way to rename a column is to use the RENAME statement together with the CREATE TABLE statement with AS *subquery*. This method is useful if you are changing the structure of a table rather than only renaming a column. The following statements re-create the sample table hr.job_history, renaming a column from department_id to dept_id:

```
CREATE TABLE temporary
   (employee_id, start_date, end_date, job_id, dept_id)
AS SELECT
     employee_id, start_date, end_date, job_id, department_id
FROM job_history;

DROP TABLE job_history;

RENAME temporary TO job_history;
```

Any integrity constraints defined on table job_history will be lost in the preceding example. You will have to redefine them on the new job_history table using an ALTER TABLE statement.

REVOKE

Purpose

Use the REVOKE statement to:

- Revoke system privileges from users and roles
- Revoke roles from users, roles, and program units.
- Revoke object privileges for a particular object from users and roles

Note on Oracle Automatic Storage Management

A user authenticated AS SYSASM can use this statement to revoke the system privileges SYSASM, SYSOPER, and SYSDBA from a user in the Oracle ASM password file of the current node.

Note on Editionable Objects

A REVOKE operation to revoke object privileges on an editionable object actualizes the object in the current edition. See *Oracle Database Development Guide* for more information about editions and editionable objects.

> **See Also:**
>
> - GRANT for information on granting system privileges and roles
> - Table 18-2 for a listing of the object privileges for each type of object

Prerequisites

To revoke a **system privilege,** you must have been granted the privilege with the ADMIN OPTION. You can revoke any privilege if you have the GRANT ANY PRIVILEGE system privilege.

To revoke a **role from a user or another role**, you must have been directly granted the role with the ADMIN OPTION or you must have created the role. You can revoke any role if you have the GRANT ANY ROLE system privilege.

To revoke a **role from a program unit**, you must be the user SYS or you must be the schema owner of the program unit.

To revoke an **object privilege**, one of the following conditions must be met:

- You must previously have granted the object privilege to the user or role.
- You must have the GRANT ANY OBJECT PRIVILEGE system privilege. In this case, you can revoke any object privilege that was granted by the object owner or on behalf of the owner by a user with the GRANT ANY OBJECT PRIVILEGE. However, you cannot revoke an object privilege that was granted by way of a WITH GRANT OPTION grant.

> **See Also:**
>
> "Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example"

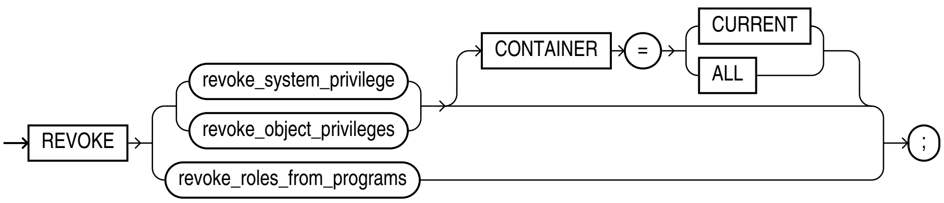
The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revokee
- Roles or object privileges granted through the operating system
- Privileges or roles granted to the revokee through roles

To specify the CONTAINER clause, you must be connected to a multitenant container database (CDB). To specify CONTAINER = ALL, the current container must be the root.

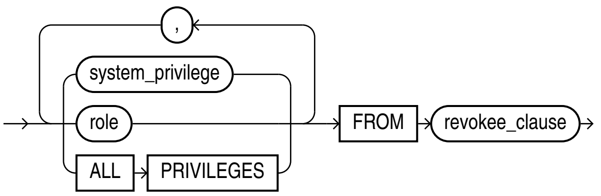
Syntax

***revoke*::=**



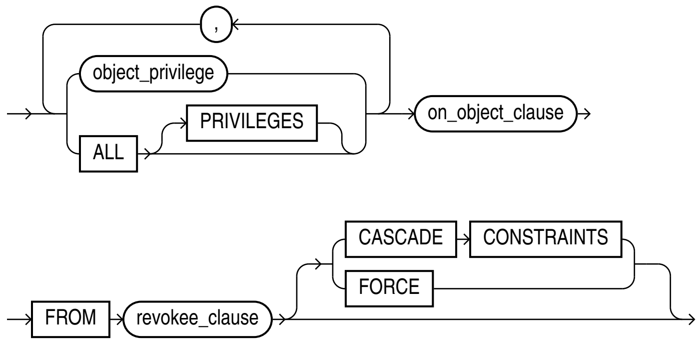
(*revoke_system_privileges::=*, *revoke_object_privileges::=*, *revoke_roles_from_programs::=*)

***revoke_system_privileges*::=**



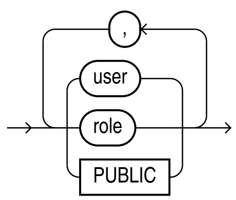
(*revokee_clause::=*)

***revoke_object_privileges*::=**

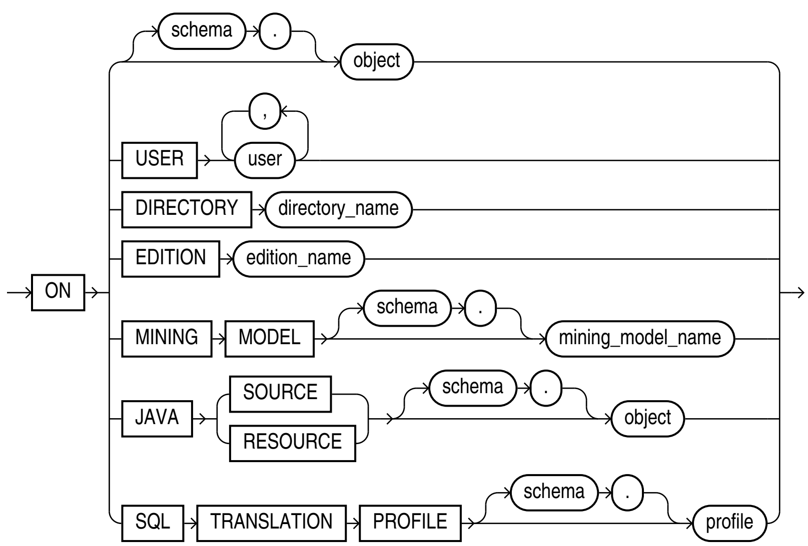


(*on_object_clause::=*, *revokee_clause::=*)

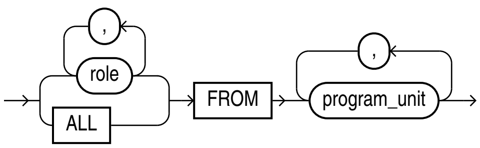
***revokee_clause*::=**



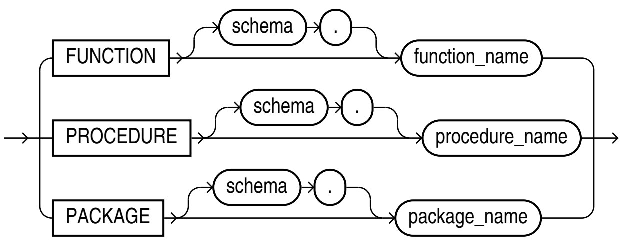
***on_object_clause*::=**



***revoke_roles_from_programs*::=**



***program_unit*::=**



Semantics

revoke_system_privileges

Use these clauses to revoke system privileges.

system_privilege

Specify the system privilege to be revoked. Refer to Table 18-1 for a list of the system privileges.

If you revoke a system privilege from a **user**, then the database removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

If you revoke a system privilege from a **role**, then the database removes the privilege from the privilege domain of the role. Effective immediately, users with the role enabled cannot exercise the privilege. Also, other users who have been granted the role and subsequently enable the role cannot exercise the privilege.

> **See Also:**
>
> "Revoking a System Privilege from a User: Example" and "Revoking a System Privilege from a Role: Example"

If you revoke a system privilege from **`PUBLIC`**, then the database removes the privilege from the privilege domain of each user who has been granted the privilege through `PUBLIC`. Effective immediately, such users can no longer exercise the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Oracle Database provides a shortcut for specifying all system privileges at once: Specify `ALL PRIVILEGES` to revoke all the system privileges listed in Table 18-1.

Restriction on Revoking System Privileges

A system privilege cannot appear more than once in the list of privileges to be revoked.

role

Specify the role to be revoked.

If you revoke a role from a **user**, then the database makes the role unavailable to the user. If the role is currently enabled for the user, then the user can continue to exercise the privileges in the role's privilege domain as long as it remains enabled. However, the user cannot subsequently enable the role.

If you revoke a role from another **role**, then the database removes the privilege domain of the revoked role from the privilege domain of the revokee role. Users who have been granted and have enabled the revokee role can continue to exercise the privileges in the privilege domain of the revoked role as long as the revokee role remains enabled. However, other users who have been granted the revokee role and subsequently enable it cannot exercise the privileges in the privilege domain of the revoked role.

> **See Also:**
>
> "Revoking a Role from a User: Example" and "Revoking a Role from a Role: Example"

If you revoke a role from **`PUBLIC`**, then the database makes the role unavailable to all users who have been granted the role through `PUBLIC`. Any user who has enabled the role can continue to exercise the privileges in its privilege domain as long as it remains enabled. However, users cannot subsequently enable the role. The role is not revoked from users who have been granted the role directly or through other roles.

Restriction on Revoking System Roles

A system role cannot appear more than once in the list of roles to be revoked. For information on the predefined roles, refer to *Oracle Database Security Guide*.

revokee_clause

Use the *`revokee_clause`* to specify the users or roles from which the system privilege, role, or object privilege is to be revoked.

PUBLIC

Specify `PUBLIC` to revoke the privileges or roles from all users.

revoke_object_privileges

Use these clauses to revoke object privileges.

object_privilege

Specify the object privilege to be revoked. The object privileges, categorized by the type of object to which they apply, are described in Table 18-2.

> **Note:**
>
> Each privilege authorizes some operation. By revoking a privilege, you prevent the revokee from performing that operation. However, multiple users may grant the same privilege to the same user, role, or `PUBLIC`. To remove the privilege from the grantee's privilege domain, all grantors must revoke the privilege. If even one grantor does not revoke the privilege, then the grantee can still exercise the privilege by virtue of that grant.

If you revoke an object privilege from a **user**, then the database removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

- If that user has granted that privilege to other users or roles, then the database also revokes the privilege from those other users or roles.
- If that user's schema contains a procedure, function, or package that contains SQL statements that exercise the privilege, then the procedure, function, or package can no longer be executed.
- If that user's schema contains a view on that object, then the database invalidates the view.
- If you revoke the `REFERENCES` object privilege from a user who has exercised the privilege to define referential integrity constraints, then you must specify the `CASCADE CONSTRAINTS` clause.

If you revoke an object privilege from a **role**, then the database removes the privilege from the privilege domain of the role. Effective immediately, users with the role enabled cannot exercise the privilege. Other users who have been granted the role cannot exercise the privilege after enabling the role.

If you revoke an object privilege from `PUBLIC`, then the database removes the privilege from the privilege domain of each user who has been granted the privilege through `PUBLIC`. Effective immediately, all such users are restricted from exercising the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

ALL [PRIVILEGES]

Specify `ALL` to revoke all object privileges that you have granted to the revokee. (The keyword `PRIVILEGES` is provided for semantic clarity and is optional.)

If no privileges have been granted on the object, then the database takes no action and does not return an error.

Restriction on Revoking Object Privileges

A privilege cannot appear more than once in the list of privileges to be revoked. A user, a role, or `PUBLIC` cannot appear more than once in the `FROM` clause.

See Also:

"Revoking an Object Privilege from a User: Example", "Revoking Object Privileges from PUBLIC: Example", and "Revoking All Object Privileges from a User: Example"

CASCADE CONSTRAINTS

This clause is relevant only if you revoke the `REFERENCES` privilege or `ALL` [`PRIVILEGES`]. It drops any referential integrity constraints that the revokee has defined using the `REFERENCES` privilege, which might have been granted either explicitly or implicitly through a grant of `ALL` [`PRIVILEGES`].

See Also:

"Revoking an Object Privilege with CASCADE CONSTRAINTS: Example"

FORCE

Specify `FORCE` to revoke the `EXECUTE` object privilege on user-defined type objects with table or type dependencies. You must use `FORCE` to revoke the `EXECUTE` object privilege on user-defined type objects with table dependencies.

If you specify `FORCE`, then all privileges are revoked, all dependent objects are marked `INVALID`, data in dependent tables becomes inaccessible, and all dependent function-based indexes are marked `UNUSABLE`. Regranting the necessary type privilege will revalidate the table.

See Also:

Oracle Database Concepts for detailed information about type dependencies and user-defined object privileges

on_object_clause

The *`on_object_clause`* identifies the objects on which privileges are to be revoked.

object

Specify the object on which the object privileges are to be revoked. This object can be:

- A table, view, sequence, procedure, stored function, package, or materialized view
- A synonym for a table, view, sequence, procedure, stored function, package, materialized view, or user-defined type
- A library, indextype, or user-defined operator

If you do not qualify object with *`schema`*, then the database assumes the object is in your own schema.

> **See Also:**
>
> "Revoking an Object Privilege on a Sequence from a User: Example"

If you revoke the `READ` or `SELECT` object privilege on the containing table or materialized view of a materialized view, whether the privilege was granted with or without the `GRANT OPTION`, then the database invalidates the materialized view.

If you revoke the `READ` or `SELECT` object privilege on any of the master tables of a materialized view, whether the privilege was granted with or without the `GRANT OPTION`, then the database invalidates both the materialized view and its containing table or materialized view.

ON USER

Specify the database user you want to revoke privileges from.

> **See Also:**
>
> "Revoking an Object Privilege on a User from a User: Example"

ON DIRECTORY

Specify the name of the directory object on which privileges are to be revoked. You cannot qualify *`directory_name`* with a schema name.

> **See Also:**
>
> CREATE DIRECTORY and "Revoking an Object Privilege on a Directory from a User: Example"

ON EDITION

Specify the name of the edition on which the `USE` object privilege is to be revoked. You cannot qualify *`edition_name`* with a schema name.

ON MINING MODEL

Specify the name of the mining model on which privileges are to be revoked. If you do not qualify *`mining_model_name`* with *`schema`*, then the database assumes that the mining model is in your own schema.

ON JAVA SOURCE | RESOURCE

Specify the name of the Java source or resource schema object on which privileges are to be revoked. If you do not qualify *`object`* with *`schema`*, then the database assumes that the object is in your own schema.

ON SQL TRANSLATION PROFILE

Specify the name of the SQL translation profile on which privileges are to be revoked. If you do not qualify *`profile`* with *`schema`*, then the database assumes the profile is in your own schema.

revoke_roles_from_programs

Use this clause to revoke code based access control (CBAC) roles from program units.

role

Specify the role you want to revoke.

ALL

Specify `ALL` to revoke all roles that are granted to the program unit.

program_unit

Specify the program unit from which the role is to be revoked. You can specify a PL/SQL function, procedure, or package. If you do not specify *`schema`*, then Oracle Database assumes the function, procedure, or package is in your own schema.

> **See Also:**
>
> *Oracle Database Security Guide* for more information on revoking CBAC roles from program units

CONTAINER Clause

If the current container is a pluggable database (PDB):

- Specify `CONTAINER = CURRENT` to revoke a locally granted system privilege, object privilege, or role from a local user, common user, local role, or common role. The privilege or role is revoked from the user or role only in the current PDB. This clause does not revoke privileges granted with `CONTAINER = ALL`.

If the current container is the root:

- Specify `CONTAINER = CURRENT` to revoke a locally granted system privilege, object privilege, or role from a common user or common role. The privilege or role is revoked from the user or role only in the root. This clause does not revoke privileges granted with `CONTAINER = ALL`.
- Specify `CONTAINER = ALL` to revoke a commonly granted system privilege, object privilege on a common object, or role from a common user or common role. The privilege or role is revoked from the user or role across the entire CDB. This clause can revoke only a privilege or role granted with `CONTAINER = ALL` from the specified common user or common role. This clause does not revoke privileges granted locally with `CONTAINER = CURRENT`. However, any locally granted privileges that depend on the commonly granted privilege being revoked are also revoked.

If you omit this clause, then `CONTAINER = CURRENT` is the default.

Examples

Revoking a System Privilege from a User: Example

The following statement revokes the DROP ANY TABLE system privilege from the users hr and oe:

```
REVOKE DROP ANY TABLE
    FROM hr, oe;
```

The users hr and oe can no longer drop tables in schemas other than their own.

Revoking a Role from a User: Example

The following statement revokes the role dw_manager from the user sh:

```
REVOKE dw_manager
    FROM sh;
```

The user sh can no longer enable the dw_manager role.

Revoking a System Privilege from a Role: Example

The following statement revokes the CREATE TABLESPACE system privilege from the dw_manager role:

```
REVOKE CREATE TABLESPACE
   FROM dw_manager;
```

Enabling the dw_manager role no longer allows users to create tablespaces.

Revoking a Role from a Role: Example

To revoke the role dw_user from the role dw_manager, issue the following statement:

```
REVOKE dw_user
  FROM dw_manager;
```

The dw_user role privileges are no longer granted to dw_manager.

Revoking an Object Privilege from a User: Example

You can grant DELETE, INSERT, READ, SELECT, and UPDATE privileges on the table orders to the user hr with the following statement:

```
GRANT ALL
   ON orders TO hr;
```

To revoke the DELETE privilege on orders from hr, issue the following statement:

```
REVOKE DELETE
   ON orders FROM hr;
```

Revoking All Object Privileges from a User: Example

To revoke the remaining privileges on orders that you granted to hr, issue the following statement:

```
REVOKE ALL
   ON orders FROM hr;
```

Revoking Object Privileges from PUBLIC: Example

You can grant SELECT and UPDATE privileges on the view emp_details_view to all users by granting the privileges to the role PUBLIC:

```
GRANT SELECT, UPDATE
    ON emp_details_view TO public;
```

The following statement revokes UPDATE privilege on emp_details_view from all users:

```
REVOKE UPDATE
    ON emp_details_view FROM public;
```

Users can no longer update the emp_details_view view, although users can still query it. However, if you have also granted the UPDATE privilege on emp_details_view to any users, either directly or through roles, then these users retain the privilege.

Revoking an Object Privilege on a User from a User: Example

You can grant the user hr the INHERIT PRIVILEGES privilege on user sh with the following statement:

```
GRANT INHERIT PRIVILEGES ON USER sh TO hr;
```

To revoke the INHERIT PRIVILEGES privilege on user sh from user hr, issue the following statement:

```
REVOKE INHERIT PRIVILEGES ON USER sh FROM hr;
```

Revoking an Object Privilege on a Sequence from a User: Example

You can grant the user oe the SELECT privilege on the departments_seq sequence in the schema hr with the following statement:

```
GRANT SELECT
    ON hr.departments_seq TO oe;
```

To revoke the SELECT privilege on departments_seq from oe, issue the following statement:

```
REVOKE SELECT
    ON hr.departments_seq FROM oe;
```

However, if the user hr has also granted SELECT privilege on departments to sh, then sh can still use departments by virtue of hr's grant.

Revoking an Object Privilege with CASCADE CONSTRAINTS: Example

You can grant to oe the privileges REFERENCES and UPDATE on the employees table in the schema hr with the following statement:

```
GRANT REFERENCES, UPDATE
    ON hr.employees TO oe;
```

The user oe can exercise the REFERENCES privilege to define a constraint in his or her own dependent table that refers to the employees table in the schema hr:

```
CREATE TABLE dependent
(dependno   NUMBER,
 dependname VARCHAR2(10),
 employee   NUMBER
    CONSTRAINT in_emp REFERENCES hr.employees(employee_id) );
```

You can revoke the REFERENCES privilege on hr.employees from oe by issuing the following statement that contains the CASCADE CONSTRAINTS clause:

```
REVOKE REFERENCES
    ON hr.employees
    FROM oe
    CASCADE CONSTRAINTS;
```

Revoking oe's REFERENCES privilege on hr.employees causes Oracle Database to drop the in_emp constraint, because oe required the privilege to define the constraint.

However, if oe has also been granted the REFERENCES privilege on hr.employees by a user other than you, then the database does not drop the constraint. oe still has the privilege necessary for the constraint by virtue of the other user's grant.

Revoking an Object Privilege on a Directory from a User: Example

You can revoke the READ object privilege on directory bfile_dir from hr by issuing the following statement:

```
REVOKE READ ON DIRECTORY bfile_dir FROM hr;
```

Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example

Suppose that the database administrator has granted GRANT ANY OBJECT PRIVILEGE to user sh. Now suppose that user hr grants the update privilege on the employees table to oe:

```
CONNECT hr
GRANT UPDATE ON employees TO oe WITH GRANT OPTION;
```

This grant gives user oe the right to pass the object privilege along to another user:

```
CONNECT oe
GRANT UPDATE ON hr.employees TO pm;
```

User sh, who has the GRANT ANY OBJECT PRIVILEGE, can now act on behalf of user hr and revoke the update privilege from user oe, because oe was granted the privilege by hr:

```
CONNECT sh
REVOKE UPDATE ON hr.employees FROM oe;
```

User sh cannot revoke the update privilege from user pm explicitly, because pm received the grant neither from the object owner (hr), nor from sh, nor from another user with GRANT ANY OBJECT PRIVILEGE, but from user oe. However, the preceding statement cascades, removing all privileges that depend on the one revoked. Therefore the object privilege is implicitly revoked from pm as well.

ROLLBACK

Purpose

Use the ROLLBACK statement to undo work done in the current transaction or to manually undo the work done by an in-doubt distributed transaction.

Note:

Oracle recommends that you explicitly end transactions in application programs using either a `COMMIT` or `ROLLBACK` statement. If you do not explicitly commit the transaction and the program terminates abnormally, then Oracle Database rolls back the last uncommitted transaction.

See Also:

- *Oracle Database Concepts* for information on transactions
- *Oracle Database Heterogeneous Connectivity User's Guide* for information on distributed transactions
- SET TRANSACTION for information on setting characteristics of the current transaction
- COMMIT and SAVEPOINT

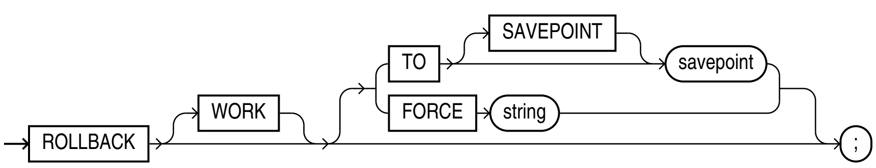
Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have the `FORCE TRANSACTION` system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have the `FORCE ANY TRANSACTION` system privilege.

Syntax

***rollback*::=**



Semantics

WORK

The keyword `WORK` is optional and is provided for SQL standard compatibility.

TO SAVEPOINT Clause

Specify the savepoint to which you want to roll back the current transaction. If you omit this clause, then the `ROLLBACK` statement rolls back the entire transaction.

Using `ROLLBACK` without the `TO SAVEPOINT` clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases any transaction locks

> **See Also:**
>
> SAVEPOINT

Using `ROLLBACK` with the `TO SAVEPOINT` clause performs the following operations:

- Rolls back just the portion of the transaction after the savepoint. It does not end the transaction.
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- Releases all table and row locks acquired since the savepoint. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

Restriction on In-doubt Transactions

You cannot manually roll back an in-doubt transaction to a savepoint.

FORCE Clause

Specify `FORCE` to manually roll back an in-doubt distributed transaction. The transaction is identified by the *`string`* containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view `DBA_2PC_PENDING`.

A `ROLLBACK` statement with a `FORCE` clause rolls back only the specified transaction. Such a statement does not affect your current transaction.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on distributed transactions and rolling back in-doubt transactions

Examples

Rolling Back Transactions: Examples

The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

The following statement rolls back your current transaction to savepoint `banda_sal`:

```
ROLLBACK TO SAVEPOINT banda_sal;
```

See "Creating Savepoints: Example" for a full version of the preceding example.

The following statement manually rolls back an in-doubt distributed transaction:

```
ROLLBACK WORK
    FORCE '25.32.87';
```

SAVEPOINT

Purpose

Use the `SAVEPOINT` statement to create a name for a system change number (SCN), to which you can later roll back.

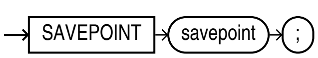
> **See Also:**
>
> - *Oracle Database Concepts* for information on savepoints.
> - ROLLBACK for information on rolling back transactions
> - SET TRANSACTION for information on setting characteristics of the current transaction

Prerequisites

None.

Syntax

***savepoint*::=**



Semantics

savepoint

Specify the name of the savepoint to be created.

Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, then the earlier savepoint is erased. After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

Examples

Creating Savepoints: Example

To update the salary for `Banda` and `Greene` in the sample table `hr.employees`, check that the total department salary does not exceed 314,000, then reenter the salary for `Greene`:

```
UPDATE employees
    SET salary = 7000
    WHERE last_name = 'Banda';
SAVEPOINT banda_sal;

UPDATE employees
    SET salary = 12000
    WHERE last_name = 'Greene';
SAVEPOINT greene_sal;

SELECT SUM(salary) FROM employees;

ROLLBACK TO SAVEPOINT banda_sal;

UPDATE employees
    SET salary = 11000
    WHERE last_name = 'Greene';

COMMIT;
```

SELECT

Purpose

Use a `SELECT` statement or subquery to retrieve data from one or more tables, object tables, views, object views, or materialized views.

If part or all of the result of a `SELECT` statement is equivalent to an existing materialized view, then Oracle Database may use the materialized view in place of one or more tables specified in the `SELECT` statement. This substitution is called **query rewrite**. It takes place only if cost optimization is enabled and the `QUERY_REWRITE_ENABLED` parameter is set to `TRUE`. To determine whether query rewrite has occurred, use the `EXPLAIN PLAN` statement.

> **See Also:**
>
> - SQL Queries and Subqueries for general information on queries and subqueries
> - *Oracle Database Data Warehousing Guide* for more information on materialized views and query rewrite
> - EXPLAIN PLAN

Prerequisites

For you to select data from a table or materialized view, the table or materialized view must be in your own schema or you must have the `READ` or `SELECT` privilege on the table or materialized view.

For you to select rows from the base tables of a view:

- The view must be in your own schema or you must have the `READ` or `SELECT` privilege on the view, and
- Whoever owns the schema containing the view must have the `READ` or `SELECT` privilege on the base tables.

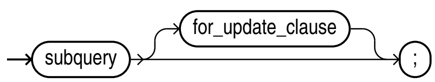
The READ ANY TABLE or SELECT ANY TABLE system privilege also allows you to select data from any table or any materialized view or the base table of any view.

To specify the FOR UPDATE clause, the preceding prerequisites apply with the following exception: The READ and READ ANY TABLE privileges, where mentioned, do not allow you to specify the FOR UPDATE clause.

To issue an Oracle Flashback Query using the *flashback_query_clause*, you must have the READ or SELECT privilege on the objects in the select list. In addition, either you must have FLASHBACK object privilege on the objects in the select list, or you must have FLASHBACK ANY TABLE system privilege.

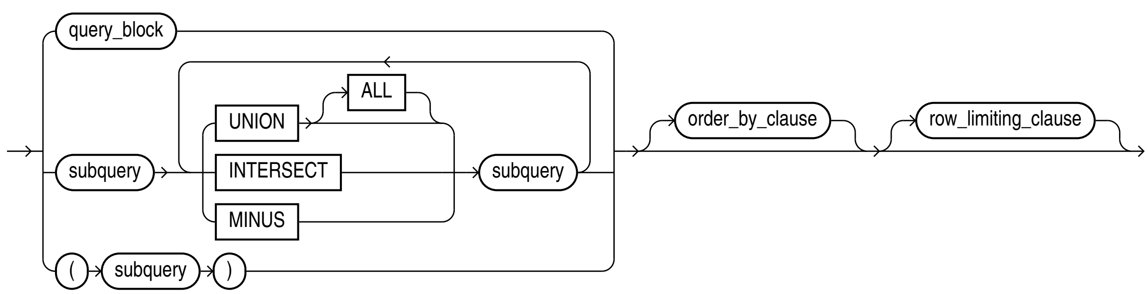
Syntax

***select*::=**



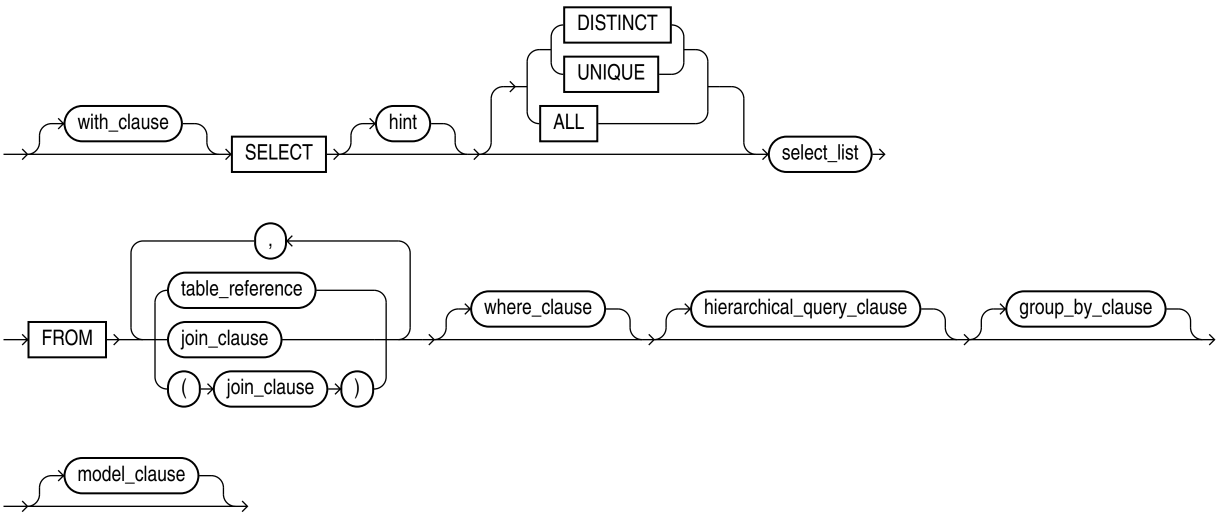
(*subquery*::=, *for_update_clause*::=)

***subquery*::=**



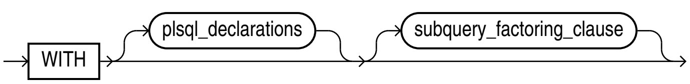
(*query_block*::=, *order_by_clause*::=, *row_limiting_clause*::=)

***query_block*::=**



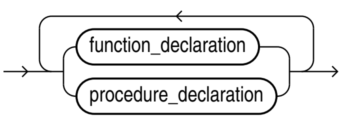
(*with_clause*::=, *select_list*::=, *table_reference*::=, *join_clause*::=, *where_clause*::=, *hierarchical_query_clause*::=, *group_by_clause*::=, *model_clause*::=)

***with_clause*::=**

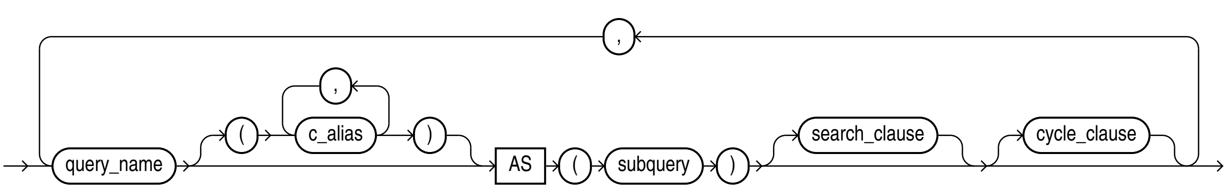


> **Note:**
>
> You cannot specify only the `WITH` keyword. You must specify at least one of the clauses *`plsql_declarations`* or *`subquery_factoring_clause`*.

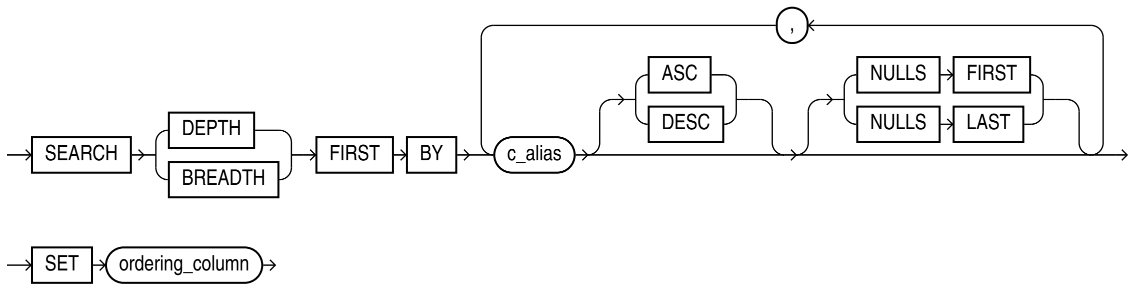
***plsql_declarations*::=**



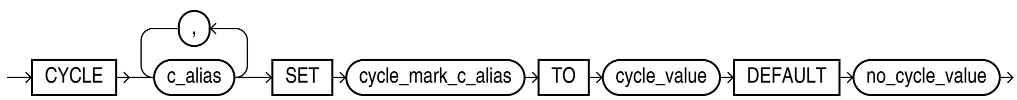
***subquery_factoring_clause*::=**



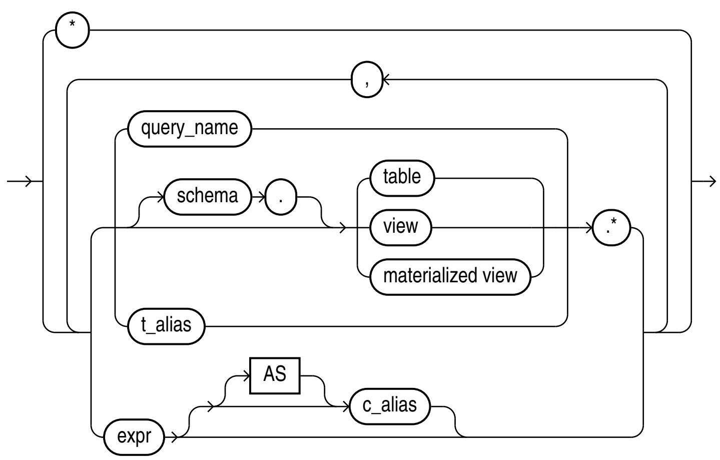
***search_clause*::=**



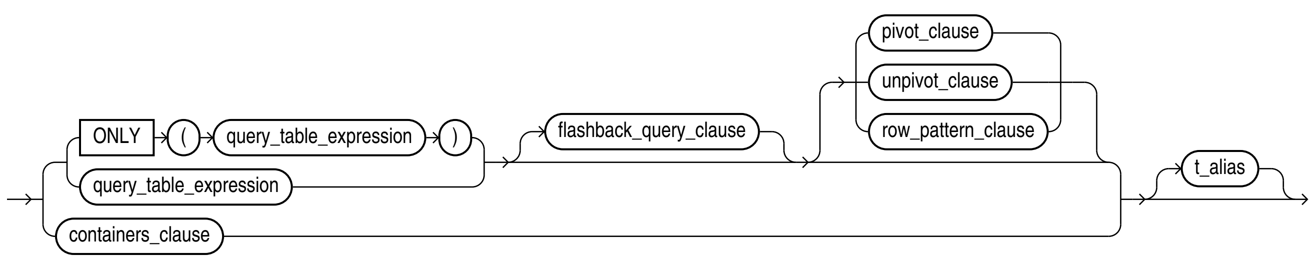
***cycle_clause*::=**



***select_list*::=**

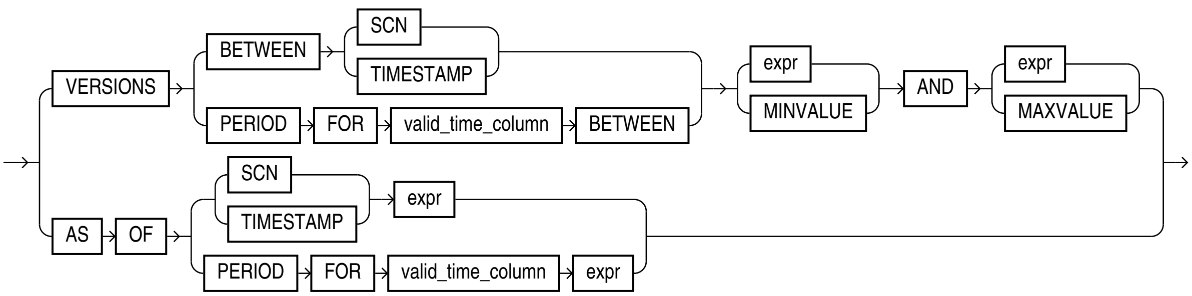


***table_reference*::=**

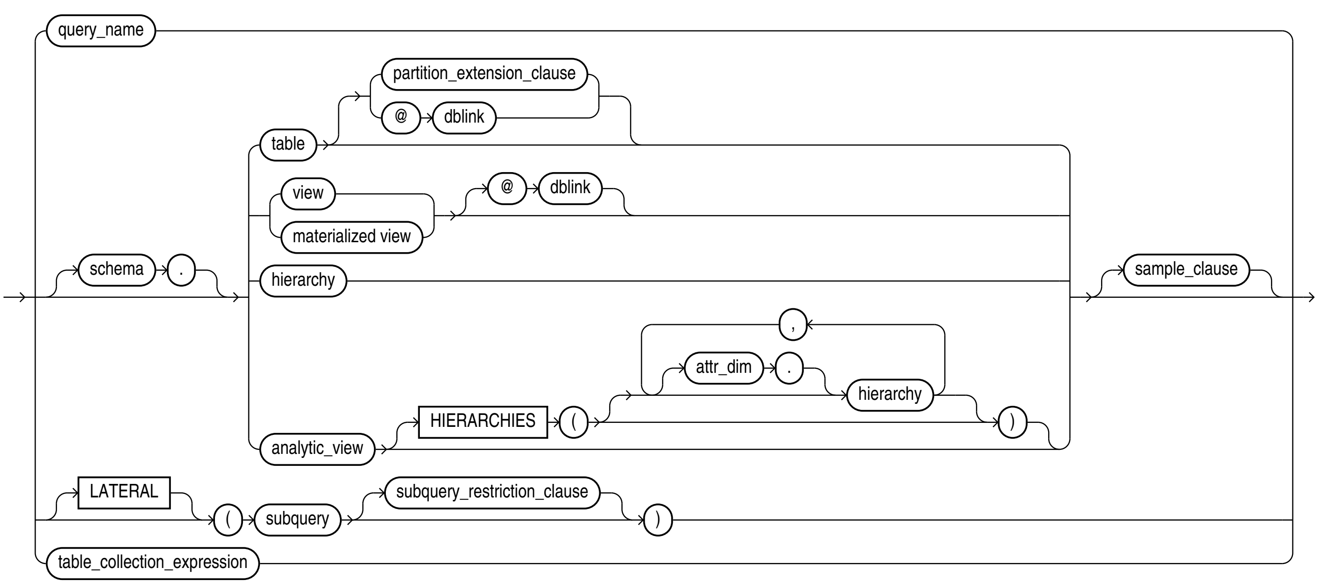


(*query_table_expression*::=, *flashback_query_clause*::=, *pivot_clause*::=, *unpivot_clause*::=, *row_pattern_clause*::=, *containers_clause*::=)

***flashback_query_clause*::=**

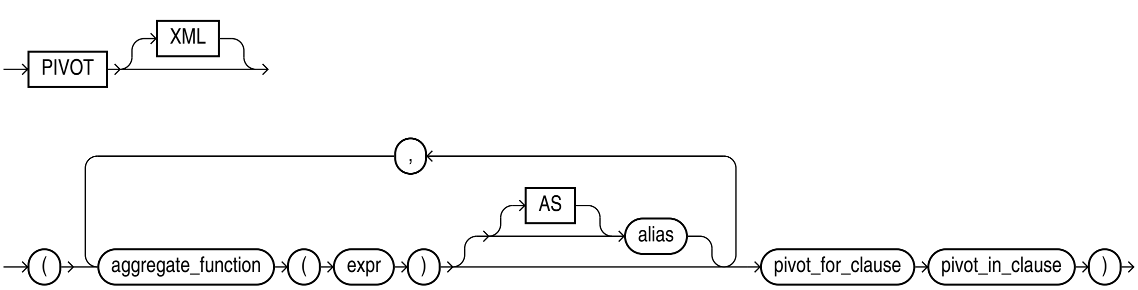


***query_table_expression*::=**

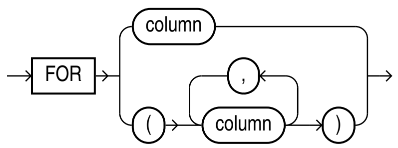


(*subquery_restriction_clause*::=, *table_collection_expression*::=)

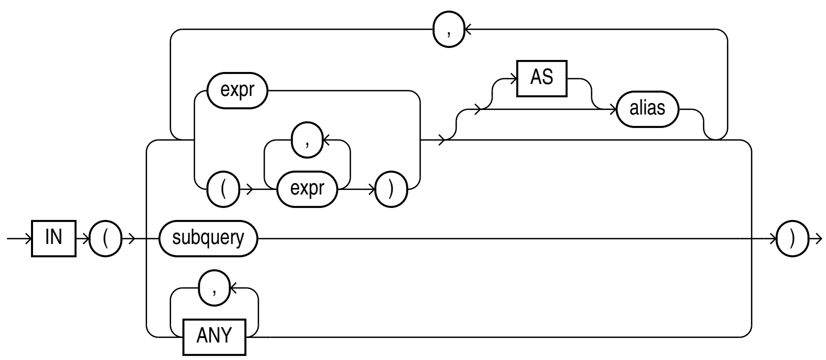
***pivot_clause*::=**



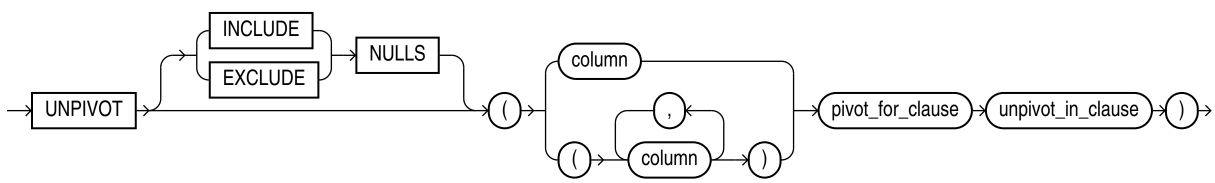
***pivot_for_clause*::=**



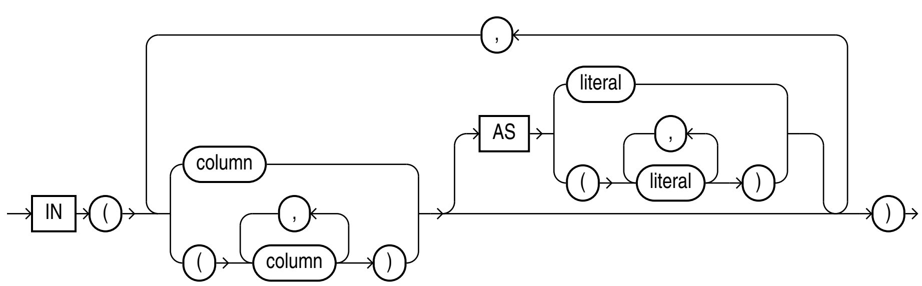
***pivot_in_clause*::=**



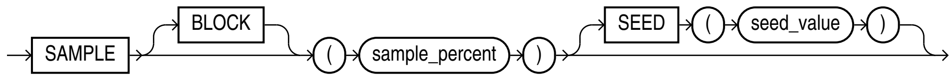
***unpivot_clause*::=**



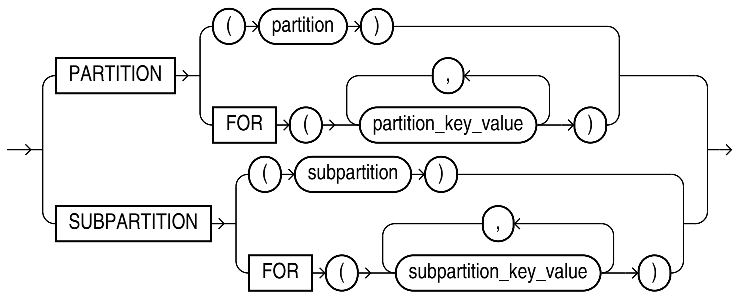
***unpivot_in_clause*::=**



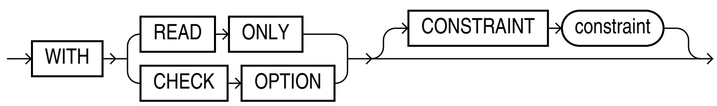
***sample_clause*::=**



***partition_extension_clause*::=**

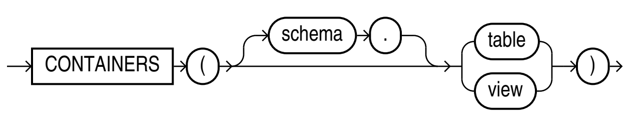


***subquery_restriction_clause*::=**

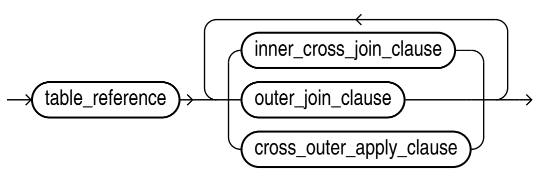


***table_collection_expression*::=**

***containers_clause*::=**

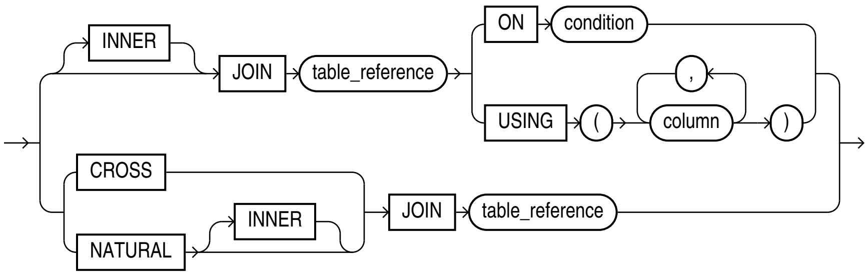


***join_clause*::=**



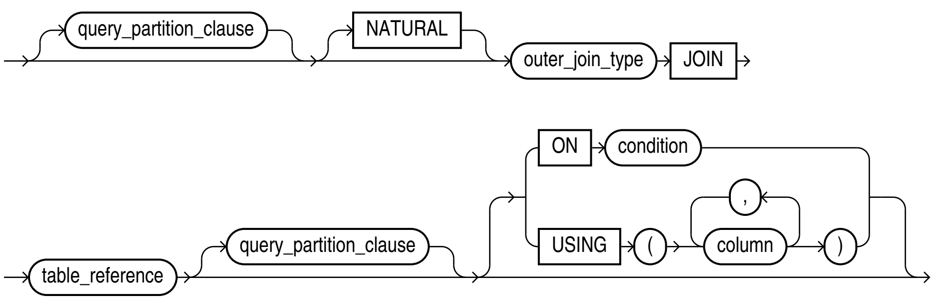
(*inner_cross_join_clause*::=, *outer_join_clause*::=, *cross_outer_apply_clause*::=)

***inner_cross_join_clause*::=**



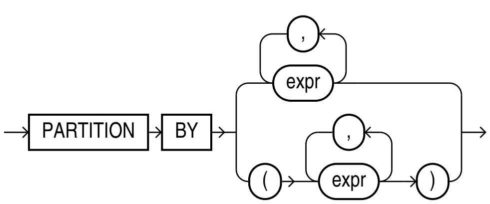
(*table_reference*::=)

***outer_join_clause*::=**

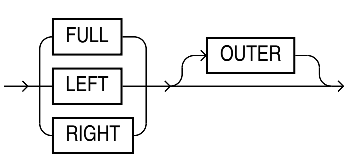


(*query_partition_clause*::=, *outer_join_type*::=, *table_reference*::=)

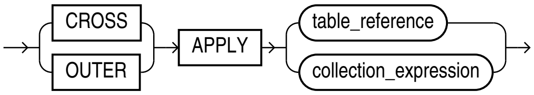
***query_partition_clause*::=**



***outer_join_type*::=**

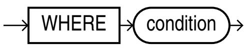


***cross_outer_apply_clause*::=**

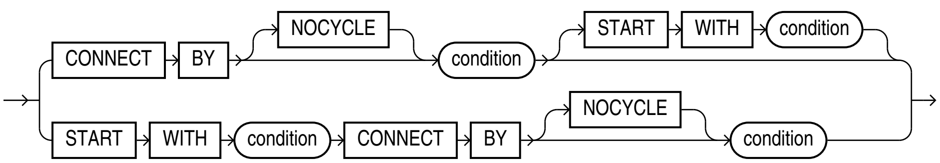


(*table_reference*::=, *query_partition_clause*::=)

***where_clause*::=**

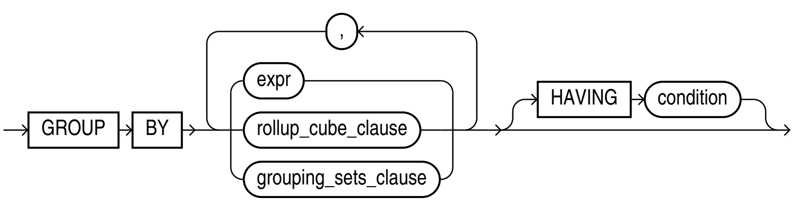


***hierarchical_query_clause*::=**



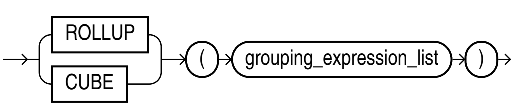
(`condition` can be any condition as described in Conditions)

***group_by_clause*::=**



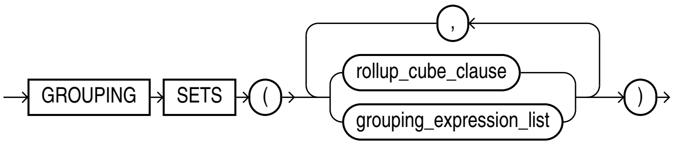
(*rollup_cube_clause*::=, *grouping_sets_clause*::=)

***rollup_cube_clause*::=**



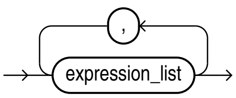
(*grouping_expression_list*::=)

***grouping_sets_clause*::=**

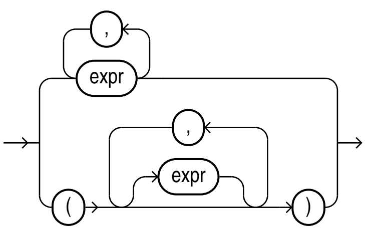


(*rollup_cube_clause::=*, *grouping_expression_list::=*)

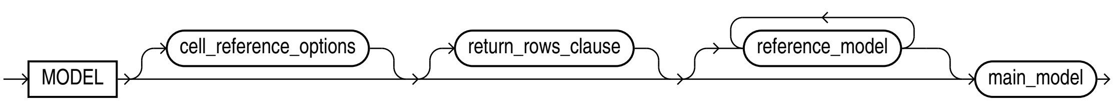
***grouping_expression_list*::=**



***expression_list*::=**

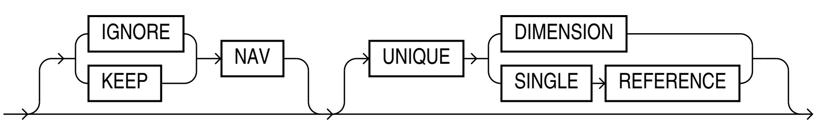


***model_clause*::=**

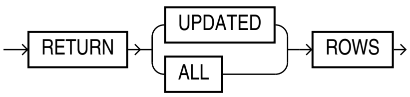


(*cell_reference_options::=*, *return_rows_clause::=*, *reference_model::=*, *main_model::=*)

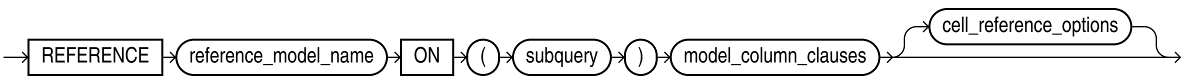
***cell_reference_options*::=**



***return_rows_clause*::=**

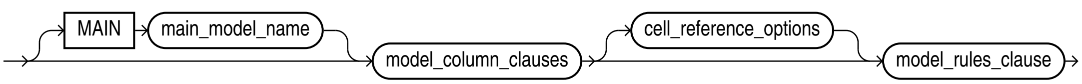


***reference_model*::=**



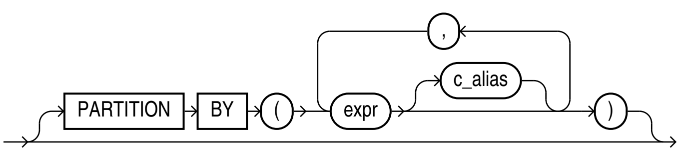
(*model_column_clauses*::=, *cell_reference_options*::=)

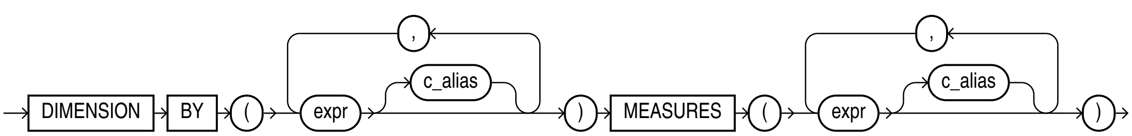
***main_model*::=**



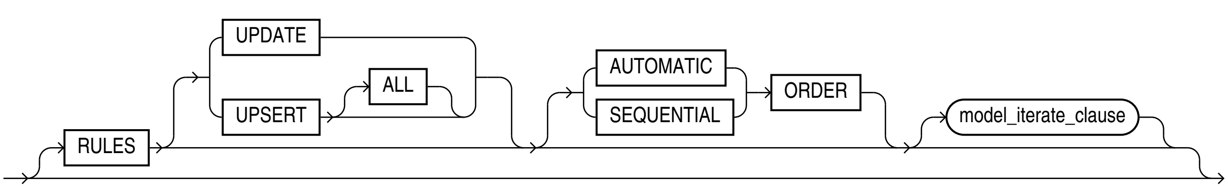
(*model_column_clauses*::=, *cell_reference_options*::=, *model_rules_clause*::=)

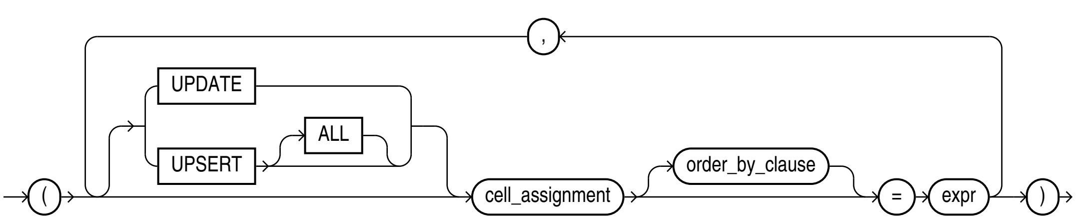
***model_column_clauses*::=**





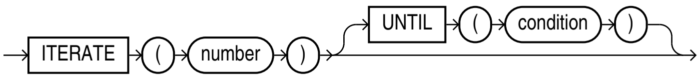
***model_rules_clause*::=**



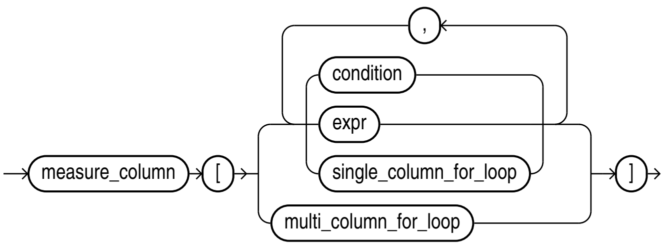


(*model_iterate_clause*::=, *cell_assignment*::=, *order_by_clause*::=)

***model_iterate_clause*::=**

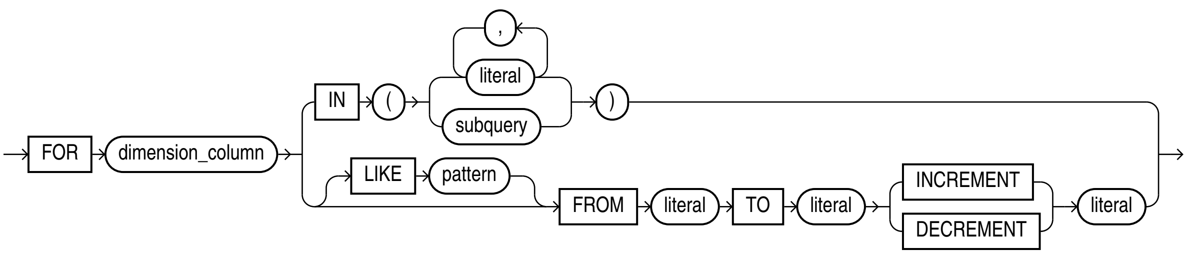


***cell_assignment*::=**

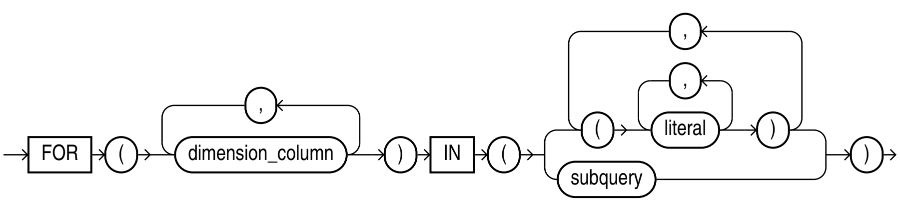


(*single_column_for_loop*::=, *multi_column_for_loop*::=)

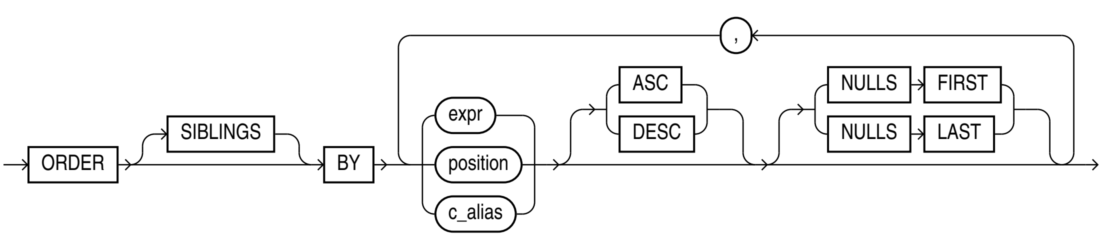
***single_column_for_loop*::=**



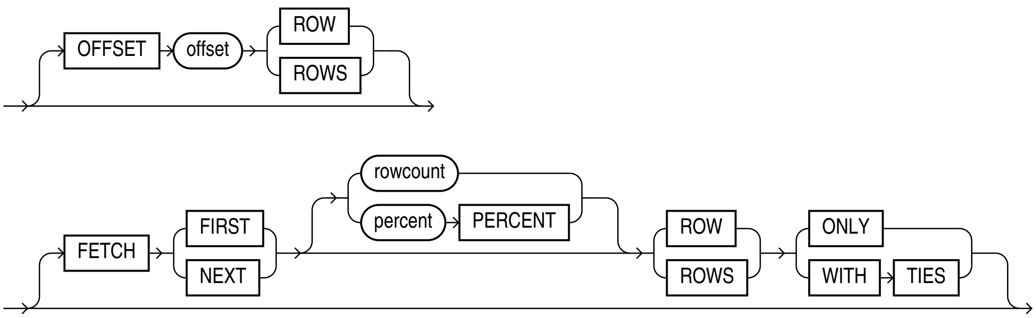
***multi_column_for_loop*::=**



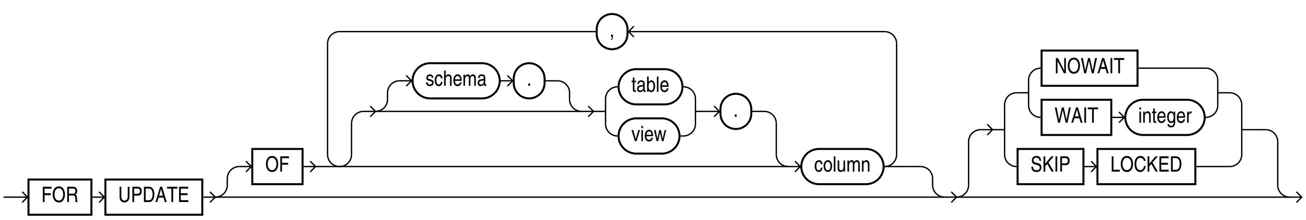
***order_by_clause*::=**



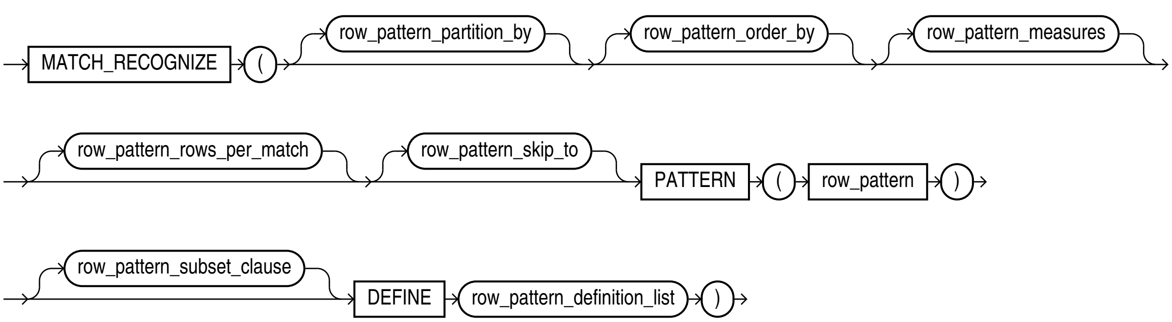
***row_limiting_clause*::=**



***for_update_clause*::=**

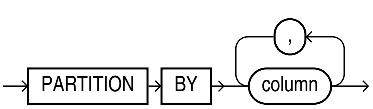


***row_pattern_clause*::=**

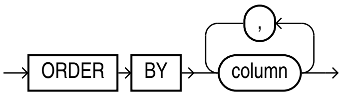


(*row_pattern_partition_by*::=, *row_pattern_order_by*::=, *row_pattern_measures*::=, *row_pattern_rows_per_match*::=, *row_pattern_skip_to*::=, *row_pattern*::=, *row_pattern_subset_clause*::=, *row_pattern_definition_list*::=)

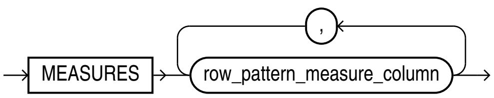
***row_pattern_partition_by*::=**



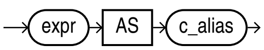
***row_pattern_order_by*::=**



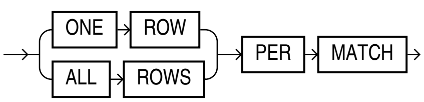
***row_pattern_measures*::=**



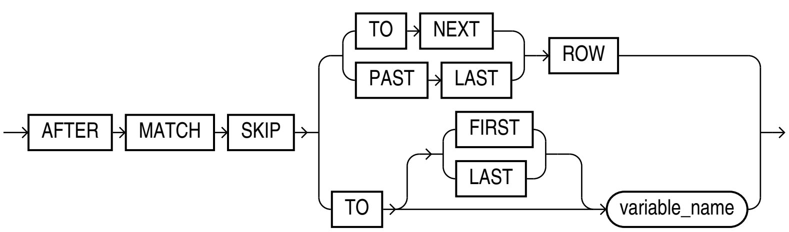
***row_pattern_measure_column*::=**



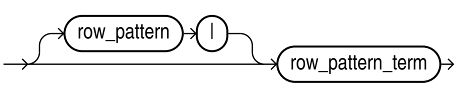
***row_pattern_rows_per_match*::=**



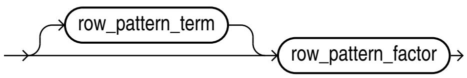
***row_pattern_skip_to*::=**



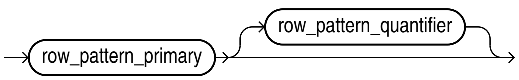
***row_pattern*::=**



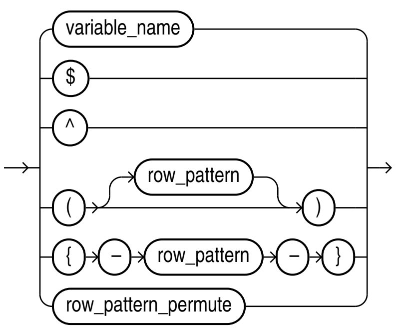
***row_pattern_term*::=**



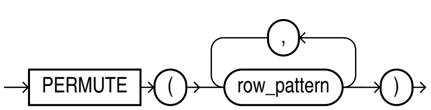
***row_pattern_factor*::=**



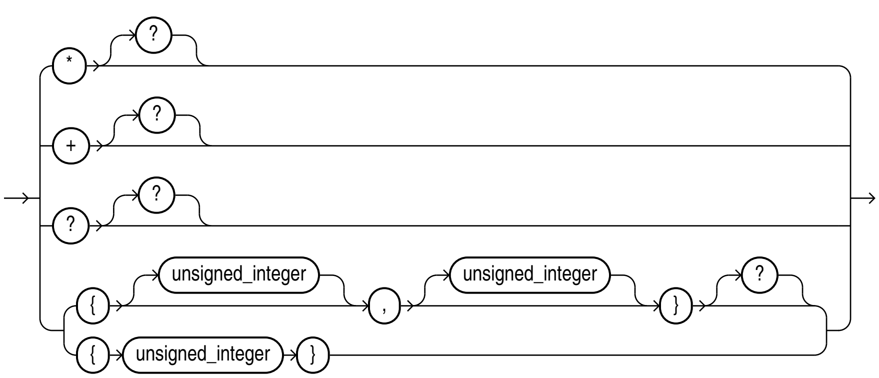
***row_pattern_primary*::=**



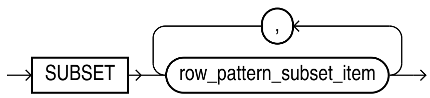
***row_pattern_permute*::=**



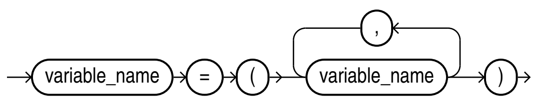
***row_pattern_quantifier*::=**



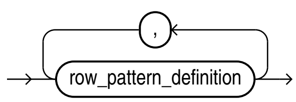
***row_pattern_subset_clause*::=**



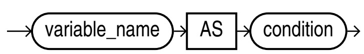
***row_pattern_subset_item*::=**



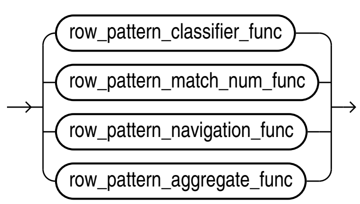
***row_pattern_definition_list*::=**



***row_pattern_definition*::=**



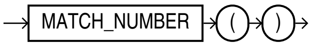
***row_pattern_rec_func*::=**



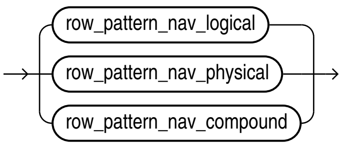
(*row_pattern_classifier_func*::=, *row_pattern_match_num_func*::=, *row_pattern_navigation_func*::=, *row_pattern_aggregate_func*::=)

***row_pattern_classifier_func*::=**

***row_pattern_match_num_func*::=**

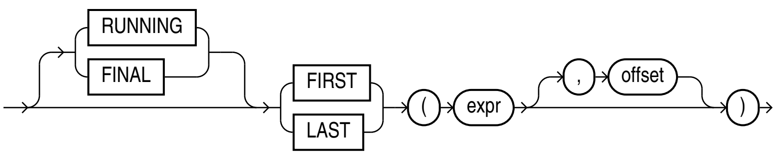


***row_pattern_navigation_func*::=**

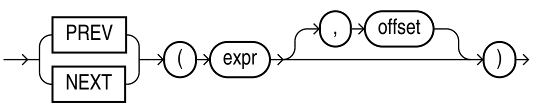


(*row_pattern_nav_logical*::=, *row_pattern_nav_physical*::=, *row_pattern_nav_compound*::=)

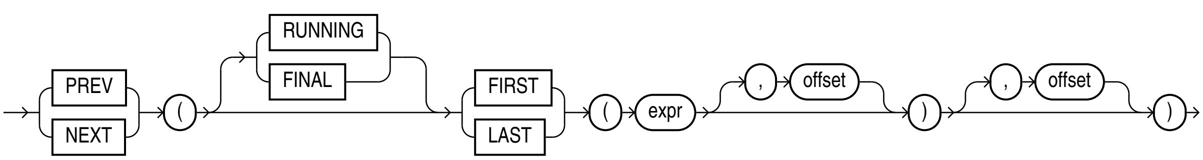
***row_pattern_nav_logical*::=**



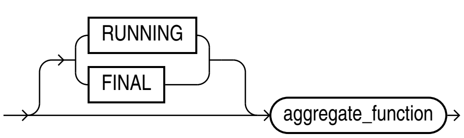
***row_pattern_nav_physical*::=**



***row_pattern_nav_compound*::=**



***row_pattern_aggregate_func*::=**



Semantics

with_clause

Use the *with_clause* to define the following:

- PL/SQL procedures and functions (using the *plsql_declarations* clause)
- Subquery blocks (using the *subquery_factoring_clause*)

plsql_declarations

The *plsql_declarations* clause lets you declare and define PL/SQL functions and procedures. You can then reference the PL/SQL functions in the query in which you specify this clause, as well as its subqueries, if any. For the purposes of name resolution, these function names have precedence over schema-level stored functions.

If the query in which you specify this clause is not a top-level `SELECT` statement, then the following rules apply to the top-level SQL statement that contains the query:

- If the top-level statement is a `SELECT` statement, then it must have either a `WITH` *plsql_declarations* clause or the `WITH_PLSQL` hint.
- If the top-level statement is a `DELETE`, `MERGE`, `INSERT`, or `UPDATE` statement, then it must have the `WITH_PLSQL` hint.

The `WITH_PLSQL` hint only enables you to specify the `WITH` *plsql_declarations* clause within the statement. It is not an optimizer hint.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for syntax and restrictions for *function_declaration* and *procedure_declaration*.
> - "Using a PL/SQL Function in the WITH Clause: Examples"

subquery_factoring_clause

The *subquery_factoring_clause* lets you assign a name (*query_name*) to a subquery block. You can then reference the subquery block multiple places in the query by specifying *query_name*. Oracle Database optimizes the query by treating the *query_name* as either an inline view or as a temporary table. The *query_name* is subject to the same naming conventions and restrictions as database schema objects. Refer to "Database Object Naming Rules " for information on database object names.

The column aliases following the *query_name* and the set operators separating multiple subqueries in the `AS` clause are valid and required for recursive subquery factoring. The *search_clause* and *cycle_clause* are valid only for recursive subquery factoring but are not required. See "Recursive Subquery Factoring".

You can specify this clause in any top-level `SELECT` statement and in most types of subqueries. The query name is visible to the main query and to all subsequent subqueries. For recursive subquery factoring, the query name is even visible to the subquery that defines the query name itself.

Recursive Subquery Factoring

If a *subquery_factoring_clause* refers to its own *query_name* in the subquery that defines it, then the *subquery_factoring_clause* is said to be **recursive**. A recursive *subquery_factoring_clause* must contain two query blocks: the first is the **anchor member** and the second is the **recursive member**. The anchor member must appear before the recursive member, and it cannot reference *query_name*. The anchor member can be composed of one or more query blocks combined by the set operators: `UNION ALL`, `UNION`, `INTERSECT` or `MINUS`. The recursive member must follow the anchor member and must reference *query_name* exactly once. You must combine the recursive member with the anchor member using the `UNION ALL` set operator.

The number of column aliases following `WITH` *query_name* and the number of columns in the `SELECT` lists of the anchor and recursive query blocks must be the same.

The recursive member cannot contain any of the following elements:

- The `DISTINCT` keyword or a `GROUP BY` clause
- The *model_clause*
- An aggregate function. However, analytic functions are permitted in the select list.
- Subqueries that refer to *query_name*.
- Outer joins that refer to *query_name* as the right table.

In previous releases of Oracle Database, the recursive member of a recursive `WITH` clause ran serially regardless of the parallelism of the entire query (also known as the top-level `SELECT` statement). Beginning with Oracle Database 12*c* Release 2 (12.2), the recursive member will run in parallel if the optimizer determines that the top-level `SELECT` statement can be executed in parallel.

search_clause

Use the `SEARCH` clause to specify an ordering for the rows.

- Specify `BREADTH FIRST BY` if you want sibling rows returned before any child rows are returned.
- Specify `DEPTH FIRST BY` if you want child rows returned before any siblings rows are returned.
- Sibling rows are ordered by the columns listed after the `BY` keyword.
- The *c_alias* list following the `SEARCH` keyword must contain column names from the column alias list for *query_name*.
- The *ordering_column* is automatically added to the column list for the query name. The query that selects from *query_name* can include an `ORDER BY` on *ordering_column* to return the rows in the order that was specified by the `SEARCH` clause.

cycle_clause

Use the `CYCLE` clause to mark cycles in the recursion.

- The *c_alias* list following the `CYCLE` keyword must contain column names from the column alias list for *query_name*. Oracle Database uses these columns to detect a cycle.
- *cycle_value* and *no_cycle_value* should be character strings of length 1.

- If a cycle is detected, then the cycle mark column specified by *cycle_mark_c_alias* for the row causing the cycle is set to the value specified for *cycle_value*. The recursion will then stop for this row. That is, it will not look for child rows for the offending row, but it will continue for other noncyclic rows.
- If no cycles are found, then the cycle mark column is set to the default value specified for *no_cycle_value*.
- The cycle mark column is automatically added to the column list for the *query_name*.
- A row is considered to form a cycle if one of its ancestor rows has the same values for the cycle columns.

If you omit the `CYCLE` clause, then the recursive `WITH` clause returns an error if cycles are discovered. In this case, a row forms a cycle if one of its ancestor rows has the same values for all the columns in the column alias list for *query_name* that are referenced in the `WHERE` clause of the recursive member.

Restrictions on Subquery Factoring

This clause is subject to the following restrictions:

- You can specify only one *subquery_factoring_clause* in a single SQL statement. Any *query_name* defined in the *subquery_factoring_clause* can be used in any subsequent named query block in the *subquery_factoring_clause*.
- In a compound query with set operators, you cannot use the *query_name* for any of the component queries, but you can use the *query_name* in the `FROM` clause of any of the component queries.
- You cannot specify duplicate names in the column alias list for *query_name*.
- The name used for the *ordering_column* has to be different from the name used for *cycle_mark_c_alias*.
- The *ordering_column* and cycle mark column names cannot already be in the column alias list for *query_name*.

See Also:

- *Oracle Database Concepts* for information about inline views
- "Subquery Factoring: Example"
- "Recursive Subquery Factoring: Examples"

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

See Also:

"Hints " for the syntax and description of hints

DISTINCT | UNIQUE

Specify DISTINCT or UNIQUE if you want the database to return only one copy of each set of duplicate rows selected. These two keywords are synonymous. Duplicate rows are those with matching values for each expression in the select list.

Restrictions on DISTINCT and UNIQUE Queries

These types of queries are subject to the following restrictions:

- When you specify DISTINCT or UNIQUE, the total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter DB_BLOCK_SIZE.
- You cannot specify DISTINCT if the *select_list* contains LOB columns.

ALL

Specify ALL if you want the database to return all rows selected, including all copies of duplicates. The default is ALL.

select_list

The *select_list* lets you specify the columns you want to retrieve from the database.

*** (all-column wildcard)**

Specify the all-column wildcard (asterisk) to select all columns, excluding pseudocolumns and INVISIBLE columns, from all tables, views, or materialized views listed in the FROM clause. The columns are returned in the order indicated by the COLUMN_ID column of the *_TAB_COLUMNS data dictionary view for the table, view, or materialized view.

If you are selecting from a table rather than from a view or a materialized view, then columns that have been marked as UNUSED by the ALTER TABLE SET UNUSED statement are not selected.

> **See Also:**
>
> ALTER TABLE, "Simple Query Examples", and "Selecting from the DUAL Table: Example"

query_name.*

Specify *query_name* followed by a period and the asterisk to select all columns from the specified subquery block. For *query_name*, specify a subquery block name already specified in the *subquery_factoring_clause*. You must have specified the *subquery_factoring_clause* in order to specify *query_name* in the *select_list*. If you specify *query_name* in the *select_list*, then you also must specify *query_name* in the *query_table_expression* (FROM clause).

table. | view.* | materialized view.**

Specify the object name followed by a period and the asterisk to select all columns from the specified table, view, or materialized view. Oracle Database returns a set of columns in the order in which the columns were specified when the object was created. A query that selects rows from two or more tables, views, or materialized views is a join.

You can use the schema qualifier to select from a table, view, or materialized view in a schema other than your own. If you omit `schema`, then the database assumes the table, view, or materialized view is in your own schema.

> **See Also:**
>
> "Joins "

*t_alias .**

Specify a correlation name (alias) followed by a period and the asterisk to select all columns from the object with that correlation name specified in the `FROM` clause of the same subquery. The object can be a table, view, materialized view, or subquery. Oracle Database returns a set of columns in the order in which the columns were specified when the object was created. A query that selects rows from two or more objects is a join.

expr

Specify an expression representing the information you want to select. A column name in this list can be qualified with `schema` only if the table, view, or materialized view containing the column is qualified with `schema` in the `FROM` clause. If you specify a member method of an object type, then you must follow the method name with parentheses even if the method takes no arguments.

c_alias

Specify an alias for the column expression. Oracle Database will use this alias in the column heading of the result set. The `AS` keyword is optional. The alias effectively renames the select list item for the duration of the query. The alias can be used in the `order_by_clause` but not other clauses in the query.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for information on using the `expr AS c_alias` syntax with the `UNION ALL` operator in queries of multiple materialized views
> - "About SQL Expressions " for the syntax of `expr`

Restrictions on the Select List

The select list is subject to the following restrictions:

- If you also specify a *group_by_clause* in this statement, then this select list can contain only the following types of expressions:
  - Constants
  - Aggregate functions and the functions `USER`, `UID`, and `SYSDATE`
  - Expressions identical to those in the `group_by_clause`. If the `group_by_clause` is in a subquery, then all columns in the select list of the subquery must match

the `GROUP BY` columns in the subquery. If the select list and `GROUP BY` columns of a top-level query or of a subquery do not match, then the statement results in ORA-00979.

  – Expressions involving the preceding expressions that evaluate to the same value for all rows in a group

- You can select a rowid from a join view only if the join has one and only one key-preserved table. The rowid of that table becomes the rowid of the view.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on key-preserved tables

- If two or more tables have some column names in common, and if you are specifying a join in the `FROM` clause, then you must qualify column names with names of tables or table aliases.

FROM Clause

The `FROM` clause lets you specify the objects from which data is selected.

ONLY

The `ONLY` clause applies only to views. Specify `ONLY` if the view in the `FROM` clause is a view belonging to a hierarchy and you do not want to include rows from any of its subviews.

query_table_expression

Use the *`query_table_expression`* clause to identify a subquery block, table, view, materialized view, analytic view, hierarchy, partition, or subpartition, or to specify a subquery that identifies the objects. In order to specify a subquery block, you must have specified the subquery block name (*`query_name`*) in the *`subquery_factoring_clause`*.

> **See Also:**
>
> "Using Subqueries: Examples"

LATERAL

Specify `LATERAL` to designate *`subquery`* as a lateral inline view. Within a lateral inline view, you can specify tables that appear to the left of the lateral inline view in the `FROM` clause of a query. You can specify this left correlation anywhere within *`subquery`* (such as the `SELECT`, `FROM`, and `WHERE` clauses) and at any nesting level.

Restrictions on LATERAL

Lateral inline views are subject to the following restrictions:

- If you specify `LATERAL`, then you cannot specify the *`pivot_clause`*, the *`unpivot_clause`*, or a pattern in the *`table_reference`* clause.
- If a lateral inline view contains the *`query_partition_clause`*, and it is the right side of a join clause, then it cannot contain a left correlation to the left table in the join clause.

However, it can contain a left correlation to a table to its left in the `FROM` clause that is not the left table.

- A lateral inline view cannot contain a left correlation to the first table in a right outer join or full outer join.

> **See Also:**
>
> "Using Lateral Inline Views: Example"

flashback_query_clause

Use the *`flashback_query_clause`* to retrieve data from a table, view, or materialized view based on time dimensions associated with the data.

This clause implements SQL-driven Flashback, which lets you specify the following:

- A different system change number or timestamp for each object in the select list, using the clauses `VERSIONS BETWEEN { SCN | TIMESTAMP }` or `VERSIONS AS OF { SCN | TIMESTAMP }`. You can also implement session-level Flashback using the `DBMS_FLASHBACK` package.
- A valid time period for each object in the select list, using the clauses `VERSIONS PERIOD FOR` or `AS OF PERIOD FOR`. You can also implement valid-time session-level Flashback using the `DBMS_FLASHBACK_ARCHIVE` package.

A Flashback Query lets you retrieve a history of changes made to a row. You can retrieve the corresponding identifier of the transaction that made the change using the `VERSIONS_XID` pseudocolumn. You can also retrieve information about the transaction that resulted in a particular row version by issuing an Oracle Flashback Transaction Query. You do this by querying the `FLASHBACK_TRANSACTION_QUERY` data dictionary view for a particular transaction ID.

VERSIONS BETWEEN { SCN | TIMESTAMP }

Specify `VERSIONS BETWEEN` to retrieve multiple versions of the rows returned by the query. Oracle Database returns all committed versions of the rows that existed between two SCNs or between two timestamp values. The first specified SCN or timestamp must be earlier than the second specified SCN or timestamp. The rows returned include deleted and subsequently reinserted versions of the rows.

- Specify `VERSIONS BETWEEN SCN` ... to retrieve the versions of the row that existed between two SCNs. Both expressions must evaluate to a number and cannot evaluate to NULL. `MINVALUE` and `MAXVALUE` resolve to the SCN of the oldest and most recent data available, respectively.
- Specify `VERSIONS BETWEEN TIMESTAMP` ... to retrieve the versions of the row that existed between two timestamps. Both expressions must evaluate to a timestamp value and cannot evaluate to NULL. `MINVALUE` and `MAXVALUE` resolve to the timestamp of the oldest and most recent data available, respectively.

AS OF { SCN | TIMESTAMP }

Specify `AS OF` to retrieve the single version of the rows returned by the query at a particular change number (SCN) or timestamp. If you specify `SCN`, then *`expr`* must evaluate to a number. If you specify `TIMESTAMP`, then *`expr`* must evaluate to a timestamp value. In either case, *expr* cannot evaluate to NULL. Oracle Database returns rows as they existed at the specified system change number or time.

Oracle Database provides a group of version query pseudocolumns that let you retrieve additional information about the various row versions. Refer to "Version Query Pseudocolumns " for more information.

When both clauses are used together, the `AS OF` clause determines the SCN or moment in time from which the database issues the query. The `VERSIONS` clause determines the versions of the rows as seen from the `AS OF` point. The database returns null for a row version if the transaction started before the first `BETWEEN` value or ended after the `AS OF` point.

VERSIONS PERIOD FOR

Specify `VERSIONS PERIOD FOR` to retrieve rows from *table* based on whether they are considered valid during the specified time period. In order to use this clause, *table* must support Temporal Validity.

- For *valid_time_column*, specify the name of the valid time dimension column for *table*.
- Use the `BETWEEN` clause to specify the time period during which rows are considered valid. Both expressions must evaluate to a timestamp value and cannot evaluate to NULL. `MINVALUE` resolves to the earliest date or timestamp in the start time column of *table*. `MAXVALUE` resolves to latest date or timestamp in the end time column of *table*.

AS OF PERIOD FOR

Specify `AS OF PERIOD FOR` to retrieve rows from *table* based on whether they are considered valid as of the specified time. In order to use this clause, *table* must support Temporal Validity.

- For *valid_time_column*, specify the name of the valid time dimension column for *table*.
- Use *expr* to specify the time as of which rows are considered valid. The expression must evaluate to a timestamp value and cannot evaluate to NULL.

See Also:

- *Oracle Database Development Guide* for more information on Temporal Validity
- `CREATE TABLE` *period_definition* to learn how to configure a table to support Temporal Validity and for information about the *valid_time_column*, start time column, and end time column

Note on Flashback Queries

When performing a flashback query, Oracle Database might not use query optimizations that it would use for other types of queries, which could have a negative impact on performance. In particular, this occurs when you specify multiple flashback queries in a hierarchical query.

Restrictions on Flashback Queries

These queries are subject to the following restrictions:

- You cannot specify a column expression or a subquery in the expression of the `AS OF` clause.

- You cannot specify the AS OF clause if you have specified the *for_update_clause*.
- You cannot use the AS OF clause in the defining query of a materialized view.
- You cannot use the VERSIONS clause in flashback queries to temporary or external tables, or tables that are part of a cluster.
- You cannot use the VERSIONS clause in flashback queries to views. However, you can use the VERSIONS syntax in the defining query of a view.
- You cannot specify the *flashback_query_clause* if you have specified *query_name* in the *query_table_expression*.

See Also:

- *Oracle Database Development Guide* for more information on Oracle Flashback Query
- "Using Flashback Queries: Example"
- *Oracle Database Development Guide* and *Oracle Database PL/SQL Packages and Types Reference* for information about session-level Flashback using the DBMS_FLASHBACK package
- *Oracle Database Administrator's Guide* and to the description of FLASHBACK_TRANSACTION_QUERY in the *Oracle Database Reference* for more information about transaction history

partition_extension_clause

For PARTITION or SUBPARTITION, specify the name or key value of the partition or subpartition within *table* from which you want to retrieve data.

For range- and list-partitioned data, as an alternative to this clause, you can specify a condition in the WHERE clause that restricts the retrieval to one or more partitions of *table*. Oracle Database will interpret the condition and fetch data from only those partitions. It is not possible to formulate such a WHERE condition for hash-partitioned data.

See Also:

"References to Partitioned Tables and Indexes " and "Selecting from a Partition: Example"

dblink

For *dblink*, specify the complete or partial name for a database link to a remote database where the table, view, or materialized view is located. This database need not be an Oracle Database.

See Also:

- "References to Objects in Remote Databases " for more information on referring to database links
- "Distributed Queries " for more information about distributed queries and "Using Distributed Queries: Example"

If you omit *dblink*, then the database assumes that the table, view, or materialized view is on the local database.

Restrictions on Database Links

Database links are subject to the following restrictions:

- You cannot query a user-defined type or an object REF on a remote table.
- You cannot query columns of type ANYTYPE, ANYDATA, or ANYDATASET from remote tables.

table* | *view* | *materialized_view* | *hierarchy* | *analytic_view

Specify the name of a table, view, materialized view, hierarchy, or analytic view from which data is selected.

sample_clause

The *sample_clause* lets you instruct the database to select from a random sample of data from the table, rather than from the entire table.

See Also:

"Selecting a Sample: Examples"

BLOCK

BLOCK instructs the database to attempt to perform random block sampling instead of random row sampling.

Block sampling is possible only during full table scans or index fast full scans. If a more efficient execution path exists, then Oracle Database does not perform block sampling. If you want to guarantee block sampling for a particular table or index, then use the FULL or INDEX_FFS hint.

Beginning with Oracle Database 12*c* Release 2 (12.2.), you can specify block sampling for external tables. In earlier releases, specifying block sampling for external tables had no effect; row sampling was performed.

sample_percent

For *sample_percent*, specify the percentage of the total row or block count to be included in the sample. The value must be in the range .000001 to, but not including, 100. This percentage indicates the probability of each row, or each cluster of rows in the case of block sampling, being selected as part of the sample. It does not mean that the database will retrieve exactly *sample_percent* of the rows of *table*.

> **WARNING:**
>
> The use of statistically incorrect assumptions when using this feature can lead to incorrect or undesirable results.

SEED *seed_value*

Specify this clause to instruct the database to attempt to return the same sample from one execution to the next. The *seed_value* must be an integer between 0 and 4294967295. If you omit this clause, then the resulting sample will change from one execution to the next.

Restrictions on *sample_clause*

The following restrictions apply to the `SAMPLE` clause:

- You cannot specify the `SAMPLE` clause in a subquery in a DML statement.
- You can specify the `SAMPLE` clause in a query on a base table, a container table of a materialized view, or a view that is key preserving. You cannot specify this clause on a view that is not key preserving.

subquery_restriction_clause

The *subquery_restriction_clause* lets you restrict the subquery in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*n*, where n is an integer that makes the constraint name unique within the database.

> **See Also:**
>
> "Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *table_collection_expression* lets you inform Oracle that the value of *collection_expression* should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the TABLE collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

> **Note:**
>
> In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as THE *subquery*. That usage is now deprecated.

The *collection_expression* can reference columns of tables defined to its left in the FROM clause. This is called **left correlation**. Left correlation can occur only in *table_collection_expression*. Other subqueries cannot contains references to columns defined outside the subquery.

The optional (+) lets you specify that *table_collection_expression* should return a row with all fields set to null if the collection is null or empty. The (+) is valid only if *collection_expression* uses left correlation. The result is similar to that of an outer join.

When you use the (+) syntax in the WHERE clause of a subquery in an UPDATE or DELETE operation, you must specify two tables in the FROM clause of the subquery. Oracle Database ignores the outer join syntax unless there is a join in the subquery itself.

> **See Also:**
>
> - "Outer Joins "
> - "Table Collections: Examples" and "Collection Unnesting: Examples"

t_alias

Specify a **correlation name**, which is an alias for the table, view, materialized view, or subquery for evaluating the query. This alias is required if the select list references any object type attributes or object type methods. Correlation names are most often used in a correlated query. Other references to the table, view, or materialized view throughout the query must refer to this alias.

> **See Also:**
>
> "Using Correlated Subqueries: Examples"

pivot_clause

The `pivot_clause` lets you write cross-tabulation queries that rotate rows into columns, aggregating data in the process of the rotation. The output of a pivot operation typically includes more columns and fewer rows than the starting data set. The `pivot_clause` performs the following steps:

1. The `pivot_clause` computes the aggregation functions specified at the beginning of the clause. Aggregation functions must specify a `GROUP BY` clause to return multiple values, yet the `pivot_clause` does not contain an explicit `GROUP BY` clause. Instead, the `pivot_clause` performs an implicit `GROUP BY`. The implicit grouping is based on all the columns not referred to in the `pivot_clause`, along with the set of values specified in the `pivot_in_clause`.). If you specify more than one aggregation function, then you must provide aliases for at least all but one of the aggregation functions.
2. The grouping columns and aggregated values calculated in Step 1 are configured to produce the following cross-tabular output:
   a. All the implicit grouping columns not referred to in the `pivot_clause`, followed by
   b. New columns corresponding to values in the `pivot_in_clause`. Each aggregated value is transposed to the appropriate new column in the cross-tabulation. If you specify the `XML` keyword, then the result is a single new column that expresses the data as an XML string. The database generates a name for each new column. If you do not provide an alias for an aggregation function, then the database uses each pivot column value as the name for each new column to which that aggregated value is transposed. If you provide an alias for an aggregation function, then the database generates a name for each new column to which that aggregated value is transposed by concatenating the pivot column name, the underscore character (_), and the aggregation function alias. If a generated column name exceeds the maximum length of a column name, then an ORA-00918 error is returned. To avoid this issue, specify a shorter alias for the pivot column heading, the aggregation function, or both.

The subclauses of the `pivot_clause` have the following semantics:

XML

The optional `XML` keyword generates XML output for the query. The `XML` keyword permits the `pivot_in_clause` to contain either a subquery or the wildcard keyword `ANY`. Subqueries and `ANY` wildcards are useful when the `pivot_in_clause` values are not known in advance. With XML output, the values of the pivot column are evaluated at execution time. You cannot specify `XML` when you specify explicit pivot values using expressions in the `pivot_in_clause`.

When XML output is generated, the aggregate function is applied to each distinct pivot value, and the database returns a column of `XMLType` containing an XML string for all value and measure pairs.

expr

For *expr*, specify an expression that evaluates to a constant value of a pivot column. You can optionally provide an alias for each pivot column value. If there is no alias, the column heading becomes a quoted identifier.

subquery

A subquery is used only in conjunction with the XML keyword. When you specify a subquery, all values found by the subquery are used for pivoting. The output is not the same cross-tabular format returned by non-XML pivot queries. Instead of multiple columns specified in the *pivot_in_clause*, the subquery produces a single XML string column. The XML string for each row holds aggregated data corresponding to the implicit GROUP BY value of that row. The XML string for each output row includes all pivot values found by the subquery, even if there are no corresponding rows in the input data.

The subquery must return a list of unique values at the execution time of the pivot query. If the subquery does not return a unique value, then Oracle Database raises a run-time error. Use the DISTINCT keyword in the subquery if you are not sure the query will return unique values.

ANY

The ANY keyword is used only in conjunction with the XML keyword. The ANY keyword acts as a wildcard and is similar in effect to *subquery*. The output is not the same cross-tabular format returned by non-XML pivot queries. Instead of multiple columns specified in the *pivot_in_clause*, the ANY keyword produces a single XML string column. The XML string for each row holds aggregated data corresponding to the implicit GROUP BY value of that row. However, in contrast to the behavior when you specify *subquery*, the ANY wildcard produces an XML string for each output row that includes only the pivot values found in the input data corresponding to that row.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information about PIVOT and UNPIVOT and "Using PIVOT and UNPIVOT: Examples"

unpivot_clause

The *unpivot_clause* rotates columns into rows.

- The INCLUDE | EXCLUDE NULLS clause gives you the option of including or excluding null-valued rows. INCLUDE NULLS causes the unpivot operation to include null-valued rows; EXCLUDE NULLS eliminates null-values rows from the return set. If you omit this clause, then the unpivot operation excludes nulls.
- For *column*, specify a name for each output column that will hold measure values, such as sales_quantity.
- In the *pivot_for_clause*, specify a name for each output column that will hold descriptor values, such as quarter or product.
- In the *unpivot_in_clause*, specify the input data columns whose names will become values in the output columns of the *pivot_for_clause*. These input data columns have names specifying a category value, such as Q1, Q2, Q3, Q4. The optional AS clause lets

you map the input data column names to the specified *literal* values in the output columns.

The unpivot operation turns a set of value columns into one column. Therefore, the data types of all the value columns must be in the same data type group, such as numeric or character.

- If all the value columns are `CHAR`, then the unpivoted column is `CHAR`. If any value column is `VARCHAR2`, then the unpivoted column is `VARCHAR2`.
- If all the value columns are `NUMBER`, then the unpivoted column is `NUMBER`. If any value column is `BINARY_DOUBLE`, then the unpivoted column is `BINARY_DOUBLE`. If no value column is `BINARY_DOUBLE` but any value column is `BINARY_FLOAT`, then the unpivoted column is `BINARY_FLOAT`.

containers_clause

The `CONTAINERS` clause is useful in a multitenant container database (CDB). This clause lets you query data in the specified table or view across all containers in a CDB.

- To query data in a CDB, you must be a common user connected to the CDB root, and the table or view must exist in the root and all PDBs. The query returns all rows from the table or view in the CDB root and in all open PDBs.
- To query data in an application container, you must be a common user connected to the application root, and the table or view must exist in the application root and all PDBs in the application container. The query returns all rows from the table or view in the application root and in all open PDBs in the application container.

The table or view must be in your own schema. It is not necessary to specify *schema*, but if you do then you must specify your own schema.

The query returns all rows from the table or view in the root and in all open PDBs, except PDBs that are open in `RESTRICTED` mode. If the queried table or view does not already contain a `CON_ID` column, then the query adds a `CON_ID` column to the query result, which identifies the container whose data a given row represents.

> **See Also:**
>
> - CONTAINERS Hint
> - *Oracle Database Administrator's Guide* for more information on the `CONTAINERS` clause

join_clause

Use the appropriate *join_clause* syntax to identify tables that are part of a join from which to select data. The *inner_cross_join_clause* lets you specify an inner or cross join. The *outer_join_clause* lets you specify an outer join. The *cross_outer_apply_clause* lets you specify a variation of an ANSI `CROSS JOIN` or an ANSI `LEFT OUTER JOIN` with left correlation support.

When you join more than two row sources, you can use parentheses to override default precedence. For example, the following syntax:

```
SELECT ... FROM a JOIN (b JOIN c) ...
```

results in a join of `b` and `c`, and then a join of that result set with `a`.

> **See Also:**
>
> "Joins " for more information on joins, "Using Join Queries: Examples", "Using Self Joins: Example", and "Using Outer Joins: Examples"

inner_cross_join_clause

Inner joins return only those rows that satisfy the join condition.

INNER

Specify `INNER` to explicitly specify an inner join.

JOIN

The `JOIN` keyword explicitly states that a join is being performed. You can use this syntax to replace the comma-delimited table expressions used in `WHERE` clause joins with `FROM` clause join syntax.

ON *condition*

Use the `ON` clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the `WHERE` clause.

USING (*column*)

When you are specifying an equijoin of columns that have the same name in both tables, the `USING` *`column`* clause indicates the columns to be used. You can use this clause only if the join columns in both tables have the same name. Within this clause, do not qualify the column name with a table name or table alias.

CROSS

The `CROSS` keyword indicates that a cross join is being performed. A cross join produces the cross-product of two relations and is essentially the same as the comma-delimited Oracle Database notation.

NATURAL

The `NATURAL` keyword indicates that a natural join is being performed. Refer to NATURAL for the full semantics of this clause.

outer_join_clause

Outer joins return all rows that satisfy the join condition and also return some or all of those rows from one table for which no rows from the other satisfy the join condition. You can specify two types of outer joins: a conventional outer join using the *`table_reference`* syntax on both sides of the join, or a partitioned outer join using the *`query_partition_clause`* on one side or the other. A partitioned outer join is similar to a conventional outer join except that the join takes place between the outer table and each partition of the inner table. This type of join lets you selectively make sparse data more dense along the dimensions of interest. This process is called **data densification**.

query_partition_clause

The *query_partition_clause* lets you define a **partitioned outer join**. Such a join extends the conventional outer join syntax by applying the outer join to partitions returned by the query. Oracle Database creates a partition of rows for each expression you specify in the PARTITION BY clause. The rows in each query partition have same value for the PARTITION BY expression.

The *query_partition_clause* can be on either side of the outer join. The result of a partitioned outer join is a UNION of the outer joins of each of the partitions in the partitioned result set and the table on the other side of the join. This type of result is useful for filling gaps in sparse data, which simplifies analytic calculations.

If you omit this clause, then the database treats the entire table expression—everything specified in *table_reference*—as a single partition, resulting in a conventional outer join.

To use the *query_partition_clause* in an analytic function, use the upper branch of the syntax (without parentheses). To use this clause in a model query (in the *model_column_clauses*) or a partitioned outer join (in the *outer_join_clause*), use the lower branch of the syntax (with parentheses).

Restrictions on Partitioned Outer Joins

Partitioned outer joins are subject to the following restrictions:

- You can specify the *query_partition_clause* on either the right or left side of the join, but not both.
- You cannot specify a FULL partitioned outer join.
- If you specify the *query_partition_clause* in an outer join with an ON clause, then you cannot specify a subquery in the ON condition.

> **See Also:**
>
> "Using Partitioned Outer Joins: Examples"

NATURAL

The NATURAL keyword indicates that a natural join is being performed. A natural join is based on all columns in the two tables that have the same name. It selects rows from the two tables that have equal values in the relevant columns. If two columns with the same name do not have compatible data types, then an error is raised. When specifying columns that are involved in the natural join, do not qualify the column name with a table name or table alias.

On occasion, the table pairings in natural or cross joins may be ambiguous. For example, consider the following join syntax:

```
a NATURAL LEFT JOIN b LEFT JOIN c ON b.c1 = c.c1
```

This example can be interpreted in either of the following ways:

```
a NATURAL LEFT JOIN (b LEFT JOIN c ON b.c1 = c.c1)
(a NATURAL LEFT JOIN b) LEFT JOIN c ON b.c1 = c.c1
```

To avoid this ambiguity, you can use parentheses to specify the pairings of joined tables. In the absence of such parentheses, the database uses left associativity, pairing the tables from left to right.

Restriction on Natural Joins

You cannot specify a LOB column, columns of `ANYTYPE`, `ANYDATA`, or `ANYDATASET`, or a collection column as part of a natural join.

outer_join_type

The *`outer_join_type`* indicates the kind of outer join being performed:

- Specify `RIGHT` to indicate a right outer join.
- Specify `LEFT` to indicate a left outer join.
- Specify `FULL` to indicate a full or two-sided outer join. In addition to the inner join, rows from both tables that have not been returned in the result of the inner join will be preserved and extended with nulls.
- You can specify the optional `OUTER` keyword following `RIGHT`, `LEFT`, or `FULL` to explicitly clarify that an outer join is being performed.

ON *condition*

Use the `ON` clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the `WHERE` clause.

Restriction on the ON *condition* Clause

You cannot specify this clause with a `NATURAL` outer join.

USING *column*

In an outer join with the `USING` clause, the query returns a single column which is a coalesce of the two matching columns in the join. The coalesce functions as follows:

```
COALESCE (a, b) = a if a NOT NULL, else b.
```

Therefore:

- A left outer join returns all the common column values from the left table in the `FROM` clause.
- A right outer join returns all the common column values from the right table in the `FROM` clause.
- A full outer join returns all the common column values from both joined tables.

Restriction on the USING *column* Clause

The `USING` *`column`* clause is subject to the following restrictions:

- Within this clause, do not qualify the column name with a table name or table alias.
- You cannot specify a LOB column or a collection column in the `USING` *`column`* clause.
- You cannot specify this clause with a `NATURAL` outer join.

> **See Also:**
>
> - "Outer Joins " for additional rules and restrictions pertaining to outer joins
> - *Oracle Database Data Warehousing Guide* for a complete discussion of partitioned outer joins and data densification
> - "Using Outer Joins: Examples"

cross_outer_apply_clause

This clause allows you to perform a variation of an ANSI `CROSS JOIN` or an ANSI `LEFT OUTER JOIN` with left correlation support. You can specify a *`table_reference`* or *`collection_expression`* to the right of the `APPLY` keyword. The *`table_reference`* can be a table, inline view, or `TABLE` collection expression. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. The *`table_reference`* or *`collection_expression`* can reference columns of tables defined in the `FROM` clause to the left of the `APPLY` keyword. This is called left correlation.

- Specify `CROSS APPLY` to perform a variation of an ANSI `CROSS JOIN`. Only rows from the table on the left side of the join that produce a result set from *`table_reference`* or *`collection_expression`* are returned.
- Specify `OUTER APPLY` to perform a variation of an ANSI `LEFT OUTER JOIN`. All rows from the table on the left side of the join are returned. Rows that do not produce a result set from *`table_reference`* or *`collection_expression`* have the NULL value in the corresponding column(s).

Restriction on the *cross_outer_apply_clause*

The *`table_reference`* cannot be a lateral inline view.

> **See Also:**
>
> Using CROSS APPLY and OUTER APPLY Joins: Examples

where_clause

The `WHERE` condition lets you restrict the rows selected to those that satisfy one or more conditions. For *`condition`*, specify any valid SQL condition.

If you omit this clause, then the database returns all rows from the tables, views, or materialized views in the `FROM` clause.

Note:

If this clause refers to a DATE column of a partitioned table or index, then the database performs partition pruning only if:

- You created the table or index partitions by fully specifying the year using the TO_DATE function with a 4-digit format mask, *and*
- You specify the date in the *where_clause* of the query using the TO_DATE function and either a 2- or 4-digit format mask.

See Also:

- Conditions for the syntax description of *condition*
- "Selecting from a Partition: Example"

hierarchical_query_clause

The *hierarchical_query_clause* lets you select rows in a hierarchical order.

SELECT statements that contain hierarchical queries can contain the LEVEL pseudocolumn in the select list. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, and so on. The number of levels returned by a hierarchical query may be limited by available user memory.

Oracle processes hierarchical queries as follows:

- A join, if present, is evaluated first, whether the join is specified in the FROM clause or with WHERE clause predicates.
- The CONNECT BY condition is evaluated.
- Any remaining WHERE clause predicates are evaluated.

If you specify this clause, then do not specify either ORDER BY or GROUP BY, because they will destroy the hierarchical order of the CONNECT BY results. If you want to order rows of siblings of the same parent, then use the ORDER SIBLINGS BY clause.

See Also:

"Hierarchical Queries " for a discussion of hierarchical queries and "Using the LEVEL Pseudocolumn: Examples"

START WITH Clause

Specify a condition that identifies the row(s) to be used as the root(s) of a hierarchical query. The *condition* can be any condition as described in Conditions. Oracle Database uses as root(s) all rows that satisfy this condition. If you omit this clause, then the database uses all rows in the table as root rows.

CONNECT BY Clause

Specify a condition that identifies the relationship between parent rows and child rows of the hierarchy. The *condition* can be any condition as described in Conditions. However, it must use the PRIOR operator to refer to the parent row.

> **See Also:**
>
> - Pseudocolumns for more information on LEVEL
> - "Hierarchical Queries " for general information on hierarchical queries
> - "Hierarchical Query Examples"

group_by_clause

Specify the GROUP BY clause if you want the database to group the selected rows based on the value of *expr*(s) for each row and return a single row of summary information for each group. If this clause contains CUBE or ROLLUP extensions, then the database produces superaggregate groupings in addition to the regular groupings.

Expressions in the GROUP BY clause can contain any columns of the tables, views, or materialized views in the FROM clause, regardless of whether the columns appear in the select list.

The GROUP BY clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for an expanded discussion and examples of using SQL grouping syntax for data aggregation
> - the GROUP_ID , GROUPING , and GROUPING_ID functions for examples
> - "Using the GROUP BY Clause: Examples"
> - Restrictions for Linguistic Collations for information on implications of how GROUP BY character values are compared linguistically
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the expressions in the GROUP BY clause

ROLLUP

The ROLLUP operation in the *simple_grouping_clause* groups the selected rows based on the values of the first n, n-1, n-2, ... 0 expressions in the GROUP BY specification, and returns a single row of summary for each group. You can use the ROLLUP operation to produce **subtotal values** by using it with the SUM function. When used with SUM, ROLLUP generates subtotals from the most detailed level to the grand total. Aggregate functions such as COUNT can be used to produce other kinds of superaggregates.

For example, given three expressions (n=3) in the ROLLUP clause of the *simple_grouping_clause*, the operation results in n+1 = 3+1 = 4 groupings.

Rows grouped on the values of the first *n* expressions are called **regular rows**, and the others are called **superaggregate rows**.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for information on using ROLLUP with materialized views

CUBE

The CUBE operation in the *simple_grouping_clause* groups the selected rows based on the values of all possible combinations of expressions in the specification. It returns a single row of summary information for each group. You can use the CUBE operation to produce **cross-tabulation values**.

For example, given three expressions (n=3) in the CUBE clause of the *simple_grouping_clause*, the operation results in $2^n = 2^3 = 8$ groupings. Rows grouped on the values of *n* expressions are called **regular rows**, and the rest are called **superaggregate rows**.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for information on using CUBE with materialized views
> - "Using the GROUP BY CUBE Clause: Example"

GROUPING SETS

GROUPING SETS are a further extension of the GROUP BY clause that let you specify multiple groupings of data. Doing so facilitates efficient aggregation by pruning the aggregates you do not need. You specify just the desired groups, and the database does not need to perform the full set of aggregations generated by CUBE or ROLLUP. Oracle Database computes all groupings specified in the GROUPING SETS clause and combines the results of individual groupings with a UNION ALL operation. The UNION ALL means that the result set can include duplicate rows.

Within the GROUP BY clause, you can combine expressions in various ways:

- To specify **composite columns**, group columns within parentheses so that the database treats them as a unit while computing ROLLUP or CUBE operations.
- To specify **concatenated grouping sets**, separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the database combines them into a single GROUP BY clause. The result is a cross-product of groupings from each grouping set.

See Also:

"Using the GROUPING SETS Clause: Example"

HAVING Clause

Use the HAVING clause to restrict the groups of returned rows to those groups for which the specified *condition* is TRUE. If you omit this clause, then the database returns summary rows for all groups.

Specify GROUP BY and HAVING after the *where_clause* and *hierarchical_query_clause*. If you specify both GROUP BY and HAVING, then they can appear in either order.

See Also:

"Using the HAVING Condition: Example"

Restrictions on the GROUP BY Clause

This clause is subject to the following restrictions:

- You cannot specify LOB columns, nested tables, or varrays as part of *expr*.
- The expressions can be of any form except scalar subquery expressions.
- If the *group_by_clause* references any object type columns, then the query will not be parallelized.

model_clause

The *model_clause* lets you view selected rows as a multidimensional array and randomly access cells within that array. Using the *model_clause*, you can specify a series of cell assignments, referred to as **rules**, that invoke calculations on individual cells and ranges of cells. These rules operate on the results of a query and do not update any database tables.

When using the *model_clause* in a query, the SELECT and ORDER BY clauses must refer only to those columns defined in the *model_column_clauses*.

See Also:

- The syntax description of *expr* in "About SQL Expressions " and the syntax description of *condition* in Conditions
- *Oracle Database Data Warehousing Guide* for an expanded discussion and examples
- "The MODEL clause: Examples"

main_model

The *main_model* clause defines how the selected rows will be viewed in a multidimensional array and what rules will operate on which cells in that array.

model_column_clauses

The *model_column_clauses* define and classify the columns of a query into three groups: partition columns, dimension columns, and measure columns. For *expr*, you can specify a column, constant, host variable, single-row function, aggregate function, or any expression involving them. If *expr* is a column, then the column alias (*c_alias*) is optional. If *expr* is not a column, then the column alias is required. If you specify a column alias, then you must use the alias to refer to the column in the *model_rules_clause*, SELECT list, and the query ORDER BY clauses.

PARTITION BY

The PARTITION BY clause specifies the columns that will be used to divide the selected rows into partitions based on the values of the specified columns.

DIMENSION BY

The DIMENSION BY clause specifies the columns that will identify a row within a partition. The values of the dimension columns, along with those of the partition columns, serve as array indexes to the measure columns within a row.

MEASURES

The MEASURES clause identifies the columns on which the calculations can be performed. Measure columns in individual rows are treated like cells that you can reference, by specifying the values for the partition and dimension columns, and update.

cell_reference_options

Use the *cell_reference_options* clause to specify how null and absent values are treated in rules and how column uniqueness is constrained.

IGNORE NAV

When you specify IGNORE NAV, the database returns the following values for the null and absent values of the data type specified:

- Zero for numeric data types
- 01-JAN-2000 for datetime data types
- An empty string for character data types
- Null for all other data types

KEEP NAV

When you specify KEEP NAV, the database returns null for both null and absent cell values. KEEP NAV is the default.

UNIQUE SINGLE REFERENCE

When you specify UNIQUE SINGLE REFERENCE, the database checks only single-cell references on the right-hand side of the rule for uniqueness, not the entire query result set.

UNIQUE DIMENSION

When you specify `UNIQUE DIMENSION`, the database checks that the `PARTITION BY` and `DIMENSION BY` columns form a unique key to the query. `UNIQUE DIMENSION` is the default.

model_rules_clause

Use the *`model_rules_clause`* to specify the cells to be updated, the rules for updating those cells, and optionally, how the rules are to be applied and processed.

Each rule represents an assignment and consists of a left-hand side and right-hand side. The left-hand side of the rule identifies the cells to be updated by the right-hand side of the rule. The right-hand side of the rule evaluates to the values to be assigned to the cells specified on the left-hand side of the rule.

UPSERT ALL

`UPSERT ALL` allows `UPSERT` behavior for a rule with both positional and symbolic references on the left-hand side of the rule. When evaluating an `UPSERT ALL` rule, Oracle performs the following steps to create a list of cell references to be upserted:

1. Find the existing cells that satisfy all the symbolic predicates of the cell reference.
2. Using just the dimensions that have symbolic references, find the distinct dimension value combinations of these cells.
3. Perform a cross product of these value combinations with the dimension values specified by way of positional references.

Refer to *Oracle Database Data Warehousing Guide* for more information on the semantics of `UPSERT ALL`.

UPSERT

When you specify `UPSERT`, the database applies the rules to those cells referenced on the left-hand side of the rule that exist in the multidimensional array, and inserts new rows for those that do not exist. `UPSERT` behavior applies only when positional referencing is used on the left-hand side and a single cell is referenced. `UPSERT` is the default. Refer to cell_assignment for more information on positional referencing and single-cell references.

`UPDATE` and `UPSERT` can be specified for individual rules as well. When either `UPDATE` or `UPSERT` is specified for a specific rule, it takes precedence over the option specified in the `RULES` clause.

> **Note:**
>
> If an `UPSERT ALL`, `UPSERT`, or `UPDATE` rule does not contain the appropriate predicates, then the database may implicitly convert it to a different type of rule:
>
> - If an `UPSERT` rule contains an existential predicate, then the rule is treated as an `UPDATE` rule.
> - An `UPSERT ALL` rule must have at least one existential predicate and one qualified predicate on its left side. If it has no existential predicate, then it is treated as an `UPSERT` rule. If it has no qualified predicate, then it is treated as an `UPDATE` rule

UPDATE

When you specify UPDATE, the database applies the rules to those cells referenced on the left-hand side of the rule that exist in the multidimensional array. If the cells do not exist, then the assignment is ignored.

AUTOMATIC ORDER

When you specify AUTOMATIC ORDER, the database evaluates the rules based on their dependency order. In this case, a cell can be assigned a value once only.

SEQUENTIAL ORDER

When you specify SEQUENTIAL ORDER, the database evaluates the rules in the order they appear. In this case, a cell can be assigned a value more than once. SEQUENTIAL ORDER is the default.

ITERATE ... [UNTIL]

Use ITERATE ... [UNTIL] to specify the number of times to cycle through the rules and, optionally, an early termination condition. The parentheses around the UNTIL condition are optional.

When you specify ITERATE ... [UNTIL], rules are evaluated in the order in which they appear. Oracle Database returns an error if both AUTOMATIC ORDER and ITERATE ... [UNTIL] are specified in the *model_rules_clause*.

cell_assignment

The *cell_assignment* clause, which is the left-hand side of the rule, specifies one or more cells to be updated. When a *cell_assignment* references a single cell, it is called a **single-cell reference**. When more than one cell is referenced, it is called a **multiple-cell reference**.

All dimension columns defined in the *model_clause* must be qualified in the *cell_assignment* clause. A dimension can be qualified using either symbolic or positional referencing.

A **symbolic reference** qualifies a single dimension column using a Boolean condition like *dimension_column=constant*. A **positional reference** is one where the dimension column is implied by its position in the DIMENSION BY clause. The only difference between symbolic references and positional references is in the treatment of nulls.

Using a single-cell symbolic reference such as a[x=null,y=2000], no cells qualify because x=null evaluates to FALSE. However, using a single-cell positional reference such as a[null,2000], a cell where x is null and y is 2000 qualifies because null = null evaluates to TRUE. With single-cell positional referencing, you can reference, update, and insert cells where dimension columns are null.

You can specify a condition or an expression representing a dimension column value using either symbolic or positional referencing. *condition* cannot contain aggregate functions or the CV function, and *condition* must reference a single dimension column. *expr* cannot contain a subquery. Refer to "Model Expressions" for information on model expressions.

single_column_for_loop

The *single_column_for_loop* clause lets you specify a range of cells to be updated within a single dimension column.

The IN clause lets you specify the values of the dimension column as either a list of values or as a subquery. When using *subquery*, it cannot:

- Be a correlated query
- Return more than 10,000 rows
- Be a query defined in the WITH clause

The FROM clause lets you specify a range of values for a dimension column with discrete increments within the range. The FROM clause can only be used for those columns with a data type for which addition and subtraction is supported. The INCREMENT and DECREMENT values must be positive.

Optionally, you can specify the LIKE clause within the FROM clause. In the LIKE clause, *pattern* is a character string containing a single pattern-matching character %. This character is replaced during execution with the current incremented or decremented value in the FROM clause.

If all dimensions other than those used by a FOR loop involve a single-cell reference, then the expressions can insert new rows. The number of dimension value combinations generated by FOR loops is counted as part of the 10,000 row limit of the MODEL clause.

multi_column_for_loop

The *multi_column_for_loop* clause lets you specify a range of cells to be updated across multiple dimension columns. The IN clause lets you specify the values of the dimension columns as either multiple lists of values or as a subquery. When using *subquery*, it cannot:

- Be a correlated query
- Return more than 10,000 rows
- Be a query defined in the WITH clause

If all dimensions other than those used by a FOR loop involve a single-cell reference, then the expressions can insert new rows. The number of dimension value combinations generated by FOR loops is counted as part of the 10,000 row limit of the MODEL clause.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information about using FOR loops in the MODEL clause

order_by_clause

Use the ORDER BY clause to specify the order in which cells on the left-hand side of the rule are to be evaluated. The *expr* must resolve to a dimension or measure column. If the ORDER BY clause is not specified, then the order defaults to the order of the columns as specified in the DIMENSION BY clause. See *order_by_clause* for more information.

Restrictions on the *order_by_clause*

Use of the ORDER BY clause in the model rule is subject to the following restrictions:

- You cannot specify SIBLINGS, *position*, or *c_alias* in the *order_by_clause* of the *model_clause*.

- You cannot specify this clause on the left-hand side of the model rule and also specify a `FOR` loop on the right-hand side of the rule.

expr

Specify an expression representing the value or values of the cell or cells specified on the right-hand side of the rule. *`expr`* cannot contain a subquery. Refer to "Model Expressions" for information on model expressions.

return_rows_clause

The *`return_rows_clause`* lets you specify whether to return all rows selected or only those rows updated by the model rules. `ALL` is the default.

reference_model

Use the *`reference_model`* clause when you need to access multiple arrays from inside the *`model_clause`*. This clause defines a read-only multidimensional array based on the results of a query.

The subclauses of the *`reference_model`* clause have the same semantics as for the *`main_model`* clause. Refer to model_column_clauses and cell_reference_options.

Restrictions on the *reference_model* Clause

This clause is subject to the following restrictions:

- `PARTITION BY` columns cannot be specified for reference models.
- The subquery of the reference model cannot refer to columns in an outer subquery.

Set Operators: UNION, UNION ALL, INTERSECT, MINUS

The set operators combine the rows returned by two `SELECT` statements into a single result. The number and data types of the columns selected by each component query must be the same, but the column lengths can be different. The names of the columns in the result set are the names of the expressions in the select list preceding the set operator.

If you combine more than two queries with set operators, then the database evaluates adjacent queries from left to right. The parentheses around the subquery are optional. You can use them to specify a different order of evaluation.

Refer to "The UNION [ALL], INTERSECT, MINUS Operators" for information on these operators, including restrictions on their use.

order_by_clause

Use the `ORDER BY` clause to order rows returned by the statement. Without an *`order_by_clause`*, no guarantee exists that the same query executed more than once will retrieve rows in the same order.

SIBLINGS

The `SIBLINGS` keyword is valid only if you also specify the *`hierarchical_query_clause`* (`CONNECT BY`). `ORDER SIBLINGS BY` preserves any ordering specified in the hierarchical query clause and then applies the *`order_by_clause`* to the siblings of the hierarchy.

expr

`expr` orders rows based on their value for *`expr`*. The expression is based on columns in the select list or columns in the tables, views, or materialized views in the `FROM` clause.

position

Specify *position* to order rows based on their value for the expression in this position of the select list. The *position* value must be an integer.

You can specify multiple expressions in the *order_by_clause*. Oracle Database first sorts rows based on their values for the first expression. Rows with the same value for the first expression are then sorted based on their values for the second expression, and so on. The database sorts nulls following all others in ascending order and preceding all others in descending order. Refer to "Sorting Query Results " for a discussion of ordering query results.

ASC | DESC

Specify whether the ordering sequence is ascending or descending. ASC is the default.

NULLS FIRST | NULLS LAST

Specify whether returned rows containing null values should appear first or last in the ordering sequence.

NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.

Restrictions on the ORDER BY Clause

The following restrictions apply to the ORDER BY clause:

- If you have specified the DISTINCT operator in this statement, then this clause cannot refer to columns unless they appear in the select list.
- An *order_by_clause* can contain no more than 255 expressions.
- You cannot order by a LOB, LONG, or LONG RAW column, nested table, or varray.
- If you specify a *group_by_clause* in the same statement, then this *order_by_clause* is restricted to the following expressions:
  - Constants
  - Aggregate functions
  - Analytic functions
  - The functions USER, UID, and SYSDATE
  - Expressions identical to those in the *group_by_clause*
  - Expressions comprising the preceding expressions that evaluate to the same value for all rows in a group

See Also:

- "Using the ORDER BY Clause: Examples"
- Restrictions for Linguistic Collations for information on implications of how ORDER BY character values are compared linguistically
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the expressions in the ORDER BY clause

row_limiting_clause

The *row_limiting_clause* allows you to limit the rows returned by the query. You can specify an offset, and the number of rows or percentage of rows to return. You can use this clause to implement top-N reporting. For consistent results, specify the *order_by_clause* to ensure a deterministic sort order.

OFFSET

Use this clause to specify the number of rows to skip before row limiting begins. *offset* must be a number or an expression that evaluates to a numeric value. If you specify a negative number, then *offset* is treated as 0. If you specify NULL, or a number greater than or equal to the number of rows returned by the query, then 0 rows are returned. If *offset* includes a fraction, then the fractional portion is truncated. If you do not specify this clause, then *offset* is 0 and row limiting begins with the first row.

ROW | ROWS

These keywords can be used interchangeably and are provided for semantic clarity.

FETCH

Use this clause to specify the number of rows or percentage of rows to return. If you do not specify this clause, then all rows are returned, beginning at row *offset* + 1.

FIRST | NEXT

These keywords can be used interchangeably and are provided for semantic clarity.

***rowcount* | *percent* PERCENT**

Use *rowcount* to specify the number of rows to return. *rowcount* must be a number or an expression that evaluates to a numeric value. If you specify a negative number, then *rowcount* is treated as 0. If *rowcount* is greater than the number of rows available beginning at row *offset* + 1, then all available rows are returned. If *rowcount* includes a fraction, then the fractional portion is truncated. If *rowcount* is NULL, then 0 rows are returned.

Use *percent* PERCENT to specify the percentage of the total number of selected rows to return. *percent* must be a number or an expression that evaluates to a numeric value. If you specify a negative number, then *percent* is treated as 0. If *percent* is NULL, then 0 rows are returned.

If you do not specify *rowcount* or *percent* PERCENT, then 1 row is returned.

ROW | ROWS

These keywords can be used interchangeably and are provided for semantic clarity.

ONLY | WITH TIES

Specify ONLY to return exactly the specified number of rows or percentage of rows.

Specify WITH TIES to return additional rows with the same sort key as the last row fetched. If you specify WITH TIES, then you must specify the *order_by_clause*. If you do not specify the *order_by_clause*, then no additional rows will be returned.

Restrictions on the *row_limiting_clause*

This clause is subject to the following restrictions:

- You cannot specify this clause with the *for_update_clause*.
- If you specify this clause, then the select list cannot contain the sequence pseudocolumns CURRVAL or NEXTVAL.
- Materialized views are not eligible for an incremental refresh if the defining query contains the *row_limiting_clause*.
- If the select list contains columns with identical names and you specify the *row_limiting_clause*, then an ORA-00918 error occurs. This error occurs whether the identically named columns are in the same table or in different tables. You can work around this issue by specifying unique column aliases for the identically named columns.

> **See Also:**
>
> "Row Limiting: Examples"

for_update_clause

The FOR UPDATE clause lets you lock the selected rows so that other users cannot lock or update the rows until you end your transaction. You can specify this clause only in a top-level SELECT statement, not in subqueries.

> **Note:**
>
> Prior to updating a LOB value, you must lock the row containing the LOB. One way to lock the row is with an embedded SELECT ... FOR UPDATE statement. You can do this using one of the programmatic languages or DBMS_LOB package. For more information on lock rows before writing to a LOB, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Nested table rows are not locked as a result of locking the parent table rows. If you want the nested table rows to be locked, then you must lock them explicitly.

Restrictions on the FOR UPDATE Clause

This clause is subject to the following restrictions:

- You cannot specify this clause with the following other constructs: the DISTINCT operator, CURSOR expression, set operators, *group_by_clause*, or aggregate functions.
- The tables locked by this clause must all be located on the same database and on the same database as any LONG columns and sequences referenced in the same statement.

See Also:

"Using the FOR UPDATE Clause: Examples"

Using the FOR UPDATE Clause on Views

In general, this clause is not supported on views. However, in some cases, a `SELECT ... FOR UPDATE` query on a view can succeed without any errors. This occurs when the view has been merged to its containing query block internally by the query optimizer, and `SELECT ... FOR UPDATE` succeeds on the internally transformed query. The examples in this section illustrate when using the `FOR UPDATE` clause on a view can succeed or fail.

- Using the `FOR UPDATE` clause on merged views

  An error can occur when you use the `FOR UPDATE` clause on a merged view if both of the following conditions apply:

  – The underlying column of the view is an expression

  – The `FOR UPDATE` clause applies to a column list

  The following statement succeeds because the underlying column of the view is not an expression:

```
SELECT employee_id FROM (SELECT * FROM employees)
   FOR UPDATE OF employee_id;
```

  The following statement succeeds because, while the underlying column of the view is an expression, the `FOR UPDATE` clause does not apply to a column list:

```
SELECT employee_id FROM (SELECT employee_id+1 AS employee_id FROM employees)
   FOR UPDATE;
```

  The following statement fails because the underlying column of the view is an expression and the `FOR UPDATE` clause applies to a column list:

```
SELECT employee_id FROM (SELECT employee_id+1 AS employee_id FROM employees)
   FOR UPDATE OF employee_id;
                 *
Error at line 2:
ORA-01733: virtual column not allowed here
```

- Using the `FOR UPDATE` clause on non-merged views

  Since the `FOR UPDATE` clause is not supported on views, anything that prevents view merging, such as the `NO_MERGE` hint, parameters that disallow view merging, or something in the query structure that prevents view merging, will result in an `ORA-02014` error.

  In the following example, the `GROUP BY` statement prevents view merging, which causes an error:

```
SELECT avgsal
   FROM (SELECT AVG(salary) AS avgsal FROM employees GROUP BY job_id)
   FOR UPDATE;
FROM (SELECT AVG(salary) AS avgsal FROM employees GROUP BY job_id)
     *
ERROR at line 2:
ORA-02014: cannot select FOR UPDATE from view with DISTINCT, GROUP BY, etc.
```

> **Note:**
>
> Due to the complexity of the view merging mechanism, Oracle recommends against using the `FOR UPDATE` clause on views.

OF ... *column*

Use the `OF` ... *`column`* clause to lock the select rows only for a particular table or view in a join. The columns in the `OF` clause only indicate which table or view rows are locked. The specific columns that you specify are not significant. However, you must specify an actual column name, not a column alias. If you omit this clause, then the database locks the selected rows from all the tables in the query.

NOWAIT | WAIT

The `NOWAIT` and `WAIT` clauses let you tell the database how to proceed if the `SELECT` statement attempts to lock a row that is locked by another user.

- Specify `NOWAIT` to return control to you immediately if a lock exists.
- Specify `WAIT` to instruct the database to wait *`integer`* seconds for the row to become available and then return control to you.

If you specify neither `WAIT` nor `NOWAIT`, then the database waits until the row is available and then returns the results of the `SELECT` statement.

SKIP LOCKED

`SKIP LOCKED` is an alternative way to handle a contending transaction that is locking some rows of interest. Specify `SKIP LOCKED` to instruct the database to attempt to lock the rows specified by the `WHERE` clause and to skip any rows that are found to be already locked by another transaction. This feature is designed for use in multiconsumer queue environments, such as Oracle Streams Advanced Queuing. It enables queue consumers to skip rows that are locked by other consumers and obtain unlocked rows without waiting for the other consumers to finish. Oracle recommends that you use the Oracle Streams Advanced Queuing APIs instead of directly using the `SKIP LOCKED` functionality. Refer to *Oracle Database Advanced Queuing User's Guide* for more information.

Note on the WAIT and SKIP LOCKED Clauses

If you specify `WAIT` or `SKIP LOCKED` and the table is locked in exclusive mode, then the database will not return the results of the `SELECT` statement until the lock on the table is released. In the case of `WAIT`, the `SELECT FOR UPDATE` clause is blocked regardless of the wait time specified.

row_pattern_clause

The `MATCH_RECOGNIZE` clause lets you perform pattern matching. Use this clause to recognize patterns in a sequence of rows in *`table`*, which is called the row pattern input table. The result of a query that uses the `MATCH_RECOGNIZE` clause is called the row pattern output table.

The `MATCH_RECOGNIZE` enables you to do the following tasks:

- Logically partition and order the data with the `PARTITION BY` and `ORDER BY` clauses.

- Define measures, which are expressions usable in other parts of the SQL query, in the `MEASURES` clause.
- Define patterns of rows to seek using the `PATTERN` clause. These patterns use regular expression syntax, a powerful and expressive feature, applied to the pattern variables you define.
- Specify the logical conditions required to map a row to a row pattern variable in the `DEFINE` clause.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for more information on pattern matching
> - "Row Pattern Matching: Example"

row_pattern_partition_by

Specify `PARTITION BY` to divide the rows in the row pattern input table into logical groups called row pattern partitions. Use *`column`* to specify one or more partitioning columns. Each partition consists of the set of rows in the row pattern input table that have the same value(s) on the partitioning column(s).

If you specify this clause, then matches are found within partitions and do not cross partition boundaries. If you do not specify this clause, then all rows of the row input table constitute a single row pattern partition.

row_pattern_order_by

Specify `ORDER BY` to order rows within each row pattern partition. Use *`column`* to specify one or more ordering columns. If you specify multiple columns, then Oracle Database first sorts rows based on their values for the first column. Rows with the same value for the first column are then sorted based on their values for the second column, and so on. Oracle Database sorts nulls following all others in ascending order.

If you do not specify this clause, then the result of the *`row_pattern_clause`* is nondeterministic and you may get inconsistent results each time you run the query.

row_pattern_measures

Use the `MEASURES` clause to define one or more row pattern measure columns. These columns are included in the row pattern output table and contain values that are useful for analyzing data.

When you define a row pattern measure column, using the *`row_pattern_measure_column`* clause, you specify its pattern measure expression. The values in the column are calculated by evaluating the pattern measure expression whenever a match is found.

row_pattern_measure_column

Use this clause to define a row pattern measure column.

- For *`expr`*, specify the pattern measure expression. A pattern measure expression is an expression as described in Expressions that can contain only the following elements:

  - Constants: Text literals and numeric literals
  - References to any column of the row pattern input table
  - The CLASSIFIER function, which returns the name of the primary row pattern variable to which the row is mapped. Refer to row_pattern_classifier_func for more information.
  - The MATCH_NUMBER function, which returns the sequential number of a row pattern match within the row pattern partition. Refer to row_pattern_match_num_func for more information.
  - Row pattern navigation functions: PREV, NEXT, FIRST, and LAST. Refer to row_pattern_navigation_func for more information.
  - Row pattern aggregate functions: AVG , COUNT , MAX , MIN , or SUM . Refer to row_pattern_aggregate_func for more information.
- For *c_alias*, specify the alias for the pattern measure expression. Oracle Database uses this alias in the column heading of the row pattern output table. The AS keyword is optional. The alias can be used in other parts of the query, such as the SELECT ... ORDER BY clause.

row_pattern_rows_per_match

This clause lets you specify whether the row pattern output table includes summary or detailed data about each match.

- If you specify ONE ROW PER MATCH, then each match produces one summary row. This is the default.
- If you specify ALL ROWS PER MATCH, then each match that spans multiple rows will produce one output row for each row in the match.

row_pattern_skip_to

This clause lets you specify the point to resume row pattern matching after a non-empty match is found.

- Specify AFTER MATCH SKIP TO NEXT ROW to resume pattern matching at the row after the first row of the current match.
- Specify AFTER MATCH SKIP PAST LAST ROW to resume pattern matching at the next row after the last row of the current match. This is the default.
- Specify AFTER MATCH SKIP TO FIRST *variable_name* to resume pattern matching at the first row that is mapped to pattern variable *variable_name*. The *variable_name* must be defined in the DEFINE clause.
- Specify AFTER MATCH SKIP TO LAST *variable_name* to resume pattern matching at the last row that is mapped to pattern variable *variable_name*. The *variable_name* must be defined in the DEFINE clause.
- AFTER MATCH SKIP TO *variable_name* has the same behavior as AFTER MATCH SKIP TO LAST *variable_name*.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on the `AFTER MATCH SKIP` clauses

PATTERN

Use the `PATTERN` clause to define which pattern variables must be matched, the sequence in which they must be matched, and the quantity of rows that must be matched for each pattern variable.

A row pattern match consists of a set of contiguous rows in a row pattern partition. Each row of the match is mapped to a pattern variable. The mapping of rows to pattern variables must conform to the regular expression specified in the *`row_pattern`* clause, and all conditions in the `DEFINE` clause must be true.

> **Note:**
>
> It is outside the scope of this document to explain regular expression concepts and details. If you are not familiar with regular expressions, then you are encouraged to familiarize yourself with the topic using other sources.

The precedence of the elements that you specify in the regular expression of the `PATTERNS` clause, in decreasing order, is as follows:

- Row pattern elements (specified in the *`row_pattern_primary`* clause)
- Row pattern quantifiers (specified in the *`row_pattern_quantifier`* clause)
- Concatenation (specified in the *`row_pattern_term`* clause)
- Alternation (specified in the *`row_pattern`* clause)

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on the `PATTERN` clause

row_pattern

Use this clause to specify the row pattern. A row pattern is a regular expression that can take one of the following forms:

- A single row pattern term

  For example: `PATTERN (A)`
- A row pattern, a vertical bar, and a row pattern term

  For example: `PATTERN (A|B)`
- A recursively built row pattern, a vertical bar, and a row pattern term

For example: `PATTERN (A|B|C)`

The vertical bar in this clause represents **alternation**. Alternation matches a single regular expression from a list of several possible regular expressions. Alternatives are preferred in the order they are specified. For example, if you specify `PATTERN (A|B|C)`, then Oracle Database attempts to match `A` first. If `A` is not matched, then it attempts to match `B`. If `B` is not matched, then it attempts to match `C`.

row_pattern_term

This clause lets you specify a row pattern term. A row pattern term can take one of the following forms:

- A single row pattern factor

  For example: `PATTERN (A)`
- A row pattern term followed by a row pattern factor.

  For example: `PATTERN (A B)`
- A recursively built row pattern term followed by a row pattern factor

  For example: `PATTERN (A B C)`

The syntax used in the second and third examples represents **concatenation**. Concatenation is used to list two or more items in a pattern to be matched and the order in which they are to be matched. For example, if you specify `PATTERN (A B C)`, then Oracle Database first matches `A`, then uses the resulting matched rows to match `B`, then uses the resulting matched rows to match `C`. Only rows that match `A`, `B`, and `C`, are included in the row pattern match.

row_pattern_factor

This clause lets you specify a row pattern factor. A row pattern factor consists of a row pattern element, specified using the *`row_pattern_primary`* clause, and an optional row pattern quantifier, specified using the *`row_pattern_quantifier`* clause.

row_pattern_primary

Use this clause to specify the row pattern element. Table 19-1 lists the valid row pattern elements and their descriptions.

Table 19-1 Row Pattern Elements

Row Pattern Element	Description
`variable_name`	Specify a primary pattern variable name that is defined in the *`row_pattern_definition`* clause. You cannot specify a union pattern variable that is defined in the *`row_pattern_subset_item`* clause.
`$`	`$` matches the position after the last row in the partition. This element is an anchor. Anchors work in terms of positions rather than rows.
`^`	`^` matches the position before the first row in the partition. This element is an anchor. Anchors work in terms of positions rather than rows
`( [`*`row_pattern`*`] )`	Use *`row_pattern`* to specify the row pattern to be matched. An empty pattern `()` matches an empty set of rows.

Table 19-1 (Cont.) Row Pattern Elements

Row Pattern Element	Description
{- *row_pattern* -}	Exclusion syntax. Use *row_pattern* to specify parts of the pattern to be excluded from the output of ALL ROWS PER MATCH.
row_pattern_permute	Use *row_pattern_permute* to specify a pattern that is a permutation of row pattern elements. Refer to row_pattern_permute for the full semantics of this clause.

row_pattern_permute

Use the PERMUTE clause to express a pattern that is a permutation of the specified row pattern elements. For example, PATTERN (PERMUTE (A, B, C)) is equivalent to an alternation of all permutations of the three row pattern elements A, B, and C, similar to the following:

```
PATTERN (A B C | A C B | B A C | B C A | C A B | C B A)
```

Note that the row pattern elements are expanded lexicographically and that each element to permute must be separated by a comma from the other elements.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on permutations

row_pattern_quantifier

Use this clause to specify the row pattern quantifier, which is a postfix operator that defines the number of iterations accepted for a match.

Row pattern quantifiers are referred to as greedy; they will attempt to match as many instances of the regular expression on which they are applied as possible. The exception is row pattern quantifiers that have a question mark (?) as a suffix, which are referred to as reluctant. They will attempt to match as few instances as possible of the regular expression on which they are applied.

Table 19-2 lists the valid row pattern quantifiers and the number of iterations they accept for a match. In this table, *n* and *m* represent unsigned integers.

Table 19-2 Row Pattern Quantifiers

Row Pattern Quantifier	Number of Iterations Accepted for a Match
*	0 or more iterations (greedy)
*?	0 or more iterations (reluctant)
+	1 or more iterations (greedy)
+?	1 or more iterations (reluctant)
?	0 or 1 iterations (greedy)
??	0 or 1 iterations (reluctant)

Table 19-2 (Cont.) Row Pattern Quantifiers

Row Pattern Quantifier	Number of Iterations Accepted for a Match
`{n,}`	*n* or more iterations, (*n* >= 0) (greedy)
`{n,}?`	*n* or more iterations, (*n* >= 0) (reluctant)
`{n,m}`	Between *n* and *m* iterations, inclusive, (0 <= *n* <= *m*, 0 < *m*) (greedy)
`{n,m}?`	Between *n* and *m* iterations, inclusive, (0 <= *n* <= *m*, 0 < *m*) (reluctant)
`{,m}`	Between 0 and *m* iterations, inclusive (*m* > 0) (greedy)
`{,m}?`	Between 0 and *m* iterations, inclusive (*m* > 0) (reluctant)
`{n}?`	*n* iterations, (*n* > 0)

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on row pattern quantifiers

row_pattern_subset_clause

The `SUBSET` clause lets you specify one or more union row pattern variables. Use the *`row_pattern_subset_item`* clause to declare each union row pattern variable.

You can specify union row pattern variables in the following clauses:

- `MEASURES` clause: In the expression for a row pattern measure column. That is, in expression *`expr`* of the *`row_pattern_measure_column`* clause.
- `DEFINE` clause: In the condition that defines a primary pattern variable. That is, in *`condition`* of the *`row_pattern_definition`* clause

row_pattern_subset_item

This clause lets you create a grouping of multiple pattern variables that can be referred to with a variable name of its own. The variable name that refers to this grouping is called a union row pattern variable.

- For *`variable_name`* on the left side of the equal sign, specify the name of the union row pattern variable.
- On the right side of the equal sign, specify a comma-separated list of distinct primary row pattern variables within parentheses. This list cannot include any union row pattern variables.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information on defining union row pattern variables

DEFINE

Use the `DEFINE` clause to specify one or more row pattern definitions. A row pattern definition specifies the conditions that a row must meet in order to be mapped to a specific pattern variable.

The `DEFINE` clause only supports running semantics.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for more information on the `DEFINE` clause
> - *Oracle Database Data Warehousing Guide* for more information on running and final semantics

row_pattern_definition_list

This clause lets you specify one or more row pattern definitions.

row_pattern_definition

This clause lets you specify a row pattern definition, which contains the conditions that a row must meet in order to be mapped to the specified pattern variable.

- For *`variable_name`*, specify the name of the pattern variable.
- For *`condition`*, specify a condition as described in Conditions, with the following extension: *`condition`* can contain any of the functions described by *row_pattern_navigation_func*::= and *row_pattern_aggregate_func*::=.

row_pattern_rec_func

This clause comprises the following clauses, which let you specify row pattern recognition functions:

- *`row_pattern_classifier_func`*: Use this clause to specify the `CLASSIFIER` function, which returns a character string whose value is the name of the variable to which the row is mapped.
- *`row_pattern_match_num_func`*: Use this clause to specify the `MATCH_NUMBER` function, which returns a numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.
- *`row_pattern_navigation_func`*: Use this clause to specify functions that perform row pattern navigation operations.
- *`row_pattern_aggregate_func`*: Use this clause to specify an aggregate function in the expression for a row pattern measure column or in the condition that defines a primary pattern variable.

You can specify row pattern recognition functions in the following clauses:

- `MEASURES` clause: In the expression for a row pattern measure column. That is, in expression *`expr`* of the *`row_pattern_measure_column`* clause.

- `DEFINE` clause: In the condition that defines a primary pattern variable. That is, in *`condition`* of the *`row_pattern_definition`* clause

A row pattern recognition function may behave differently depending whether you specify it in the `MEASURES` or `DEFINE` clause. These details are explained in the semantics for each clause.

row_pattern_classifier_func

The `CLASSIFIER` function returns a character string whose value is the name of the variable to which the row is mapped.

- In the `MEASURES` clause:
  - If you specify `ONE ROW PER MATCH`, then the query uses the last row of the match when processing the `MEASURES` clause, so the `CLASSIFIER` function returns the name of the pattern variable to which the last row of the match is mapped.
  - If you specify `ALL ROWS PER MATCH`, then for each row of the match found, the `CLASSIFIER` function returns the name of the pattern variable to which the row is mapped.

  For empty matches—that is, matches that contain no rows, the `CLASSIFER` function returns NULL.
- In the `DEFINE` clause, the `CLASSIFIER` function returns the name of the primary pattern variable to which the current row is mapped.

row_pattern_match_num_func

The `MATCH_NUMBER` function returns a numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.

Matches within a row pattern partition are numbered sequentially starting with 1 in the order in which they are found. If multiple rows satisfy a match, then they are all assigned the same match number. Note that match numbering starts over again at 1 in each row pattern partition, because there is no inherent ordering between row pattern partitions.

- In the `MEASURES` clause: You can use `MATCH_NUMBER` to obtain the sequential number of the match within the row pattern.
- In the `DEFINE` clause: You can use `MATCH_NUMBER` to define conditions that depend upon the match number.

row_pattern_navigation_func

This clause lets you perform the following row pattern navigation operations:

- Navigate among the group of rows mapped to a pattern variable using the `FIRST` and `LAST` functions of the *`row_pattern_nav_logical`* clause.
- Navigate among all rows in a row pattern partition using the `PREV` and `NEXT` functions of the *`row_pattern_nav_physical`* clause
- Nest the `FIRST` or `LAST` function within the `PREV` or `NEXT` function using the *`row_pattern_nav_compound`* clause.

row_pattern_nav_logical

This clause lets you use the `FIRST` and `LAST` functions to navigate among the group of rows mapped to a pattern variable using an optional logical offset.

- The `FIRST` function returns the value of expression *`expr`* when evaluated in the first row of the group of rows mapped to the pattern variable that is specified in *`expr`*. If no rows are mapped to the pattern variable, then the `FIRST` function returns NULL.
- The `LAST` function returns the value of expression *`expr`* when evaluated in the last row of the group of rows mapped to the pattern variable that is specified in *`expr`*. If no rows are mapped to the pattern variable, then the `LAST` function returns NULL.
- Use *`expr`* to specify the expression to be evaluated. It must contain at least one row pattern column reference. If it contains more than one row pattern column reference, then all must refer to the same pattern variable.
- Use the optional *`offset`* to specify the logical offset within the set of rows mapped to the pattern variable. When specified with the `FIRST` function, the offset is the number of rows from the first row, in ascending order. When specified with the `LAST` function, the offset is the number of rows from the last row in descending order. The default offset is 0.

  For *`offset`*, specify a non-negative integer. It must be a runtime constant (literal, bind variable, or expressions involving them), but not a column or subquery.

  If you specify an *`offset`* that is greater than or equal to the number of rows mapped to the pattern variable minus 1, then the function returns NULL.

You can specify running or final semantics for the `FIRST` and `LAST` functions as follows:

- The `MEASURES` clause supports running and final semantics. Specify `RUNNING` for running semantics. Specify `FINAL` for final semantics. The default is `RUNNING`.
- The `DEFINE` clause supports only running semantics. Therefore, running semantics will be used whether you specify or omit `RUNNING`. You cannot specify `FINAL`.

> **See Also:**
>
> – *Oracle Database Data Warehousing Guide* for more information on the `FIRST` and `LAST` functions
>
> – *Oracle Database Data Warehousing Guide* for more information on running and final semantics

row_pattern_nav_physical

This clause lets you use the `PREV` and `NEXT` functions to navigate all rows in a row pattern partition using an optional physical offset.

- The `PREV` function returns the value of expression *`expr`* when evaluated in the previous row in the partition. If there is no previous row in the partition, then the `PREV` function returns NULL.
- The `NEXT` function returns the value of expression *`expr`* when evaluated in the next row in the partition. If there is no next row in the partition, then the NEXT function returns NULL.
- Use *`expr`* to specify the expression to be evaluated. It must contain at least one row pattern column reference. If it contains more than one row pattern column reference, then all must refer to the same pattern variable.
- Use the optional *`offset`* to specify the physical offset within the partition. When specified with the `PREV` function, it is the number of rows before the current row. When specified

with the NEXT function, it is the number of rows after the current row. The default is 1. If you specify an offset of 0, then the current row is evaluated.

For *offset*, specify a non-negative integer. It must be a runtime constant (literal, bind variable, or expressions involving them), but not a column or subquery.

The PREV and NEXT functions always use running semantics. Therefore, you cannot specify the RUNNING or FINAL keywords with this clause.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on the PREV and NEXT functions
- *Oracle Database Data Warehousing Guide* for more information on running and final semantics

row_pattern_nav_compound

This clause lets you nest the *row_pattern_nav_logical* clause within the *row_pattern_nav_physical* clause. That is, it lets you nest the FIRST or LAST function within the PREV or NEXT function. The *row_pattern_nav_logical* clause is evaluated first and then the result is supplied to the *row_pattern_nav_physical* clause.

Refer to row_pattern_nav_logical and row_pattern_nav_physical for the full semantics of these clauses.

See Also:

Oracle Database Data Warehousing Guide for more information on nesting the FIRST and LAST functions within the PREV and NEXT functions

row_pattern_aggregate_func

This clause lets you use an aggregate function in the expression for a row pattern measure column or in the condition that defines a primary pattern variable.

For *aggregate_function*, specify any one of the AVG , COUNT , MAX , MIN , or SUM functions. The DISTINCT keyword is not supported.

You can specify running or final semantics for aggregate functions as follows:

- The MEASURES clause supports running and final semantics. Specify RUNNING for running semantics. Specify FINAL for final semantics. The default is RUNNING.
- The DEFINE clause supports only running semantics. Therefore, running semantics will be used whether you specify or omit RUNNING. You cannot specify FINAL.

See Also:

- *Oracle Database Data Warehousing Guide* for more information on aggregate functions
- *Oracle Database Data Warehousing Guide* for more information on running and final semantics

Examples

Using a PL/SQL Function in the WITH Clause: Examples

The following example declares and defines a PL/SQL function `get_domain` in the `WITH` clause. The `get_domain` function returns the domain name from a URL string, assuming that the URL string has the "`www`" prefix immediately preceding the domain name, and the domain name is separated by dots on the left and right. The `SELECT` statement uses `get_domain` to find distinct catalog domain names from the `orders` table in the `oe` schema.

```
WITH
 FUNCTION get_domain(url VARCHAR2) RETURN VARCHAR2 IS
   pos BINARY_INTEGER;
   len BINARY_INTEGER;
 BEGIN
   pos := INSTR(url, 'www.');
   len := INSTR(SUBSTR(url, pos + 4), '.') - 1;
   RETURN SUBSTR(url, pos + 4, len);
 END;
SELECT DISTINCT get_domain(catalog_url)
  FROM product_information;
/
```

Subquery Factoring: Example

The following statement creates the query names `dept_costs` and `avg_cost` for the initial query block containing a join, and then uses the query names in the body of the main query.

```
WITH
   dept_costs AS (
      SELECT department_name, SUM(salary) dept_total
         FROM employees e, departments d
         WHERE e.department_id = d.department_id
      GROUP BY department_name),
   avg_cost AS (
      SELECT SUM(dept_total)/COUNT(*) avg
      FROM dept_costs)
SELECT * FROM dept_costs
   WHERE dept_total >
      (SELECT avg FROM avg_cost)
      ORDER BY department_name;

DEPARTMENT_NAME                DEPT_TOTAL
------------------------------ ----------
Sales                              304500
Shipping                           156400
```

Recursive Subquery Factoring: Examples

The following statement shows the employees who directly or indirectly report to employee 101 and their reporting level.

```
WITH
  reports_to_101 (eid, emp_last, mgr_id, reportLevel) AS
  (
     SELECT employee_id, last_name, manager_id, 0 reportLevel
     FROM employees
     WHERE employee_id = 101
   UNION ALL
     SELECT e.employee_id, e.last_name, e.manager_id, reportLevel+1
     FROM reports_to_101 r, employees e
     WHERE r.eid = e.manager_id
  )
SELECT eid, emp_last, mgr_id, reportLevel
FROM reports_to_101
ORDER BY reportLevel, eid;

       EID EMP_LAST                      MGR_ID REPORTLEVEL
---------- ------------------------- ---------- -----------
       101 Kochhar                          100           0
       108 Greenberg                        101           1
       200 Whalen                           101           1
       203 Mavris                           101           1
       204 Baer                             101           1
       205 Higgins                          101           1
       109 Faviet                           108           2
       110 Chen                             108           2
       111 Sciarra                          108           2
       112 Urman                            108           2
       113 Popp                             108           2
       206 Gietz                            205           2
```

The following statement shows employees who directly or indirectly report to employee 101, their reporting level, and their management chain.

```
WITH
  reports_to_101 (eid, emp_last, mgr_id, reportLevel, mgr_list) AS
  (
     SELECT employee_id, last_name, manager_id, 0 reportLevel,
            CAST(manager_id AS VARCHAR2(2000))
     FROM employees
     WHERE employee_id = 101
  UNION ALL
     SELECT e.employee_id, e.last_name, e.manager_id, reportLevel+1,
            CAST(mgr_list || ',' || manager_id AS VARCHAR2(2000))
     FROM reports_to_101 r, employees e
     WHERE r.eid = e.manager_id
  )
SELECT eid, emp_last, mgr_id, reportLevel, mgr_list
FROM reports_to_101
ORDER BY reportLevel, eid;

        EID EMP_LAST                      MGR_ID REPORTLEVEL MGR_LIST
 ---------- ------------------------- ---------- ----------- --------
       101 Kochhar                          100           0  100
       108 Greenberg                        101           1  100,101
       200 Whalen                           101           1  100,101
       203 Mavris                           101           1  100,101
       204 Baer                             101           1  100,101
       205 Higgins                          101           1  100,101
```

```
       109 Faviet                            108          2  100,101,108
       110 Chen                              108          2  100,101,108
       111 Sciarra                           108          2  100,101,108
       112 Urman                             108          2  100,101,108
       113 Popp                              108          2  100,101,108
       206 Gietz                             205          2  100,101,205
```

The following statement shows the employees who directly or indirectly report to employee 101 and their reporting level. It stops at reporting level 1.

```
WITH
  reports_to_101 (eid, emp_last, mgr_id, reportLevel) AS
  (
    SELECT employee_id, last_name, manager_id, 0 reportLevel
    FROM employees
    WHERE employee_id = 101
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id, reportLevel+1
    FROM reports_to_101 r, employees e
    WHERE r.eid = e.manager_id
  )
SELECT eid, emp_last, mgr_id, reportLevel
FROM reports_to_101
WHERE reportLevel <= 1
ORDER BY reportLevel, eid;

       EID EMP_LAST                      MGR_ID REPORTLEVEL
---------- ------------------------- ---------- -----------
       101 Kochhar                          100           0
       108 Greenberg                        101           1
       200 Whalen                           101           1
       203 Mavris                           101           1
       204 Baer                             101           1
       205 Higgins                          101           1
```

The following statement shows the entire organization, indenting for each level of management.

```
WITH
  org_chart (eid, emp_last, mgr_id, reportLevel, salary, job_id) AS
  (
    SELECT employee_id, last_name, manager_id, 0 reportLevel, salary, job_id
    FROM employees
    WHERE manager_id is null
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id,
           r.reportLevel+1 reportLevel, e.salary, e.job_id
    FROM org_chart r, employees e
    WHERE r.eid = e.manager_id
  )
  SEARCH DEPTH FIRST BY emp_last SET order1
SELECT lpad(' ',2*reportLevel)||emp_last emp_name, eid, mgr_id, salary, job_id
FROM org_chart
ORDER BY order1;

EMP_NAME                    EID     MGR_ID     SALARY JOB_ID
-------------------- ---------- ---------- ---------- ----------
King                        100                 24000 AD_PRES
  Cambrault                 148        100      11000 SA_MAN
    Bates                   172        148       7300 SA_REP
    Bloom                   169        148      10000 SA_REP
```

```
    Fox                        170        148       9600 SA_REP
    Kumar                      173        148       6100 SA_REP
    Ozer                       168        148      11500 SA_REP
    Smith                      171        148       7400 SA_REP
  De Haan                      102        100      17000 AD_VP
    Hunold                     103        102       9000 IT_PROG
      Austin                   105        103       4800 IT_PROG
      Ernst                    104        103       6000 IT_PROG
      Lorentz                  107        103       4200 IT_PROG
      Pataballa                106        103       4800 IT_PROG
  Errazuriz                    147        100      12000 SA_MAN
    Ande                       166        147       6400 SA_REP
. . .
```

The following statement shows the entire organization, indenting for each level of management, with each level ordered by *hire_date*. The value of *is_cycle* is set to `Y` for any employee who has the same *hire_date* as any manager above him in the management chain.

```
WITH
  dup_hiredate (eid, emp_last, mgr_id, reportLevel, hire_date, job_id) AS
  (
    SELECT employee_id, last_name, manager_id, 0 reportLevel, hire_date, job_id
    FROM employees
    WHERE manager_id is null
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id,
           r.reportLevel+1 reportLevel, e.hire_date, e.job_id
    FROM dup_hiredate r, employees e
    WHERE r.eid = e.manager_id
  )
  SEARCH DEPTH FIRST BY hire_date SET order1
  CYCLE hire_date SET is_cycle TO 'Y' DEFAULT 'N'
SELECT lpad(' ',2*reportLevel)||emp_last emp_name, eid, mgr_id,
       hire_date, job_id, is_cycle
FROM dup_hiredate
ORDER BY order1;

EMP_NAME                    EID     MGR_ID HIRE_DATE JOB_ID     IS_CYCLE
-------------------- ---------- ---------- --------- ---------- --------
King                        100            17-JUN-03 AD_PRES           N
  De Haan                   102        100 13-JAN-01 AD_VP             N
    Hunold                  103        102 03-JAN-06 IT_PROG           N
      Austin                105        103 25-JUN-05 IT_PROG           N
. . .
  Kochhar                   101        100 21-SEP-05 AD_VP             N
    Mavris                  203        101 07-JUN-02 HR_REP            N
    Baer                    204        101 07-JUN-02 PR_REP            N
    Higgins                 205        101 07-JUN-02 AC_MGR            N
      Gietz                 206        205 07-JUN-02 AC_ACCOUNT        Y
    Greenberg               108        101 17-AUG-02 FI_MGR            N
      Faviet                109        108 16-AUG-02 FI_ACCOUNT        N
      Chen                  110        108 28-SEP-05 FI_ACCOUNT        N
. . .
```

The following statement counts the number of employees under each manager.

```
WITH
  emp_count (eid, emp_last, mgr_id, mgrLevel, salary, cnt_employees) AS
  (
    SELECT employee_id, last_name, manager_id, 0 mgrLevel, salary, 0
```

```
cnt_employees
    FROM employees
  UNION ALL
    SELECT e.employee_id, e.last_name, e.manager_id,
           r.mgrLevel+1 mgrLevel, e.salary, 1 cnt_employees
    FROM emp_count r, employees e
    WHERE e.employee_id = r.mgr_id
  )
  SEARCH DEPTH FIRST BY emp_last SET order1
SELECT emp_last, eid, mgr_id, salary, sum(cnt_employees), max(mgrLevel) mgrLevel
FROM emp_count
GROUP BY emp_last, eid, mgr_id, salary
HAVING max(mgrLevel) > 0
ORDER BY mgr_id NULLS FIRST, emp_last;

EMP_LAST                  EID     MGR_ID     SALARY SUM(CNT_EMPLOYEES)   MGRLEVEL
------------------ ---------- ---------- ---------- ------------------ ----------
King                      100                 24000                106          3
Cambrault                 148        100      11000                  7          2
De Haan                   102        100      17000                  5          2
Errazuriz                 147        100      12000                  6          1
Fripp                     121        100       8200                  8          1
Hartstein                 201        100      13000                  1          1
Kaufling                  122        100       7900                  8          1
. . .
```

Simple Query Examples

The following statement selects rows from the `employees` table with the department number of 30:

```
SELECT *
   FROM employees
   WHERE department_id = 30
   ORDER BY last_name;
```

The following statement selects the name, job, salary and department number of all employees except purchasing clerks from department number 30:

```
SELECT last_name, job_id, salary, department_id
   FROM employees
   WHERE NOT (job_id = 'PU_CLERK' AND department_id = 30)
   ORDER BY last_name;
```

The following statement selects from subqueries in the `FROM` clause and for each department returns the total employees and salaries as a decimal value of all the departments:

```
SELECT a.department_id "Department",
   a.num_emp/b.total_count "%_Employees",
   a.sal_sum/b.total_sal "%_Salary"
FROM
(SELECT department_id, COUNT(*) num_emp, SUM(salary) sal_sum
   FROM employees
   GROUP BY department_id) a,
(SELECT COUNT(*) total_count, SUM(salary) total_sal
   FROM employees) b
ORDER BY a.department_id;
```

Selecting from a Partition: Example

You can select rows from a single partition of a partitioned table by specifying the keyword PARTITION in the FROM clause. This SQL statement assigns an alias for and retrieves rows from the sales_q2_2000 partition of the sample table sh.sales:

```
SELECT * FROM sales PARTITION (sales_q2_2000) s
   WHERE s.amount_sold > 1500
   ORDER BY cust_id, time_id, channel_id;
```

The following example selects rows from the oe.orders table for orders earlier than a specified date:

```
SELECT * FROM orders
   WHERE order_date < TO_DATE('2006-06-15', 'YYYY-MM-DD');
```

Selecting a Sample: Examples

The following query estimates the number of orders in the oe.orders table:

```
SELECT COUNT(*) * 10 FROM orders SAMPLE (10);

COUNT(*)*10
-----------
         70
```

Because the query returns an estimate, the actual return value may differ from one query to the next.

```
SELECT COUNT(*) * 10 FROM orders SAMPLE (10);

COUNT(*)*10
-----------
         80
```

The following query adds a seed value to the preceding query. Oracle Database always returns the same estimate given the same seed value:

```
SELECT COUNT(*) * 10 FROM orders SAMPLE(10) SEED (1);

COUNT(*)*10
-----------
        130

SELECT COUNT(*) * 10 FROM orders SAMPLE(10) SEED(4);

COUNT(*)*10
-----------
        120

SELECT COUNT(*) * 10 FROM orders SAMPLE(10) SEED (1);

COUNT(*)*10
-----------
        130
```

Using Flashback Queries: Example

The following statements show a current value from the sample table hr.employees and then change the value. The intervals used in these examples are very short for demonstration purposes. Time intervals in your own environment are likely to be larger.

```
SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      3800

UPDATE employees SET salary = 4000
   WHERE last_name = 'Chung';
1 row updated.

SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      4000
```

To learn what the value was before the update, you can use the following Flashback Query:

```
SELECT salary FROM employees
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' MINUTE)
   WHERE last_name = 'Chung';

    SALARY
----------
      3800
```

To learn what the values were during a particular time period, you can use a version Flashback Query:

```
SELECT salary FROM employees
  VERSIONS BETWEEN TIMESTAMP
    SYSTIMESTAMP - INTERVAL '10' MINUTE AND
    SYSTIMESTAMP - INTERVAL '1' MINUTE
  WHERE last_name = 'Chung';
```

To revert to the earlier value, use the Flashback Query as the subquery of another `UPDATE` statement:

```
UPDATE employees SET salary =
   (SELECT salary FROM employees
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '2' MINUTE)
   WHERE last_name = 'Chung')
   WHERE last_name = 'Chung';
1 row updated.

SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      3800
```

Using the GROUP BY Clause: Examples

To return the minimum and maximum salaries for each department in the `employees` table, issue the following statement:

```
SELECT department_id, MIN(salary), MAX (salary)
     FROM employees
```

```
     GROUP BY department_id
   ORDER BY department_id;
```

To return the minimum and maximum salaries for the clerks in each department, issue the following statement:

```
SELECT department_id, MIN(salary), MAX (salary)
     FROM employees
     WHERE job_id = 'PU_CLERK'
     GROUP BY department_id
   ORDER BY department_id;
```

Using the GROUP BY CUBE Clause: Example

To return the number of employees and their average yearly salary across all possible combinations of department and job category, issue the following query on the sample tables `hr.employees` and `hr.departments`:

```
SELECT DECODE(GROUPING(department_name), 1, 'All Departments',
      department_name) AS department_name,
   DECODE(GROUPING(job_id), 1, 'All Jobs', job_id) AS job_id,
   COUNT(*) "Total Empl", AVG(salary) * 12 "Average Sal"
   FROM employees e, departments d
   WHERE d.department_id = e.department_id
   GROUP BY CUBE (department_name, job_id)
   ORDER BY department_name, job_id;

DEPARTMENT_NAME                JOB_ID     Total Empl Average Sal
------------------------------ ---------- ---------- -----------
Accounting                     AC_ACCOUNT          1       99600
Accounting                     AC_MGR              1      144000
Accounting                     All Jobs            2      121800
Administration                 AD_ASST             1       52800
. . .
Shipping                       ST_CLERK           20       33420
Shipping                       ST_MAN              5       87360
```

Using the GROUPING SETS Clause: Example

The following example finds the sum of sales aggregated for three precisely specified groups:

- `(channel_desc, calendar_month_desc, country_id)`
- `(channel_desc, country_id)`
- `(calendar_month_desc, country_id)`

Without the `GROUPING SETS` syntax, you would have to write less efficient queries with more complicated SQL. For example, you could run three separate queries and `UNION` them, or run a query with a `CUBE(channel_desc, calendar_month_desc, country_id)` operation and filter out five of the eight groups it would generate.

```
SELECT channel_desc, calendar_month_desc, co.country_id,
      TO_CHAR(sum(amount_sold) , '9,999,999,999') SALES$
   FROM sales, customers, times, channels, countries co
   WHERE sales.time_id=times.time_id
      AND sales.cust_id=customers.cust_id
      AND sales.channel_id= channels.channel_id
      AND customers.country_id = co.country_id
      AND channels.channel_desc IN ('Direct Sales', 'Internet')
      AND times.calendar_month_desc IN ('2000-09', '2000-10')
```

```
      AND co.country_iso_code IN ('UK', 'US')
  GROUP BY GROUPING SETS(
      (channel_desc, calendar_month_desc, co.country_id),
      (channel_desc, co.country_id),
      (calendar_month_desc, co.country_id) );

CHANNEL_DESC         CALENDAR COUNTRY_ID     SALES$
-------------------- -------- ----------     ----------
Internet             2000-09       52790        124,224
Direct Sales         2000-09       52790        638,201
Internet             2000-10       52790        137,054
Direct Sales         2000-10       52790        682,297
                     2000-09       52790        762,425
                     2000-10       52790        819,351
Internet                           52790        261,278
Direct Sales                       52790      1,320,497
```

> **See Also:**
>
> The functions GROUP_ID , GROUPING , and GROUPING_ID for more information on those functions

Hierarchical Query Examples

The following query with a CONNECT BY clause defines a hierarchical relationship in which the employee_id value of the parent row is equal to the manager_id value of the child row:

```
SELECT last_name, employee_id, manager_id FROM employees
   CONNECT BY employee_id = manager_id
   ORDER BY last_name;
```

In the following CONNECT BY clause, the PRIOR operator applies only to the employee_id value. To evaluate this condition, the database evaluates employee_id values for the parent row and manager_id, salary, and commission_pct values for the child row:

```
SELECT last_name, employee_id, manager_id FROM employees
   CONNECT BY PRIOR employee_id = manager_id
   AND salary > commission_pct
   ORDER BY last_name;
```

To qualify as a child row, a row must have a manager_id value equal to the employee_id value of the parent row and it must have a salary value greater than its commission_pct value.

Using the HAVING Condition: Example

To return the minimum and maximum salaries for the employees in each department whose lowest salary is less than $5,000, issue the next statement:

```
SELECT department_id, MIN(salary), MAX (salary)
   FROM employees
   GROUP BY department_id
   HAVING MIN(salary) < 5000
   ORDER BY department_id;

DEPARTMENT_ID MIN(SALARY) MAX(SALARY)
------------- ----------- -----------
```

```
10          4400          4400
30          2500         11000
50          2100          8200
60          4200          9000
```

The following example uses a correlated subquery in a `HAVING` clause that eliminates from the result set any departments without managers and managers without departments:

```
SELECT department_id, manager_id
   FROM employees
   GROUP BY department_id, manager_id HAVING (department_id, manager_id) IN
   (SELECT department_id, manager_id FROM employees x
      WHERE x.department_id = employees.department_id)
   ORDER BY department_id;
```

Using the ORDER BY Clause: Examples

To select all purchasing clerk records from `employees` and order the results by salary in descending order, issue the following statement:

```
SELECT *
   FROM employees
   WHERE job_id = 'PU_CLERK'
   ORDER BY salary DESC;
```

To select information from `employees` ordered first by ascending department number and then by descending salary, issue the following statement:

```
SELECT last_name, department_id, salary
   FROM employees
   ORDER BY department_id ASC, salary DESC, last_name;
```

To select the same information as the previous `SELECT` and use the positional `ORDER BY` notation, issue the following statement, which orders by ascending `department_id`, then descending `salary`, and finally alphabetically by `last_name`:

```
SELECT last_name, department_id, salary
   FROM employees
   ORDER BY 2 ASC, 3 DESC, 1;
```

The MODEL clause: Examples

The view created below is based on the sample `sh` schema and is used by the example that follows.

```
CREATE OR REPLACE VIEW sales_view_ref AS
  SELECT country_name country,
         prod_name prod,
         calendar_year year,
         SUM(amount_sold) sale,
         COUNT(amount_sold) cnt
    FROM sales,times,customers,countries,products
    WHERE sales.time_id = times.time_id
      AND sales.prod_id = products.prod_id
      AND sales.cust_id = customers.cust_id
      AND customers.country_id = countries.country_id
      AND ( customers.country_id = 52779
            OR customers.country_id = 52776 )
      AND ( prod_name = 'Standard Mouse'
            OR prod_name = 'Mouse Pad' )
```

```
    GROUP BY country_name,prod_name,calendar_year;

SELECT country, prod, year, sale
  FROM sales_view_ref
  ORDER BY country, prod, year;

COUNTRY        PROD                                   YEAR       SALE
----------     -----------------------------------  --------  ---------
France         Mouse Pad                              1998    2509.42
France         Mouse Pad                              1999    3678.69
France         Mouse Pad                              2000    3000.72
France         Mouse Pad                              2001    3269.09
France         Standard Mouse                         1998    2390.83
France         Standard Mouse                         1999    2280.45
France         Standard Mouse                         2000    1274.31
France         Standard Mouse                         2001    2164.54
Germany        Mouse Pad                              1998    5827.87
Germany        Mouse Pad                              1999    8346.44
Germany        Mouse Pad                              2000    7375.46
Germany        Mouse Pad                              2001    9535.08
Germany        Standard Mouse                         1998    7116.11
Germany        Standard Mouse                         1999    6263.14
Germany        Standard Mouse                         2000    2637.31
Germany        Standard Mouse                         2001    6456.13

16 rows selected.
```

The next example creates a multidimensional array from `sales_view_ref` with columns containing country, product, year, and sales. It also:

- Assigns the sum of the sales of the Mouse Pad for years 1999 and 2000 to the sales of the Mouse Pad for year 2001, if a row containing sales of the Mouse Pad for year 2001 exists.
- Assigns the value of sales of the Standard Mouse for year 2001 to sales of the Standard Mouse for year 2002, creating a new row if a row containing sales of the Standard Mouse for year 2002 does not exist.

```
SELECT country,prod,year,s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[prod='Mouse Pad', year=2001] =
        s['Mouse Pad', 1999] + s['Mouse Pad', 2000],
      s['Standard Mouse', 2002] = s['Standard Mouse', 2001]
    )
  ORDER BY country, prod, year;

COUNTRY        PROD                                   YEAR       SALE
----------     -----------------------------------  --------  ---------
France         Mouse Pad                              1998    2509.42
France         Mouse Pad                              1999    3678.69
France         Mouse Pad                              2000    3000.72
France         Mouse Pad                              2001    6679.41
France         Standard Mouse                         1998    2390.83
```

```
France          Standard Mouse                                  1999     2280.45
France          Standard Mouse                                  2000     1274.31
France          Standard Mouse                                  2001     2164.54
France          Standard Mouse                                  2002     2164.54
Germany         Mouse Pad                                       1998     5827.87
Germany         Mouse Pad                                       1999     8346.44
Germany         Mouse Pad                                       2000     7375.46
Germany         Mouse Pad                                       2001     15721.9
Germany         Standard Mouse                                  1998     7116.11
Germany         Standard Mouse                                  1999     6263.14
Germany         Standard Mouse                                  2000     2637.31
Germany         Standard Mouse                                  2001     6456.13
Germany         Standard Mouse                                  2002     6456.13

18 rows selected.
```

The first rule uses `UPDATE` behavior because symbolic referencing is used on the left-hand side of the rule. The rows represented by the left-hand side of the rule exist, so the measure columns are updated. If the rows did not exist, then no action would have been taken.

The second rule uses `UPSERT` behavior because positional referencing is used on the left-hand side and a single cell is referenced. The rows do not exist, so new rows are inserted and the related measure columns are updated. If the rows did exist, then the measure columns would have been updated.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for an expanded discussion and examples

The next example uses the same `sales_view_ref` view and the analytic function `SUM` to calculate a cumulative sum (`csum`) of sales per country and per year.

```
SELECT country, year, sale, csum
   FROM
   (SELECT country, year, SUM(sale) sale
    FROM sales_view_ref
    GROUP BY country, year
   )
   MODEL DIMENSION BY (country, year)
         MEASURES (sale, 0 csum)
         RULES (csum[any, any]=
                  SUM(sale) OVER (PARTITION BY country
                                  ORDER BY year
                                  ROWS UNBOUNDED PRECEDING)
                )
   ORDER BY country, year;

COUNTRY               YEAR       SALE       CSUM
--------------- ---------- ---------- ----------
France                1998    4900.25    4900.25
France                1999    5959.14   10859.39
France                2000    4275.03   15134.42
France                2001    5433.63   20568.05
Germany               1998   12943.98   12943.98
Germany               1999   14609.58   27553.56
```

```
Germany              2000   10012.77   37566.33
Germany              2001   15991.21   53557.54

8 rows selected.
```

Row Limiting: Examples

The following statement returns the 5 employees with the lowest `employee_id` values:

```
SELECT employee_id, last_name
  FROM employees
  ORDER BY employee_id
  FETCH FIRST 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- -------------------------
        100 King
        101 Kochhar
        102 De Haan
        103 Hunold
        104 Ernst
```

The following statement returns the next 5 employees with the lowest `employee_id` values:

```
SELECT employee_id, last_name
  FROM employees
  ORDER BY employee_id
  OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;

EMPLOYEE_ID LAST_NAME
----------- -------------------------
        105 Austin
        106 Pataballa
        107 Lorentz
        108 Greenberg
        109 Faviet
```

The following statement returns the 5 percent of employees with the lowest salaries:

```
SELECT employee_id, last_name, salary
  FROM employees
  ORDER BY salary
  FETCH FIRST 5 PERCENT ROWS ONLY;

EMPLOYEE_ID LAST_NAME                     SALARY
----------- ------------------------- ----------
        132 Olson                           2100
        128 Markle                          2200
        136 Philtanker                      2200
        127 Landry                          2400
        135 Gee                             2400
        119 Colmenares                      2500
```

Because `WITH TIES` is specified, the following statement returns the 5 percent of employees with the lowest salaries, plus all additional employees with the same salary as the last row fetched in the previous example:

```
SELECT employee_id, last_name, salary
  FROM employees
  ORDER BY salary
  FETCH FIRST 5 PERCENT ROWS WITH TIES;
```

```
EMPLOYEE_ID LAST_NAME                     SALARY
----------- ------------------------- ----------
        132 Olson                           2100
        128 Markle                          2200
        136 Philtanker                      2200
        127 Landry                          2400
        135 Gee                             2400
        119 Colmenares                      2500
        131 Marlow                          2500
        140 Patel                           2500
        144 Vargas                          2500
        182 Sullivan                        2500
        191 Perkins                         2500
```

Using the FOR UPDATE Clause: Examples

The following statement locks rows in the `employees` table with purchasing clerks located in Oxford, which has `location_id` 2500, and locks rows in the `departments` table with departments in Oxford that have purchasing clerks:

```
SELECT e.employee_id, e.salary, e.commission_pct
   FROM employees e, departments d
   WHERE job_id = 'SA_REP'
   AND e.department_id = d.department_id
   AND location_id = 2500
   ORDER BY e.employee_id
   FOR UPDATE;
```

The following statement locks only those rows in the `employees` table with purchasing clerks located in Oxford. No rows are locked in the `departments` table:

```
SELECT e.employee_id, e.salary, e.commission_pct
   FROM employees e JOIN departments d
   USING (department_id)
   WHERE job_id = 'SA_REP'
   AND location_id = 2500
   ORDER BY e.employee_id
   FOR UPDATE OF e.salary;
```

Using the WITH CHECK OPTION Clause: Example

The following statement is legal even though the third value inserted violates the condition of the subquery *`where_clause`*:

```
INSERT INTO (SELECT department_id, department_name, location_id
   FROM departments WHERE location_id < 2000)
   VALUES (9999, 'Entertainment', 2500);
```

However, the following statement is illegal because it contains the `WITH CHECK OPTION` clause:

```
INSERT INTO (SELECT department_id, department_name, location_id
   FROM departments WHERE location_id < 2000 WITH CHECK OPTION)
   VALUES (9999, 'Entertainment', 2500);
     *
ERROR at line 2:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Using PIVOT and UNPIVOT: Examples

The oe.orders table contains information about when an order was placed (order_date), how it was place (order_mode), and the total amount of the order (order_total), as well as other information. The following example shows how to use the PIVOT clause to pivot order_mode values into columns, aggregating order_total data in the process, to get yearly totals by order mode:

```
CREATE TABLE pivot_table AS
SELECT * FROM
(SELECT EXTRACT(YEAR FROM order_date) year, order_mode, order_total FROM orders)
PIVOT
(SUM(order_total) FOR order_mode IN ('direct' AS Store, 'online' AS Internet));

SELECT * FROM pivot_table ORDER BY year;

      YEAR      STORE   INTERNET
---------- ---------- ----------
      2004     5546.6
      2006   371895.5   100056.6
      2007  1274078.8  1271019.5
      2008   252108.3   393349.4
```

The UNPIVOT clause lets you rotate specified columns so that the input column headings are output as values of one or more descriptor columns, and the input column values are output as values of one or more measures columns. The first query that follows shows that nulls are excluded by default. The second query shows that you can include nulls using the INCLUDE NULLS clause.

```
SELECT * FROM pivot_table
  UNPIVOT (yearly_total FOR order_mode IN (store AS 'direct',
           internet AS 'online'))
  ORDER BY year, order_mode;

      YEAR ORDER_ YEARLY_TOTAL
---------- ------ ------------
      2004 direct       5546.6
      2006 direct     371895.5
      2006 online     100056.6
      2007 direct    1274078.8
      2007 online    1271019.5
      2008 direct     252108.3
      2008 online     393349.4

7 rows selected.

SELECT * FROM pivot_table
  UNPIVOT INCLUDE NULLS
    (yearly_total FOR order_mode IN (store AS 'direct', internet AS 'online'))
  ORDER BY year, order_mode;

      YEAR ORDER_ YEARLY_TOTAL
---------- ------ ------------
      2004 direct       5546.6
      2004 online
      2006 direct     371895.5
      2006 online     100056.6
      2007 direct    1274078.8
      2007 online    1271019.5
      2008 direct     252108.3
      2008 online     393349.4
```

```
8 rows selected.
```

Using Join Queries: Examples

The following examples show various ways of joining tables in a query. In the first example, an equijoin returns the name and job of each employee and the number and name of the department in which the employee works:

```
SELECT last_name, job_id, departments.department_id, department_name
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   ORDER BY last_name, job_id;

LAST_NAME            JOB_ID     DEPARTMENT_ID DEPARTMENT_NAME
-------------------- ---------- ------------- ----------------------
Abel                  SA_REP               80 Sales
Ande                  SA_REP               80 Sales
Atkinson              ST_CLERK             50 Shipping
Austin                IT_PROG              60 IT
. . .
```

You must use a join to return this data because employee names and jobs are stored in a different table than department names. Oracle Database combines rows of the two tables according to this join condition:

```
employees.department_id = departments.department_id
```

The following equijoin returns the name, job, department number, and department name of all sales managers:

```
SELECT last_name, job_id, departments.department_id, department_name
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   AND job_id = 'SA_MAN'
   ORDER BY last_name;

LAST_NAME            JOB_ID     DEPARTMENT_ID DEPARTMENT_NAME
-------------------- ---------- ------------- -----------------------
Cambrault            SA_MAN                80 Sales
Errazuriz            SA_MAN                80 Sales
Partners             SA_MAN                80 Sales
Russell              SA_MAN                80 Sales
Zlotkey              SA_MAN                80 Sales
```

This query is identical to the preceding example, except that it uses an additional *where_clause* condition to return only rows with a `job` value of '`SA_MAN`'.

Using Subqueries: Examples

To determine who works in the same department as employee '`Lorentz`', issue the following statement:

```
SELECT last_name, department_id FROM employees
   WHERE department_id =
     (SELECT department_id FROM employees
      WHERE last_name = 'Lorentz')
   ORDER BY last_name, department_id;
```

To give all employees in the `employees` table a 10% raise if they have changed jobs—if they appear in the `job_history` table—issue the following statement:

```
UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id IN (SELECT employee_id FROM job_history);
```

To create a second version of the `departments` table `new_departments`, with only three of the columns of the original table, issue the following statement:

```
CREATE TABLE new_departments
   (department_id, department_name, location_id)
   AS SELECT department_id, department_name, location_id
   FROM departments;
```

Using Self Joins: Example

The following query uses a self join to return the name of each employee along with the name of the employee's manager. A `WHERE` clause is added to shorten the output.

```
SELECT e1.last_name||' works for '||e2.last_name
   "Employees and Their Managers"
   FROM employees e1, employees e2
   WHERE e1.manager_id = e2.employee_id
      AND e1.last_name LIKE 'R%'
   ORDER BY e1.last_name;

Employees and Their Managers
-------------------------------
Rajs works for Mourgos
Raphaely works for King
Rogers works for Kaufling
Russell works for King
```

The join condition for this query uses the aliases `e1` and `e2` for the sample table `employees`:

```
e1.manager_id = e2.employee_id
```

Using Outer Joins: Examples

The following example shows how a partitioned outer join fills data gaps in rows to facilitate analytic function specification and reliable report formatting. The example first creates a small data table to be used in the join:

```
SELECT d.department_id, e.last_name
   FROM departments d LEFT OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id, e.last_name;
```

Users familiar with the traditional Oracle Database outer joins syntax will recognize the same query in this form:

```
SELECT d.department_id, e.last_name
   FROM departments d, employees e
   WHERE d.department_id = e.department_id(+)
   ORDER BY d.department_id, e.last_name;
```

Oracle strongly recommends that you use the more flexible `FROM` clause join syntax shown in the former example.

The left outer join returns all departments, including those without any employees. The same statement with a right outer join returns all employees, including those not yet assigned to a department:

> **Note:**
>
> The employee Zeuss was added to the employees table for these examples, and is not part of the sample data.

```
SELECT d.department_id, e.last_name
   FROM departments d RIGHT OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id, e.last_name;

DEPARTMENT_ID LAST_NAME
------------- -------------------------
. . .
          110 Gietz
          110 Higgins
              Grant
              Zeuss
```

It is not clear from this result whether employees Grant and Zeuss have `department_id` `NULL`, or whether their `department_id` is not in the `departments` table. To determine this requires a full outer join:

```
SELECT d.department_id as d_dept_id, e.department_id as e_dept_id,
      e.last_name
   FROM departments d FULL OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id, e.last_name;

 D_DEPT_ID  E_DEPT_ID LAST_NAME
---------- ---------- -------------------------
  . . .
       110        110 Gietz
       110        110 Higgins
  . . .
       260
       270
                  999 Zeuss
                      Grant
```

Because the column names in this example are the same in both tables in the join, you can also use the common column feature by specifying the `USING` clause of the join syntax. The output is the same as for the preceding example except that the `USING` clause coalesces the two matching columns `department_id` into a single column output:

```
SELECT department_id AS d_e_dept_id, e.last_name
   FROM departments d FULL OUTER JOIN employees e
   USING (department_id)
   ORDER BY department_id, e.last_name;

D_E_DEPT_ID LAST_NAME
----------- -------------------------
  . . .
        110 Higgins
        110 Gietz
  . . .
        260
```

```
       270
       999 Zeuss
           Grant
```

Using Partitioned Outer Joins: Examples

The following example shows how a partitioned outer join fills in gaps in rows to facilitate analytic calculation specification and reliable report formatting. The example first creates and populates a simple table to be used in the join:

```
CREATE TABLE inventory (time_id    DATE,
                        product    VARCHAR2(10),
                        quantity   NUMBER);

INSERT INTO inventory VALUES (TO_DATE('01/04/01', 'DD/MM/YY'), 'bottle', 10);
INSERT INTO inventory VALUES (TO_DATE('06/04/01', 'DD/MM/YY'), 'bottle', 10);
INSERT INTO inventory VALUES (TO_DATE('01/04/01', 'DD/MM/YY'), 'can', 10);
INSERT INTO inventory VALUES (TO_DATE('04/04/01', 'DD/MM/YY'), 'can', 10);

SELECT times.time_id, product, quantity FROM inventory
   PARTITION BY  (product)
   RIGHT OUTER JOIN times ON (times.time_id = inventory.time_id)
   WHERE times.time_id BETWEEN TO_DATE('01/04/01', 'DD/MM/YY')
      AND TO_DATE('06/04/01', 'DD/MM/YY')
   ORDER BY  2,1;

TIME_ID   PRODUCT      QUANTITY
--------- ---------- ----------
01-APR-01 bottle             10
02-APR-01 bottle
03-APR-01 bottle
04-APR-01 bottle
05-APR-01 bottle
06-APR-01 bottle             10
01-APR-01 can                10
02-APR-01 can
03-APR-01 can
04-APR-01 can                10
05-APR-01 can
06-APR-01 can

12 rows selected.
```

The data is now more dense along the time dimension for each partition of the product dimension. However, each of the newly added rows within each partition is null in the quantity column. It is more useful to see the nulls replaced by the preceding non-`NULL` value in time order. You can achieve this by applying the analytic function `LAST_VALUE` on top of the query result:

```
SELECT time_id, product, LAST_VALUE(quantity IGNORE NULLS)
   OVER (PARTITION BY product ORDER BY time_id) quantity
   FROM ( SELECT times.time_id, product, quantity
             FROM inventory PARTITION BY  (product)
                RIGHT OUTER JOIN times ON (times.time_id = inventory.time_id)
   WHERE times.time_id BETWEEN TO_DATE('01/04/01', 'DD/MM/YY')
      AND TO_DATE('06/04/01', 'DD/MM/YY'))
   ORDER BY  2,1;

TIME_ID   PRODUCT      QUANTITY
--------- ---------- ----------
01-APR-01 bottle             10
```

```
02-APR-01 bottle            10
03-APR-01 bottle            10
04-APR-01 bottle            10
05-APR-01 bottle            10
06-APR-01 bottle            10
01-APR-01 can               10
02-APR-01 can               10
03-APR-01 can               10
04-APR-01 can               10
05-APR-01 can               10
06-APR-01 can               10

12 rows selected.
```

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for an expanded discussion on filling gaps in time series calculations and examples of usage

Using Antijoins: Example

The following example selects a list of employees who are not in a particular set of departments:

```
SELECT * FROM employees
   WHERE department_id NOT IN
   (SELECT department_id FROM departments
       WHERE location_id = 1700)
   ORDER BY last_name;
```

Using Semijoins: Example

In the following example, only one row needs to be returned from the `departments` table, even though many rows in the `employees` table might match the subquery. If no index has been defined on the `salary` column in `employees`, then a semijoin can be used to improve query performance.

```
SELECT * FROM departments
   WHERE EXISTS
   (SELECT * FROM employees
       WHERE departments.department_id = employees.department_id
       AND employees.salary > 2500)
   ORDER BY department_name;
```

Using CROSS APPLY and OUTER APPLY Joins: Examples

The following statement uses the `CROSS APPLY` clause of the *`cross_outer_apply_clause`*. The join returns only rows from the table on the left side of the join (`departments`) that produce a result from the inline view on the right side of the join. That is, the join returns only the departments that have at least one employee. The `WHERE` clause restricts the result set to include only the Marketing, Operations, and Public Relations departments. However, the Operations department is not included in the result set because it has no employees.

```
SELECT d.department_name, v.employee_id, v.last_name
  FROM departments d CROSS APPLY (SELECT * FROM employees e
                                  WHERE e.department_id = d.department_id) v
```

```
  WHERE d.department_name IN ('Marketing', 'Operations', 'Public Relations')
  ORDER BY d.department_name, v.employee_id;

DEPARTMENT_NAME                EMPLOYEE_ID LAST_NAME
------------------------------ ----------- -------------------------
Marketing                      201         Hartstein
Marketing                      202         Fay
Public Relations               204         Baer
```

The following statement uses the `OUTER APPLY` clause of the *`cross_outer_apply_clause`*. The join returns all rows from the table on the left side of the join (`departments`) regardless of whether they produce a result from the inline view on the right side of the join. That is, the join returns all departments regardless of whether the departments have any employees. The `WHERE` clause restricts the result set to include only the Marketing, Operations, and Public Relations departments. The Operations department is included in the result set even though it has no employees.

```
SELECT d.department_name, v.employee_id, v.last_name
  FROM departments d OUTER APPLY (SELECT * FROM employees e
                                  WHERE e.department_id = d.department_id) v
  WHERE d.department_name IN ('Marketing', 'Operations', 'Public Relations')
  ORDER by d.department_name, v.employee_id;

DEPARTMENT_NAME                EMPLOYEE_ID LAST_NAME
------------------------------ ----------- -------------------------
Marketing                      201         Hartstein
Marketing                      202         Fay
Operations
Public Relations               204         Baer
```

Using Lateral Inline Views: Example

The following example shows a join with two operands. The second operand is an inline view that specifies the first operand, table `e`, in the `WHERE` clause. This results in an error.

```
SELECT * FROM employees e, (SELECT * FROM departments d
                            WHERE e.department_id = d.department_id);
ORA-00904: "E"."DEPARTMENT_ID": invalid identifier
```

The following example shows a join with two operands. The second operand is a lateral inline view that specifies the first operand, table `e`, in the `WHERE` clause and succeeds without an error.

```
SELECT * FROM employees e, LATERAL(SELECT * FROM departments d
                                   WHERE e.department_id = d.department_id);
```

Table Collections: Examples

You can perform DML operations on nested tables only if they are defined as columns of a table. Therefore, when the *`query_table_expr_clause`* of an `INSERT`, `DELETE`, or `UPDATE` statement is a *`table_collection_expression`*, the collection expression must be a subquery that uses the `TABLE` collection expression to select the nested table column of the table. The examples that follow are based on the following scenario:

Suppose the database contains a table `hr_info` with columns `department_id`, `location_id`, and `manager_id`, and a column of nested table type `people` which has `last_name`, `department_id`, and `salary` columns for all the employees of each respective manager:

```
CREATE TYPE people_typ AS OBJECT (
   last_name      VARCHAR2(25),
```

```
   department_id  NUMBER(4),
   salary         NUMBER(8,2));
/
CREATE TYPE people_tab_typ AS TABLE OF people_typ;
/
CREATE TABLE hr_info (
   department_id   NUMBER(4),
   location_id     NUMBER(4),
   manager_id      NUMBER(6),
   people          people_tab_typ)
   NESTED TABLE people STORE AS people_stor_tab;

INSERT INTO hr_info VALUES (280, 1800, 999, people_tab_typ());
```

The following example inserts into the `people` nested table column of the `hr_info` table for department 280:

```
INSERT INTO TABLE(SELECT h.people FROM hr_info h
   WHERE h.department_id = 280)
   VALUES ('Smith', 280, 1750);
```

The next example updates the department 280 `people` nested table:

```
UPDATE TABLE(SELECT h.people FROM hr_info h
   WHERE h.department_id = 280) p
   SET p.salary = p.salary + 100;
```

The next example deletes from the department 280 `people` nested table:

```
DELETE TABLE(SELECT h.people FROM hr_info h
   WHERE h.department_id = 280) p
   WHERE p.salary > 1700;
```

Collection Unnesting: Examples

To select data from a nested table column, use the `TABLE` collection expression to treat the nested table as columns of a table. This process is called **collection unnesting**.

You could get all the rows from `hr_info`, which was created in the preceding example, and all the rows from the `people` nested table column of `hr_info` using the following statement:

```
SELECT t1.department_id, t2.* FROM hr_info t1, TABLE(t1.people) t2
   WHERE t2.department_id = t1.department_id;
```

Now suppose that `people` is not a nested table column of `hr_info`, but is instead a separate table with columns `last_name`, `department_id`, `address`, `hiredate`, and `salary`. You can extract the same rows as in the preceding example with this statement:

```
SELECT t1.department_id, t2.*
   FROM hr_info t1, TABLE(CAST(MULTISET(
      SELECT t3.last_name, t3.department_id, t3.salary
         FROM people t3
      WHERE t3.department_id = t1.department_id)
      AS people_tab_typ)) t2;
```

Finally, suppose that `people` is neither a nested table column of table `hr_info` nor a table itself. Instead, you have created a function `people_func` that extracts from various sources the name, department, and salary of all employees. You can get the same information as in the preceding examples with the following query:

```
SELECT t1.department_id, t2.* FROM hr_info t1, TABLE(CAST
   (people_func( ... ) AS people_tab_typ)) t2;
```

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more examples of collection unnesting.

Using the LEVEL Pseudocolumn: Examples

The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is `AD_VP`. The child rows of a parent row are defined to be those who have the employee number of the parent row as their manager number.

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
        employee_id, manager_id, job_id
    FROM employees
    START WITH job_id = 'AD_VP'
    CONNECT BY PRIOR employee_id = manager_id;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
Kochhar                    101        100 AD_VP
  Greenberg                108        101 FI_MGR
    Faviet                 109        108 FI_ACCOUNT
    Chen                   110        108 FI_ACCOUNT
    Sciarra                111        108 FI_ACCOUNT
    Urman                  112        108 FI_ACCOUNT
    Popp                   113        108 FI_ACCOUNT
  Whalen                   200        101 AD_ASST
  Mavris                   203        101 HR_REP
  Baer                     204        101 PR_REP
  Higgins                  205        101 AC_MGR
    Gietz                  206        205 AC_ACCOUNT
De Haan                    102        100 AD_VP
  Hunold                   103        102 IT_PROG
    Ernst                  104        103 IT_PROG
    Austin                 105        103 IT_PROG
    Pataballa              106        103 IT_PROG
    Lorentz                107        103 IT_PROG
```

The following statement is similar to the previous one, except that it does not select employees with the job `FI_MGR`.

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
        employee_id, manager_id, job_id
    FROM employees
    WHERE job_id != 'FI_MGR'
    START WITH job_id = 'AD_VP'
    CONNECT BY PRIOR employee_id = manager_id;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
Kochhar                    101        100 AD_VP
    Faviet                 109        108 FI_ACCOUNT
```

```
    Chen                       110        108 FI_ACCOUNT
    Sciarra                    111        108 FI_ACCOUNT
    Urman                      112        108 FI_ACCOUNT
    Popp                       113        108 FI_ACCOUNT
  Whalen                       200        101 AD_ASST
  Mavris                       203        101 HR_REP
  Baer                         204        101 PR_REP
  Higgins                      205        101 AC_MGR
    Gietz                      206        205 AC_ACCOUNT
De Haan                        102        100 AD_VP
  Hunold                       103        102 IT_PROG
    Ernst                      104        103 IT_PROG
    Austin                     105        103 IT_PROG
    Pataballa                  106        103 IT_PROG
    Lorentz                    107        103 IT_PROG
```

Oracle Database does not return the manager `Greenberg`, although it does return employees who are managed by `Greenberg`.

The following statement is similar to the first one, except that it uses the `LEVEL` pseudocolumn to select only the first two levels of the management hierarchy:

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
employee_id, manager_id, job_id
    FROM employees
    START WITH job_id = 'AD_PRES'
    CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 2;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
King                       100            AD_PRES
  Kochhar                  101        100 AD_VP
  De Haan                  102        100 AD_VP
  Raphaely                 114        100 PU_MAN
  Weiss                    120        100 ST_MAN
  Fripp                    121        100 ST_MAN
  Kaufling                 122        100 ST_MAN
  Vollman                  123        100 ST_MAN
  Mourgos                  124        100 ST_MAN
  Russell                  145        100 SA_MAN
  Partners                 146        100 SA_MAN
  Errazuriz                147        100 SA_MAN
  Cambrault                148        100 SA_MAN
  Zlotkey                  149        100 SA_MAN
  Hartstein                201        100 MK_MAN
```

Using Distributed Queries: Example

This example shows a query that joins the `departments` table on the local database with the `employees` table on the `remote` database:

```
SELECT last_name, department_name
   FROM employees@remote, departments
   WHERE employees.department_id = departments.department_id;
```

Using Correlated Subqueries: Examples

The following examples show the general syntax of a correlated subquery:

```
SELECT select_list
    FROM table1 t_alias1
    WHERE expr operator
        (SELECT column_list
            FROM table2 t_alias2
            WHERE t_alias1.column
               operator t_alias2.column);

UPDATE table1 t_alias1
    SET column =
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);

DELETE FROM table1 t_alias1
    WHERE column operator
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
```

The following statement returns data about employees whose salaries exceed their department average. The following statement assigns an alias to `employees`, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT department_id, last_name, salary
   FROM employees x
   WHERE salary > (SELECT AVG(salary)
      FROM employees
      WHERE x.department_id = department_id)
   ORDER BY department_id;
```

For each row of the `employees` table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs the following steps for each row of the `employees` table:

1. The `department_id` of the row is determined.
2. The `department_id` is then used to evaluate the parent query.
3. If the salary in that row is greater than the average salary of the departments of that row, then the row is returned.

The subquery is evaluated once for each row of the `employees` table.

Selecting from the DUAL Table: Example

The following statement returns the current date:

```
SELECT SYSDATE FROM DUAL;
```

You could select `SYSDATE` from the `employees` table, but the database would return 14 rows of the same `SYSDATE`, one for every row of the `employees` table. Selecting from `DUAL` is more convenient.

Selecting Sequence Values: Examples

The following statement increments the `employees_seq` sequence and returns the new value:

```
SELECT employees_seq.nextval
    FROM DUAL;
```

The following statement selects the current value of `employees_seq`:

```
SELECT employees_seq.currval
    FROM DUAL;
```

Row Pattern Matching: Example

This example uses row pattern matching to query stock price data. The following statements create table `Ticker` and inserts stock price data into the table:

```
CREATE TABLE Ticker (SYMBOL VARCHAR2(10), tstamp DATE, price NUMBER);

INSERT INTO Ticker VALUES('ACME', '01-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '02-Apr-11', 17);
INSERT INTO Ticker VALUES('ACME', '03-Apr-11', 19);
INSERT INTO Ticker VALUES('ACME', '04-Apr-11', 21);
INSERT INTO Ticker VALUES('ACME', '05-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '06-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '07-Apr-11', 15);
INSERT INTO Ticker VALUES('ACME', '08-Apr-11', 20);
INSERT INTO Ticker VALUES('ACME', '09-Apr-11', 24);
INSERT INTO Ticker VALUES('ACME', '10-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '11-Apr-11', 19);
INSERT INTO Ticker VALUES('ACME', '12-Apr-11', 15);
INSERT INTO Ticker VALUES('ACME', '13-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '14-Apr-11', 25);
INSERT INTO Ticker VALUES('ACME', '15-Apr-11', 14);
INSERT INTO Ticker VALUES('ACME', '16-Apr-11', 12);
INSERT INTO Ticker VALUES('ACME', '17-Apr-11', 14);
INSERT INTO Ticker VALUES('ACME', '18-Apr-11', 24);
INSERT INTO Ticker VALUES('ACME', '19-Apr-11', 23);
INSERT INTO Ticker VALUES('ACME', '20-Apr-11', 22);
```

The following query uses row pattern matching to find all cases where stock prices dipped to a bottom price and then rose. This is generally called a V-shape. The resulting output contains only three rows because the query specifies `ONE ROW PER MATCH`, and three matches were found.

```
SELECT *
FROM Ticker MATCH_RECOGNIZE (
     PARTITION BY symbol
     ORDER BY tstamp
     MEASURES STRT.tstamp AS start_tstamp,
              LAST(DOWN.tstamp) AS bottom_tstamp,
              LAST(UP.tstamp) AS end_tstamp
     ONE ROW PER MATCH
     AFTER MATCH SKIP TO LAST UP
     PATTERN (STRT DOWN+ UP+)
     DEFINE
        DOWN AS DOWN.price < PREV(DOWN.price),
        UP AS UP.price > PREV(UP.price)
     ) MR
ORDER BY MR.symbol, MR.start_tstamp;

SYMBOL     START_TST BOTTOM_TS END_TSTAM
---------- --------- --------- ---------
ACME       05-APR-11 06-APR-11 10-APR-11
ACME       10-APR-11 12-APR-11 13-APR-11
ACME       14-APR-11 16-APR-11 18-APR-11
```

SET CONSTRAINT[S]

Purpose

Use the SET CONSTRAINTS statement to specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement (IMMEDIATE) or when the transaction is committed (DEFERRED). You can use this statement to set the mode for a list of constraint names or for ALL constraints.

The SET CONSTRAINTS mode lasts for the duration of the transaction or until another SET CONSTRAINTS statement resets the mode.

> **Note:**
>
> You can also use an ALTER SESSION statement with the SET CONSTRAINTS clause to set *all* deferrable constraints. This is equivalent to making issuing a SET CONSTRAINTS statement at the start of each transaction in the current session.

You cannot specify this statement inside of a trigger definition.

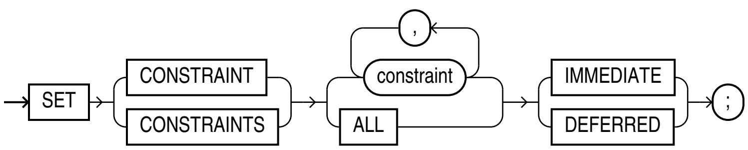
SET CONSTRAINTS can be a distributed statement. Existing database links that have transactions in process are notified when a SET CONSTRAINTS ALL statement is issued, and new links are notified that it was issued as soon as they start a transaction.

Prerequisites

To specify when a deferrable constraint is checked, you must have the READ or SELECT privilege on the table to which the constraint is applied unless the table is in your schema.

Syntax

***set_constraints*::=**



Semantics

constraint

Specify the name of one or more integrity constraints.

ALL

Specify ALL to set all deferrable constraints for this transaction.

IMMEDIATE

Specify `IMMEDIATE` to cause the specified constraints to be checked immediately on execution of each constrained DML statement. Oracle Database first checks any constraints that were deferred earlier in the transaction and then continues immediately checking constraints of any further statements in that transaction, as long as all the checked constraints are consistent and no other `SET CONSTRAINTS` statement is issued. If any constraint fails the check, then an error is signaled. At that point, a `COMMIT` statement causes the whole transaction to undo.

Making constraints immediate at the end of a transaction is a way of checking whether `COMMIT` can succeed. You can avoid unexpected rollbacks by setting constraints to `IMMEDIATE` as the last statement in a transaction. If any constraint fails the check, you can then correct the error before committing the transaction.

DEFERRED

Specify `DEFERRED` to indicate that the conditions specified by the deferrable constraint are checked when the transaction is committed.

> **Note:**
>
> You can verify the success of deferrable constraints prior to committing them by issuing a `SET CONSTRAINTS ALL IMMEDIATE` statement.

Examples

Setting Constraints: Examples

The following statement sets all deferrable constraints in this transaction to be checked immediately following each DML statement:

```
SET CONSTRAINTS ALL IMMEDIATE;
```

The following statement checks three deferred constraints when the transaction is committed. This example fails if the constraints were specified to be `NOT DEFERRABLE`.

```
SET CONSTRAINTS emp_job_nn, emp_salary_min,
   hr.jhist_dept_fk@remote DEFERRED;
```

SET ROLE

Purpose

When a user logs on to Oracle Database, the database enables all privileges granted explicitly to the user and all privileges in the user's default roles. During the session, the user or an application can use the `SET ROLE` statement any number of times to enable or disable the roles currently enabled for the session.

You cannot enable more than 148 user-defined roles at one time.

Note:

- For most roles, you cannot enable or disable a role unless it was granted to you either directly or through other roles. However, a secure application role can be granted and enabled by its associated PL/SQL package. See the `CREATE ROLE` semantics for USING *package* and *Oracle Database Security Guide* for information about secure application roles.
- `SET ROLE` succeeds only if there are no definer's rights units on the call stack. If at least one DR unit is on the call stack, then issuing the `SET ROLE` command causes `ORA-06565`. See *Oracle Database PL/SQL Language Reference* for more information about definer's rights units.
- To run the `SET ROLE` command from PL/SQL, you must use dynamic SQL, preferably the `EXECUTE IMMEDIATE` statement. See *Oracle Database PL/SQL Language Reference* for more information about this statement.

You can see which roles are currently enabled by examining the `SESSION_ROLES` data dictionary view.

See Also:

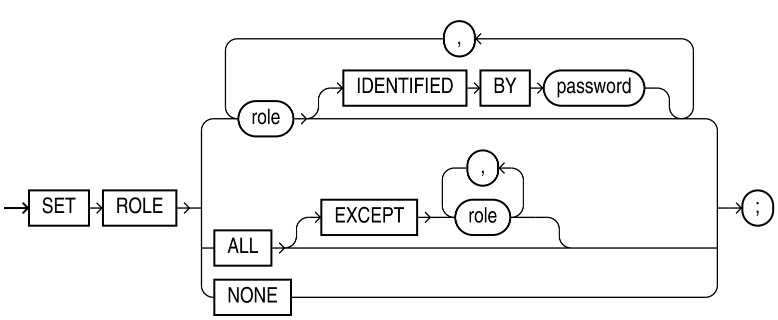
- CREATE ROLE for information on creating roles
- ALTER USER for information on changing a user's default roles
- *Oracle Database Reference* for information on the `SESSION_ROLES` session parameter

Prerequisites

You must already have been granted the roles that you name in the `SET ROLE` statement.

Syntax

set_role::=



Semantics

role

Specify one or more roles to be enabled for the current session. All roles not specified are disabled for the current session or until another `SET ROLE` statement is issued in the current session.

In the `IDENTIFIED BY` *`password`* clause, specify the password for a role. If the role has a password, then you must specify the password to enable the role.

Restriction on Setting Roles

You cannot specify a role identified globally. Global roles are enabled by default at login, and cannot be reenabled later.

ALL Clause

Specify `ALL` to enable all roles granted to you for the current session except those optionally listed in the `EXCEPT` clause.

Roles listed in the `EXCEPT` clause must be roles granted directly to you. They cannot be roles granted to you through other roles.

If you list a role in the `EXCEPT` clause that has been granted to you both directly and through another role, then the role remains enabled by virtue of the role to which it has been granted.

Restrictions on the ALL Clause

The following restrictions apply to the `ALL` clause:

- You cannot use this clause to enable roles with passwords that have been granted directly to you.
- You cannot use this clause to enable a secure application role, which is a role that can be enabled only by applications using an authorized package. Refer to *Oracle Database Security Guide* for information on creating a secure application role and *Oracle Database 2 Day + Security Guide* for a tutorial.

NONE

Specify `NONE` to disable all roles for the current session, including the `DEFAULT` role.

Examples

Setting Roles: Examples

To enable the role `dw_manager` identified by the password `warehouse` for your current session, issue the following statement:

```
SET ROLE dw_manager IDENTIFIED BY warehouse;
```

To enable all roles granted to you for the current session, issue the following statement:

```
SET ROLE ALL;
```

To enable all roles granted to you except `dw_manager`, issue the following statement:

```
SET ROLE ALL EXCEPT dw_manager;
```

To disable all roles granted to you for the current session, issue the following statement:

```
SET ROLE NONE;
```

SET TRANSACTION

Purpose

Use the `SET TRANSACTION` statement to establish the current transaction as read-only or read/write, establish its isolation level, assign it to a specified rollback segment, or assign a name to the transaction.

A transaction implicitly begins with any operation that obtains a TX lock:

- When a statement that modifies data is issued
- When a `SELECT` ... `FOR UPDATE` statement is issued
- When a transaction is explicitly started with a `SET TRANSACTION` statement or the `DBMS_TRANSACTION` package

Issuing either a `COMMIT` or `ROLLBACK` statement explicitly ends the current transaction.

The operations performed by a `SET TRANSACTION` statement affect only your current transaction, not other users or other transactions. Your transaction ends whenever you issue a `COMMIT` or `ROLLBACK` statement. Oracle Database implicitly commits the current transaction before and after executing a data definition language (DDL) statement.

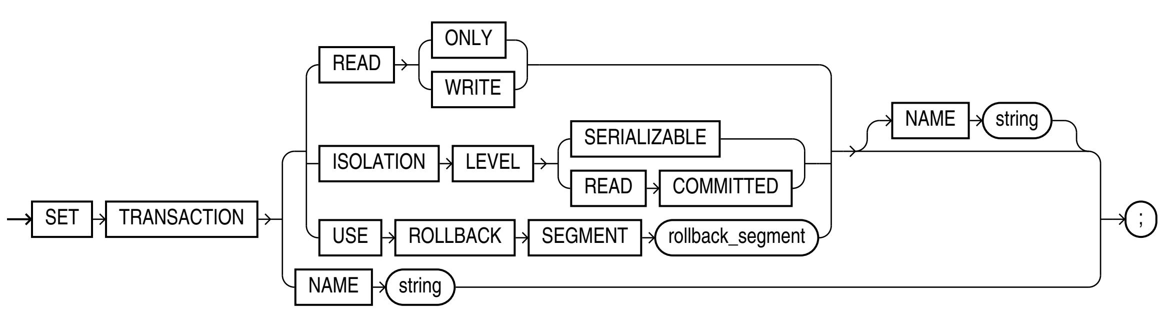
> **See Also:**
>
> COMMIT and ROLLBACK

Prerequisites

If you use a `SET TRANSACTION` statement, then it must be the first statement in your transaction. However, a transaction need not have a `SET TRANSACTION` statement.

Syntax

***set_transaction*::=**



Semantics

READ ONLY

The `READ ONLY` clause establishes the current transaction as a read-only transaction. This clause established **transaction-level read consistency**.

All subsequent queries in that transaction see only changes that were committed before the transaction began. Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

This clause is not supported for the user `SYS`. Queries by `SYS` will return changes made during the transaction even if `SYS` has set the transaction to be `READ ONLY`.

Restriction on Read-only Transactions

Only the following statements are permitted in a read-only transaction:

- Subqueries—`SELECT` statements without the *`for_update_clause`*
- `LOCK TABLE`
- `SET ROLE`
- `ALTER SESSION`
- `ALTER SYSTEM`

READ WRITE

Specify `READ WRITE` to establish the current transaction as a read/write transaction. This clause establishes **statement-level read consistency**, which is the default.

Restriction on Read/Write Transactions

You cannot toggle between transaction-level and statement-level read consistency in the same transaction.

ISOLATION LEVEL Clause

- The `SERIALIZABLE` setting specifies serializable transaction isolation mode as defined in the SQL standard. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.
- The `READ COMMITTED` setting is the default Oracle Database transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.

USE ROLLBACK SEGMENT Clause

> **Note:**
>
> This clause is relevant and valid only if you are using rollback segments for undo. Oracle strongly recommends that you use automatic undo management to handle undo space. If you follow this recommendation and run your database in automatic undo mode, then Oracle Database ignores this clause.

Specify `USE ROLLBACK SEGMENT` to assign the current transaction to the specified rollback segment. This clause also implicitly establishes the transaction as a read/write transaction.

Parallel DML requires more than one rollback segment. Therefore, if your transaction contains parallel DML operations, then the database ignores this clause.

NAME Clause

Use the `NAME` clause to assign a name to the current transaction. This clause is especially useful in distributed database environments when you must identify and resolve in-doubt transactions. The *`string`* value is limited to 255 bytes.

If you specify a name for a distributed transaction, then when the transaction commits, the name becomes the commit comment, overriding any comment specified explicitly in the `COMMIT` statement.

> **See Also:**
>
> *Oracle Database Concepts* for more information about transaction naming

Examples

Setting Transactions: Examples

The following statements could be run at midnight of the last day of every month to count the products and quantities on hand in the Toronto warehouse in the sample Order Entry (`oe`) schema. This report would not be affected by any other user who might be adding or removing inventory to a different warehouse.

```
COMMIT;

SET TRANSACTION READ ONLY NAME 'Toronto';

SELECT product_id, quantity_on_hand FROM inventories
   WHERE warehouse_id = 5
   ORDER BY product_id;

COMMIT;
```

The first `COMMIT` statement ensures that `SET TRANSACTION` is the first statement in the transaction. The last `COMMIT` statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

TRUNCATE CLUSTER

Purpose

> **Note:**
>
> You cannot roll back a `TRUNCATE CLUSTER` statement.

Use the `TRUNCATE CLUSTER` statement to remove all rows from a cluster. By default, Oracle Database also performs the following tasks:

- Deallocates all space used by the removed rows except that specified by the `MINEXTENTS` storage parameter
- Sets the `NEXT` storage parameter to the size of the last extent removed from the segment by the truncation process

Removing rows with the `TRUNCATE` statement can be more efficient than dropping and re-creating a cluster. Dropping and re-creating a cluster invalidates dependent objects of the cluster, requires you to regrant object privileges on the cluster, and requires you to re-create the indexes and cluster on the table and respecify its storage parameters. Truncating has none of these effects.

Removing rows with the `TRUNCATE CLUSTER` statement can be faster than removing all rows with the `DELETE` statement, especially if the cluster has numerous indexes and other dependencies.

> **See Also:**
>
> - DELETE and DROP CLUSTER for information on other ways of dropping data from a cluster
> - TRUNCATE TABLE for information on truncating a table

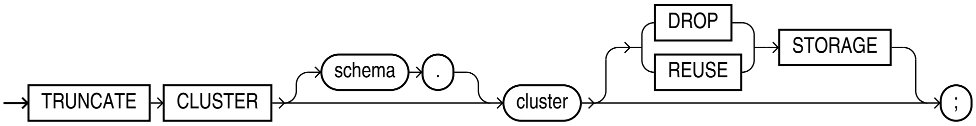
Prerequisites

To truncate a cluster, the cluster must be in your schema or you must have `DROP ANY TABLE` system privilege.

> **See Also:**
>
> "Restrictions on Truncating Tables"

Syntax

***truncate_cluster*::=**



Semantics

CLUSTER Clause

Specify the schema and name of the cluster to be truncated. You can truncate only an indexed cluster, not a hash cluster. If you omit *schema*, then the database assumes the cluster is in your own schema.

When you truncate a cluster, the database also automatically deletes all data in the indexes of the cluster tables.

STORAGE Clauses

The STORAGE clauses let you determine what happens to the space freed by the truncated rows. The DROP STORAGE clause and REUSE STORAGE clause also apply to the space freed by the data deleted from associated indexes.

DROP STORAGE

Specify DROP STORAGE to deallocate all space from the deleted rows from the cluster except the space allocated by the MINEXTENTS parameter of the cluster. This space can subsequently be used by other objects in the tablespace. Oracle Database also sets the NEXT storage parameter to the size of the last extent removed from the segment in the truncation process. This is the default.

REUSE STORAGE

Specify REUSE STORAGE to retain the space from the deleted rows allocated to the cluster. Storage values are not reset to the values when the table or cluster was created. This space can subsequently be used only by new data in the cluster resulting from insert or update operations. This clause leaves storage parameters at their current settings.

If you have specified more than one free list for the object you are truncating, then the REUSE STORAGE clause also removes any mapping of free lists to instances and resets the high-water mark to the beginning of the first extent.

Examples

Truncating a Cluster: Example

The following statement removes all rows from all tables in the personnel cluster, but leaves the freed space allocated to the tables:

```
TRUNCATE CLUSTER personnel REUSE STORAGE;
```

The preceding statement also removes all data from all indexes on the tables in the personnel cluster.

TRUNCATE TABLE

Purpose

> **Note:**
>
> You cannot roll back a TRUNCATE TABLE statement, nor can you use a FLASHBACK TABLE statement to retrieve the contents of a table that has been truncated.

Use the TRUNCATE TABLE statement to remove all rows from a table. By default, Oracle Database also performs the following tasks:

- Deallocates all space used by the removed rows except that specified by the MINEXTENTS storage parameter
- Sets the NEXT storage parameter to the size of the last extent removed from the segment by the truncation process

Removing rows with the TRUNCATE TABLE statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table invalidates dependent objects of the table, requires you to regrant object privileges on the table, and requires you to re-create the indexes, integrity constraints, and triggers on the table and respecify its storage parameters. Truncating has none of these effects.

Removing rows with the TRUNCATE TABLE statement can be faster than removing all rows with the DELETE statement, especially if the table has numerous triggers, indexes, and other dependencies.

> **See Also:**
>
> - DELETE and DROP TABLE for information on other ways of removing data from a table
> - TRUNCATE CLUSTER for information on truncating a cluster

Prerequisites

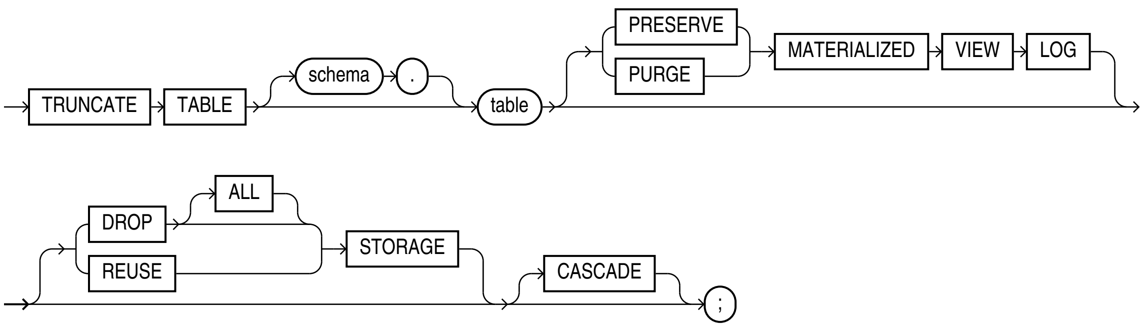
To truncate a table, the table must be in your schema or you must have the DROP ANY TABLE system privilege.

To specify the CASCADE clause, all affected child tables must be in your schema or you must have the DROP ANY TABLE system privilege.

> **See Also:**
>
> "Restrictions on Truncating Tables"

Syntax

***truncate_table*::=**



Semantics

TABLE Clause

Specify the schema and name of the table to be truncated. This table cannot be part of a cluster. If you omit *schema*, then Oracle Database assumes the table is in your own schema.

- You can truncate index-organized tables and temporary tables. When you truncate a temporary table, only the rows created during the current session are removed.
- Oracle Database changes the NEXT storage parameter of *table* to be the size of the last extent deleted from the segment in the process of truncation.
- Oracle Database also automatically truncates and resets any existing UNUSABLE indicators for the following indexes on *table*: range and hash partitions of local indexes and subpartitions of local indexes.
- If *table* is not empty, then the database marks UNUSABLE all nonpartitioned indexes and all partitions of global partitioned indexes on the table. However, when the table is truncated, the index is also truncated, and a new high water mark is calculated for the index segment. This operation is equivalent to creating a new segment for the index. Therefore, at the end of the truncate operation, the indexes are once again USABLE.
- For a domain index, this statement invokes the appropriate truncate routine to truncate the domain index data.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information on domain indexes

- If a regular or index-organized table contains LOB columns, then all LOB data and LOB index segments are truncated.
- If *table* is partitioned, then all partitions or subpartitions, as well as the LOB data and LOB index segments for each partition or subpartition, are truncated.

> **Note:**
>
> When you truncate a table, Oracle Database automatically removes all data in the table's indexes and any materialized view direct-path `INSERT` information held in association with the table. This information is independent of any materialized view log. If this direct-path `INSERT` information is removed, then an incremental refresh of the materialized view may lose data.

- All cursors are invalidated.

Restrictions on Truncating Tables

This statement is subject to the following restrictions:

- You cannot roll back a `TRUNCATE TABLE` statement.
- You cannot flash back to the state of the table before the truncate operation.
- You cannot individually truncate a table that is part of a cluster. You must either truncate the cluster, delete all rows from the table, or drop and re-create the table.
- You cannot truncate the parent table of an enabled foreign key constraint. You must disable the constraint before truncating the table. An exception is that you can truncate the table if the integrity constraint is self-referential.
- If a domain index is defined on *`table`*, then neither the index nor any index partitions can be marked `IN_PROGRESS`.
- You cannot truncate the parent table of a reference-partitioned table. You must first drop the reference-partitioned child table.
- You cannot truncate a duplicated table.

MATERIALIZED VIEW LOG Clause

The `MATERIALIZED VIEW LOG` clause lets you specify whether a materialized view log defined on the table is to be preserved or purged when the table is truncated. This clause permits materialized view master tables to be reorganized through export or import without affecting the ability of primary key materialized views defined on the master to be fast refreshed. To support continued fast refresh of primary key materialized views, the materialized view log must record primary key information.

> **Note:**
>
> The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

PRESERVE

Specify `PRESERVE` if any materialized view log should be preserved when the master table is truncated. This is the default.

PURGE

Specify PURGE if any materialized view log should be purged when the master table is truncated.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about materialized view logs and the TRUNCATE statement

STORAGE Clauses

The STORAGE clauses let you determine what happens to the space freed by the truncated rows. The DROP STORAGE clause, DROP ALL STORAGE clause, and REUSE STORAGE clause also apply to the space freed by the data deleted from associated indexes.

DROP STORAGE

Specify DROP STORAGE to deallocate all space from the deleted rows from the table except the space allocated by the MINEXTENTS parameter of the table. This space can subsequently be used by other objects in the tablespace. Oracle Database also sets the NEXT storage parameter to the size of the last extent removed from the segment in the truncation process. This setting, which is the default, is useful for small and medium-sized objects. The extent management in locally managed tablespace is very fast in these cases, so there is no need to reserve space.

DROP ALL STORAGE

Specify DROP ALL STORAGE to deallocate all space from the deleted rows from the table, including the space allocated by the MINEXTENTS parameter. All segments for the table, as well as all segments for its dependent objects, will be deallocated.

Restrictions on DROP ALL STORAGE

This clause is subject to the same restrictions as described in "Restrictions on Deferred Segment Creation".

REUSE STORAGE

Specify REUSE STORAGE to retain the space from the deleted rows allocated to the table. Storage values are not reset to the values when the table was created. This space can subsequently be used only by new data in the table resulting from insert or update operations. This clause leaves storage parameters at their current settings.

This setting is useful as an alternative to deleting all rows of a very large table—when the number of rows is very large, the table entails many thousands of extents, and when data is to be reinserted in the future.

This clause is not valid for temporary tables. A session becomes unbound from the temporary table when the table is truncated, so the storage is automatically dropped.

If you have specified more than one free list for the object you are truncating, then the REUSE STORAGE clause also removes any mapping of free lists to instances and resets the high-water mark to the beginning of the first extent.

CASCADE

If you specify `CASCADE`, then Oracle Database truncates all child tables that reference *`table`* with an enabled `ON DELETE CASCADE` referential constraint. This is a recursive operation that will truncate all child tables, granchild tables, and so on, using the specified options.

Examples

Truncating a Table: Example

The following statement removes all rows from a hypothetical copy of the sample table `hr.employees` and returns the freed space to the tablespace containing `employees`:

```
TRUNCATE TABLE employees_demo;
```

The preceding statement also removes all data from all indexes on `employees` and returns the freed space to the tablespaces containing them.

Preserving Materialized View Logs After Truncate: Example

The following statements are examples of `TRUNCATE` statements that preserve materialized view logs:

```
TRUNCATE TABLE sales_demo PRESERVE MATERIALIZED VIEW LOG;

TRUNCATE TABLE orders_demo;
```

UPDATE

Purpose

Use the `UPDATE` statement to change existing values in a table or in the base table of a view or the master table of a materialized view.

Prerequisites

For you to update values in a table, the table must be in your own schema or you must have the `UPDATE` object privilege on the table.

For you to update values in the base table of a view:

- You must have the `UPDATE` object privilege on the view, and
- Whoever owns the schema containing the view must have the `UPDATE` object privilege on the base table.

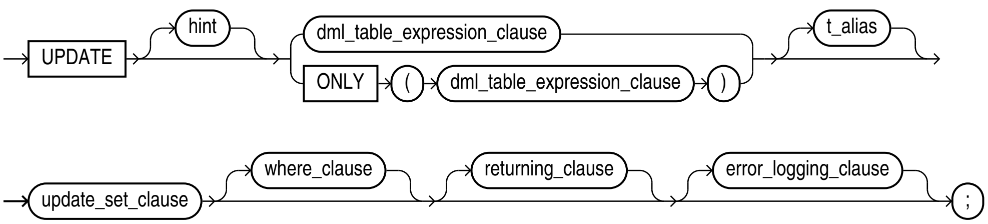
The `UPDATE ANY TABLE` system privilege also allows you to update values in any table or in the base table of any view.

To update values in an object on a remote database, you must also have the `READ` or `SELECT` object privilege on the object.

If the `SQL92_SECURITY` initialization parameter is set to `TRUE` and the `UPDATE` operation references table columns, such as the columns in a *`where_clause`*, then you must also have the `SELECT` object privilege on the object you want to update.

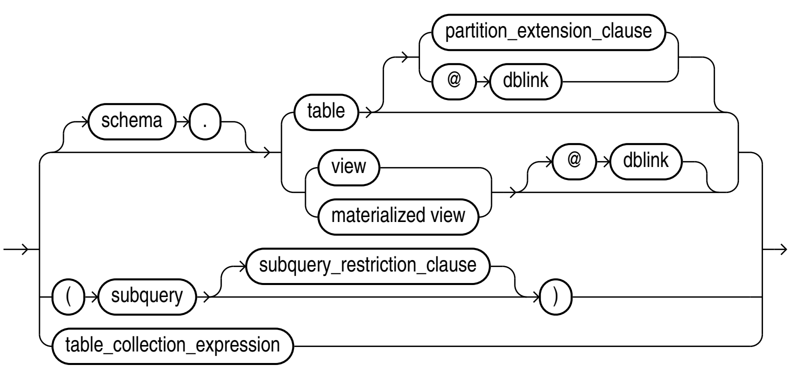
Syntax

***update*::=**



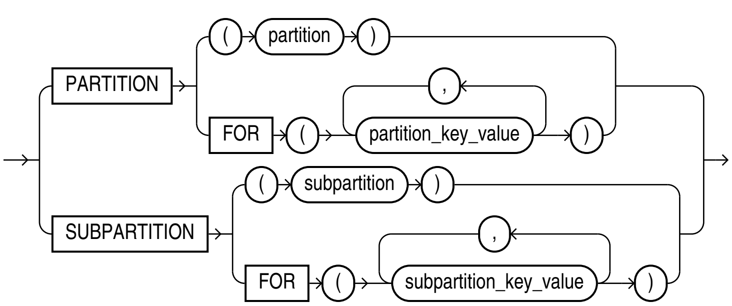
(*DML_table_expression_clause*::=, *update_set_clause*::=, *where_clause*::=, *returning_clause*::=, *error_logging_clause*::=)

***DML_table_expression_clause*::=**

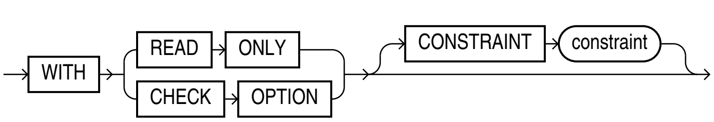


(*partition_extension_clause*::=, *subquery*::=--part of `SELECT`, *subquery_restriction_clause*::=, *table_collection_expression*::=)

***partition_extension_clause*::=**

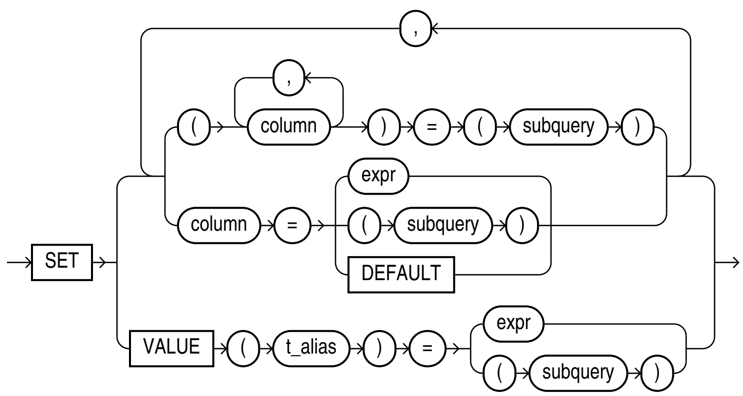


***subquery_restriction_clause*::=**

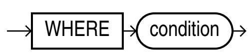


***table_collection_expression*::=**

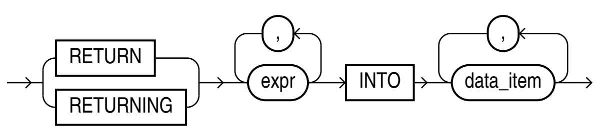
***update_set_clause*::=**



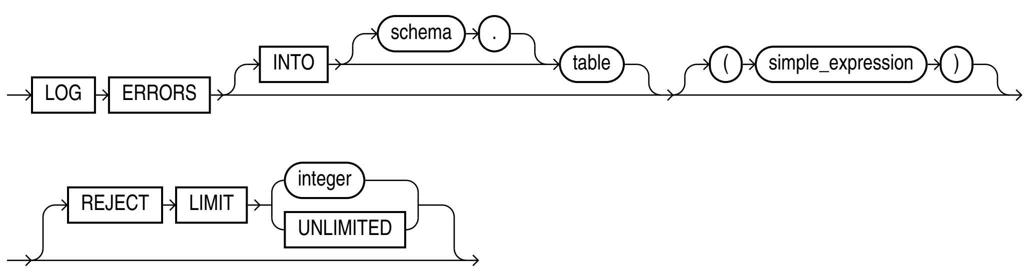
***where_clause*::=**



***returning_clause*::=**



***error_logging_clause*::=**



Semantics

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

You can place a parallel hint immediately after the `UPDATE` keyword to parallelize both the underlying scan and `UPDATE` operations.

> **See Also:**
>
> - "Hints " for the syntax and description of hints
> - *Oracle Database Concepts* for detailed information about parallel execution

DML_table_expression_clause

The `ONLY` clause applies only to views. Specify `ONLY` syntax if the view in the `UPDATE` clause is a view that belongs to a hierarchy and you do not want to update rows from any of its subviews.

> **See Also:**
>
> "Restrictions on the DML_table_expression_clause" and "Updating a Table: Examples"

schema

Specify the schema containing the object to be updated. If you omit *`schema`*, then the database assumes the object is in your own schema.

table | *view* | *materialized_view* |*subquery*

Specify the name of the table, view, materialized view, or the columns returned by a subquery to be updated. Issuing an `UPDATE` statement against a table fires any `UPDATE` triggers associated with the table.

- If you specify *`view`*, then the database updates the base table of the view. You cannot update a view except with `INSTEAD OF` triggers if the defining query of the view contains one of the following constructs:

  A set operator
  A `DISTINCT` operator
  An aggregate or analytic function
  A `GROUP BY`, `ORDER BY`, `MODEL`, `CONNECT BY`, or `START WITH` clause
  A collection expression in a `SELECT` list
  A subquery in a `SELECT` list
  A subquery designated `WITH READ ONLY`
  A recursive `WITH` clause

Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

- You cannot update more than one base table through a view.
- In addition, if the view was created with the `WITH CHECK OPTION`, then you can update the view only if the resulting data satisfies the view's defining query.
- If *`table`* or the base table of *`view`* contains one or more domain index columns, then this statement executes the appropriate indextype update routine.
- You cannot update rows in a read-only materialized view. If you update rows in a writable materialized view, then the database updates the rows from the underlying container table. However, the updates are overwritten at the next refresh operation. If you update rows in an updatable materialized view that is part of a materialized view group, then the database also updates the corresponding rows in the master table.

> **See Also:**
>
> - *Oracle Database Data Cartridge Developer's Guide* for more information on the indextype update routines
> - CREATE MATERIALIZED VIEW for information on creating updatable materialized views

_partition_extension_clause_

Specify the name or partition key value of the partition or subpartition within *`table`* targeted for updates. You need not specify the partition name when updating values in a partitioned table. However in some cases specifying the partition name can be more efficient than a complicated *`where_clause`*.

> **See Also:**
>
> "References to Partitioned Tables and Indexes " and "Updating a Partition: Example"

dblink

Specify a complete or partial name of a database link to a remote database where the object is located. You can use a database link to update a remote object only if you are using Oracle Database distributed functionality.

If you omit *`dblink`*, then the database assumes the object is on the local database.

Note:

Starting with Oracle Database 12*c* Release 2 (12.2), the `UPDATE` statement accepts remote LOB locators as bind variables. Refer to the “Distributed LOBs” chapter in *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

See Also:

"References to Objects in Remote Databases " for information on referring to database links

subquery_restriction_clause

Use the *`subquery_restriction_clause`* to restrict the subquery in one of the following ways:

WITH READ ONLY

Specify `WITH READ ONLY` to indicate that the table or view cannot be updated.

WITH CHECK OPTION

Specify `WITH CHECK OPTION` to indicate that Oracle Database prohibits any changes to the table or view that would produce rows that are not included in the subquery. When used in the subquery of a DML statement, you can specify this clause in a subquery in the `FROM` clause but not in subquery in the `WHERE` clause.

CONSTRAINT *constraint*

Specify the name of the `CHECK OPTION` constraint. If you omit this identifier, then Oracle automatically assigns the constraint a name of the form `SYS_C`*`n`*, where n is an integer that makes the constraint name unique within the database.

See Also:

"Using the WITH CHECK OPTION Clause: Example"

table_collection_expression

The *`table_collection_expression`* lets you inform Oracle that the value of *`collection_expression`* should be treated as a table for purposes of query and DML operations. The *`collection_expression`* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value—that is, a value whose type is nested table or varray. This process of extracting the elements of a collection is called **collection unnesting**.

The optional plus (+) is relevant if you are joining the `TABLE` collection expression with the parent table. The + creates an outer join of the two, so that the query returns rows from the outer table even if the collection expression is null.

> **Note:**
>
> In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as THE *subquery*. That usage is now deprecated.

You can use a *table_collection_expression* to update rows in one table based on rows from another table. For example, you could roll up four quarterly sales tables into a yearly sales table.

t_alias

Specify a **correlation name** (alias) for the table, view, or subquery to be referenced elsewhere in the statement. This alias is required if the *DML_table_expression_clause* references any object type attributes or object type methods.

> **See Also:**
>
> "Correlated Update: Example"

Restrictions on the *DML_table_expression_clause*

This clause is subject to the following restrictions:

- You cannot execute this statement if *table* or the base table of *view* contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- You cannot specify the *order_by_clause* in the subquery of the *DML_table_expression_clause*.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the UPDATE statement will fail unless the SKIP_UNUSABLE_INDEXES session parameter has been set to TRUE.

> **See Also:**
>
> ALTER SESSION for information on the SKIP_UNUSABLE_INDEXES session parameter

update_set_clause

The *update_set_clause* lets you set column values.

column

Specify the name of a column of the object that is to be updated. If you omit a column of the table from the *update_set_clause*, then the value of that column remains unchanged.

If *column* refers to a LOB object attribute, then you must first initialize it with a value of empty or null. You cannot update it with a literal. Also, if you are updating a LOB value using some method other than a direct UPDATE SQL statement, then you must first lock the row containing the LOB. See *for_update_clause* for more information.

If *column* is a virtual column, you cannot specify it here. Rather, you must update the values from which the virtual column is derived.

If *column* is part of the partitioning key of a partitioned table, then UPDATE will fail if you change a value in the column that would move the row to a different partition or subpartition, unless you enable row movement. Refer to the *row_movement_clause* of CREATE TABLE or ALTER TABLE.

In addition, if *column* is part of the partitioning key of a list-partitioned table, then UPDATE will fail if you specify a value for the column that does not already exist in the *partition_key_value* list of one of the partitions.

subquery

Specify a subquery that returns exactly one row for each row updated.

- If you specify only one column in the *update_set_clause*, then the subquery can return only one value.
- If you specify multiple columns in the *update_set_clause*, then the subquery must return as many values as you have specified columns.
- If the subquery returns no rows, then the column is assigned a null.
- If this *subquery* refers to remote objects, then the UPDATE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *DML_table_expression_clause* refers to any remote objects, then the UPDATE operation will run serially without notification.

You can use the *flashback_query_clause* within the subquery to update *table* with past data. Refer to the *flashback_query_clause* of SELECT for more information on this clause.

> **See Also:**
>
> - SELECT and "Using Subqueries "
> - parallel_clause in the CREATE TABLE documentation

expr

Specify an expression that resolves to the new value assigned to the corresponding column.

> **Note:**
>
> Expressions for the syntax of *expr* and "Updating an Object Table: Example"

DEFAULT

Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, then the database sets the column to null.

Restriction on Updating to Default Values

You cannot specify `DEFAULT` if you are updating a view.

You cannot use the `DEFAULT` clause in an `UPDATE` statement if the table that you are specifying has an Oracle Label Security policy enabled.

VALUE Clause

The `VALUE` clause lets you specify the entire row of an object table.

Restriction on the VALUE clause

You can specify this clause only for an object table.

> **Note:**
>
> If you insert string literals into a `RAW` column, then during subsequent queries, Oracle Database will perform a full table scan rather than using any index that might exist on the `RAW` column.

> **See Also:**
>
> "Updating an Object Table: Example"

where_clause

The *`where_clause`* lets you restrict the rows updated to those for which the specified *`condition`* is true. If you omit this clause, then the database updates all rows in the table or view. Refer to Conditions for the syntax of *`condition`*.

The *`where_clause`* determines the rows in which values are updated. If you do not specify the *`where_clause`*, then all rows are updated. For each row that satisfies the *`where_clause`*, the columns to the left of the equality operator (=) in the *`update_set_clause`* are set to the values of the corresponding expressions to the right of the operator. The expressions are evaluated as the row is updated.

returning_clause

The returning clause retrieves the rows affected by a DML statement. You can specify this clause for tables and materialized views and for views with a single base table.

When operating on a single row, a DML statement with a *`returning_clause`* can retrieve column expressions using the affected row, rowid, and `REF`s to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the `returning_clause` stores values from expressions, rowids, and `REF`s involving the affected rows in bind arrays.

expr

Each item in the `expr` list must be a valid expression syntax.

INTO

The `INTO` clause indicates that the values of the changed rows are to be stored in the variable(s) specified in `data_item` list.

data_item

Each `data_item` is a host variable or PL/SQL variable that stores the retrieved `expr` value.

For each expression in the `RETURNING` list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the `INTO` list.

Restrictions

The following restrictions apply to the `RETURNING` clause:

- The `expr` is restricted as follows:
  - For `UPDATE` and `DELETE` statements each `expr` must be a simple expression or a single-set aggregate function expression. You cannot combine simple expressions and single-set aggregate function expressions in the same `returning_clause`. For `INSERT` statements, each `expr` must be a simple expression. Aggregate functions are not supported in an `INSERT` statement `RETURNING` clause.
  - Single-set aggregate function expressions cannot include the `DISTINCT` keyword.
- If the `expr` list contains a primary key column or other `NOT NULL` column, then the update statement fails if the table has a `BEFORE UPDATE` trigger defined on it.
- You cannot specify the `returning_clause` for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve `LONG` types with this clause.
- You cannot specify this clause for a view on which an `INSTEAD OF` trigger has been defined.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information on using the `BULK COLLECT` clause to return multiple values to collection variables

error_logging_clause

The `error_logging_clause` has the same behavior in an `UPDATE` statement as it does in an `INSERT` statement. Refer to the `INSERT` statement *error_logging_clause* for more information.

> **See Also:**
>
> "Inserting Into a Table with Error Logging: Example"

Examples

Updating a Table: Examples

The following statement gives null commissions to all employees with the job `SH_CLERK`:

```
UPDATE employees
   SET commission_pct = NULL
   WHERE job_id = 'SH_CLERK';
```

The following statement promotes Douglas Grant to manager of Department 20 with a $1,000 raise:

```
UPDATE employees SET
    job_id = 'SA_MAN', salary = salary + 1000, department_id = 120
    WHERE first_name||' '||last_name = 'Douglas Grant';
```

The following statement increases the salary of an employee in the `employees` table on the `remote` database:

```
UPDATE employees@remote
   SET salary = salary*1.1
   WHERE last_name = 'Baer';
```

The next example shows the following syntactic constructs of the `UPDATE` statement:

- Both forms of the *`update_set_clause`* together in a single statement
- A correlated subquery
- A *`where_clause`* to limit the updated rows

```
UPDATE employees a
    SET department_id =
        (SELECT department_id
            FROM departments
            WHERE location_id = '2100'),
        (salary, commission_pct) =
        (SELECT 1.1*AVG(salary), 1.5*AVG(commission_pct)
          FROM employees b
          WHERE a.department_id = b.department_id)
    WHERE department_id IN
        (SELECT department_id
          FROM departments
          WHERE location_id = 2900
              OR location_id = 2700);
```

The preceding `UPDATE` statement performs the following operations:

- Updates only those employees who work in Geneva or Munich (locations 2900 and 2700)
- Sets `department_id` for these employees to the `department_id` corresponding to Bombay (`location_id` 2100)

- Sets each employee's salary to 1.1 times the average salary of their department
- Sets each employee's commission to 1.5 times the average commission of their department

Updating a Partition: Example

The following example updates values in a single partition of the `sales` table:

```
UPDATE sales PARTITION (sales_q1_1999) s
   SET s.promo_id = 494
   WHERE amount_sold > 1000;
```

Updating an Object Table: Example

The following statement creates two object tables, `people_demo1` and `people_demo2`, of the `people_typ` object created in Table Collections: Examples. The example shows how to update a row of `people_demo1` by selecting a row from `people_demo2`:

```
CREATE TABLE people_demo1 OF people_typ;

CREATE TABLE people_demo2 OF people_typ;

UPDATE people_demo1 p SET VALUE(p) =
   (SELECT VALUE(q) FROM people_demo2 q
    WHERE p.department_id = q.department_id)
   WHERE p.department_id = 10;
```

The example uses the `VALUE` object reference function in both the `SET` clause and the subquery.

Correlated Update: Example

For an example that uses a correlated subquery to update nested table rows, refer to "Table Collections: Examples".

Using the RETURNING Clause During UPDATE: Example

The following example returns values from the updated row and stores the result in PL/SQL variables `bnd1`, `bnd2`, `bnd3`:

```
UPDATE employees
  SET job_id ='SA_MAN', salary = salary + 1000, department_id = 140
  WHERE last_name = 'Jones'
  RETURNING salary*0.25, last_name, department_id
    INTO :bnd1, :bnd2, :bnd3;
```

The following example shows that you can specify a single-set aggregate function in the expression of the returning clause:

```
UPDATE employees
   SET salary = salary * 1.1
   WHERE department_id = 100
   RETURNING SUM(salary) INTO :bnd1;
```

A How to Read Syntax Diagrams

This appendix describes how to read syntax diagrams.

This reference presents Oracle SQL syntax in both graphic diagrams and in text (Backus-Naur Form—BNF). This appendix contains these sections:

- Graphic Syntax Diagrams
- Backus-Naur Form Syntax

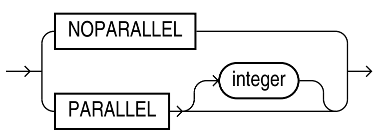
Graphic Syntax Diagrams

Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

Commands and other keywords appear in UPPERCASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lowercase inside ovals. Variables are used for the parameters. Punctuation, operators, delimiters, and terminators appear inside circles.

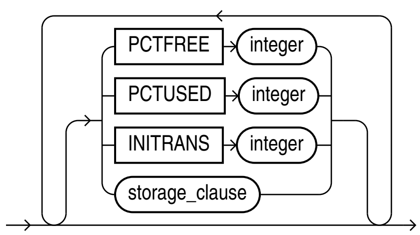
If the syntax diagram has more than one path, then you can choose any path. For example, in the following syntax you can specify either `NOPARALLEL` or `PARALLEL`:

***parallel_clause*::=**



If you have the choice of more than one keyword, operator, or parameter, then your options appear in a vertical list. For example, in the following syntax diagram, you can specify one or more of the four parameters in the stack:

***physical_attributes_clause*::=**



The following table shows parameters that appear in the syntax diagrams and provides examples of the values you might substitute for them in your statements:

Table A-1 Syntax Parameters

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter. For a list of all types of objects, see the section, "Schema Objects ".	employees
c	The substitution value must be a single character from your database character set.	T s
'text'	The substitution value must be a text string in single quotation marks. See the syntax description of '*text*' in "Text Literals ".	'Employee records'
char	The substitution value must be an expression of data type CHAR or VARCHAR2 or a character literal in single quotation marks.	last_name 'Smith'
condition	The substitution value must be a condition that evaluates to TRUE or FALSE. See the syntax description of *condition* in Conditions.	last_name >'A'
date *d*	The substitution value must be a date constant or an expression of DATE data type.	TO_DATE('01-Jan-2002', 'DD-MON-YYYY')
expr	The substitution value can be an expression of any data type as defined in the syntax description of *expr* in "About SQL Expressions ".	salary + 1000
integer	The substitution value must be an integer as defined by the syntax description of integer in "Integer Literals ".	72
number *m* *n*	The substitution value must be an expression of NUMBER data type or a number constant as defined in the syntax description of *number* in "Numeric Literals ".	AVG(salary) 15 * 7
raw	The substitution value must be an expression of data type RAW.	HEXTORAW('7D')
subquery	The substitution value must be a SELECT statement that will be used in another SQL statement. See SELECT .	SELECT last_name FROM employees
db_name	The substitution value must be the name of a nondefault database in an embedded SQL program.	sales_db
db_string	The substitution value must be the database identification string for an Oracle Net database connection. For details, see the user's guide for your specific Oracle Net protocol.	—

Required Keywords and Parameters

Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the main path, which is the horizontal line you are currently traveling. In the following example, `library_name` is a required parameter:

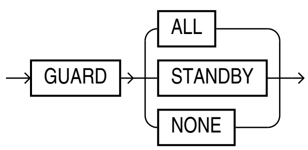
drop_library::=

If there is a library named `HQ_LIB`, then, according to the diagram, the following statement is valid:

```
DROP LIBRARY hq_lib;
```

If multiple keywords or parameters appear in a vertical list that intersects the main path, then one of them is required. You must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose `ALL`, `STANDBY`, or `NONE`:

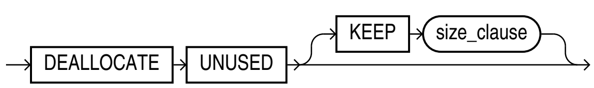
security_clause::=



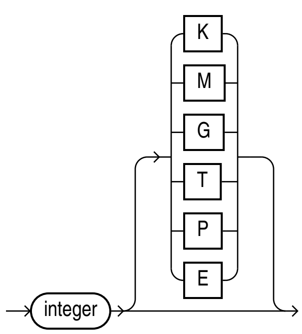
Optional Keywords and Parameters

If keywords and parameters appear in a vertical list above the main path, then they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:

deallocate_unused_clause::=



size_clause::=



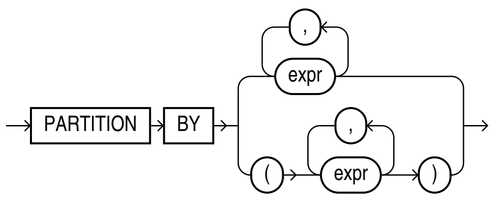
According to the diagrams, all of the following statements are valid:

```
DEALLOCATE UNUSED;
DEALLOCATE UNUSED KEEP 1000;
DEALLOCATE UNUSED KEEP 10G;
DEALLOCATE UNUSED 8T;
```

Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, after choosing one value expression, you can go back repeatedly to choose another, separated by commas.

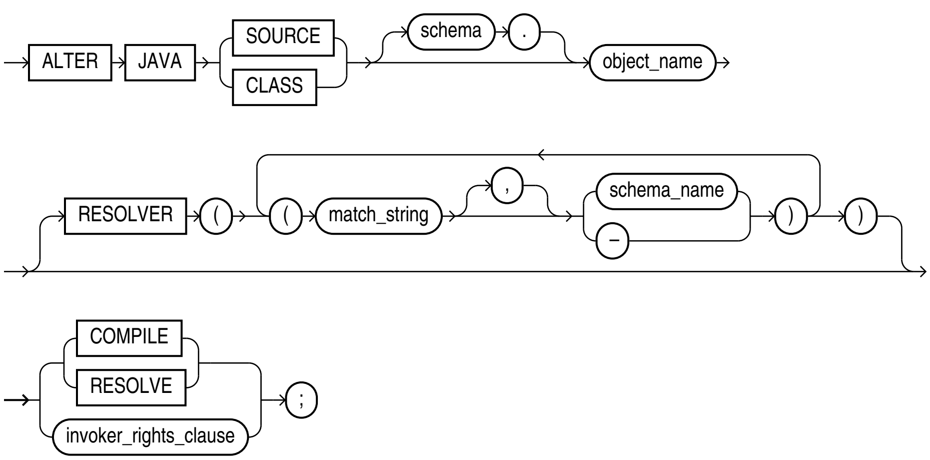
query_partition_clause::=



Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end to end. The following example is a three-part diagram:

alter_java::=



According to the diagram, the following statement is valid:

```
ALTER JAVA SOURCE jsource_1 COMPILE;
```

Backus-Naur Form Syntax

Each graphic syntax diagram in this reference is followed by a link to a text description of the graphic. The text descriptions consist of a simple variant of Backus-Naur Form (BNF) that includes the following symbols and conventions:

Symbol or Convention	Meaning
[ ]	Brackets enclose optional items.
{ }	Braces enclose items only one of which is required.
|	A vertical bar separates alternatives within brackets or braces.
...	Ellipsis points show that the preceding syntactic element can be repeated.
delimiters	Delimiters other than brackets, braces, vertical bars, and ellipses must be entered as shown.
boldface	Words appearing in boldface are keywords. They must be typed as shown. (Keywords are case-sensitive in some, but not all, operating systems.) Words that are not in boldface are placeholders for which you must substitute a name or value.

B Automatic and Manual Locking Mechanisms During SQL Operations

This appendix describes mechanisms that lock data either automatically or as specified by the user during SQL statements. For a general discussion of locking mechanisms in the context of data concurrency and consistency, see *Oracle Database Concepts*.

This appendix contains the following sections:

- Automatic Locks in DML Operations
- Automatic Locks in DDL Operations
- Manual Data Locking

Automatic Locks in DML Operations

The purpose of a DML lock, also called a **data lock**, is to guarantee the integrity of data being accessed concurrently by multiple users. For example, a DML lock can prevent multiple customers from buying the last copy of a book available from an online bookseller. DML locks prevent destructive interference of simultaneous conflicting DML or DDL operations.

DML statements automatically acquire locks at both the table level and the row level. In the sections that follow, the acronym in parentheses after each type of lock or lock mode is the abbreviation used in the Locks Monitor of Oracle Enterprise Manager. Enterprise Manager might display "TM" for any table lock, rather than indicate the mode of table lock (such as RS or SRX).

The types of row and table locks are summarized here. For a more complete discussion of the types of row and table locks, see *Oracle Database Concepts*.

Row Locks (TX)

A **row lock**, also called a **TX lock**, is a lock on a single row of a table. A transaction acquires a row lock for each row modified by one of the following statements: `INSERT`, `UPDATE`, `DELETE`, `MERGE`, and `SELECT` ... `FOR UPDATE`. The row lock exists until the transaction commits or rolls back.

When a transaction obtains a row lock for a row, the transaction also acquires a table lock for the table in which the row resides. The table lock prevents conflicting DDL operations that would override data changes in a current transaction.

Table Locks (TM)

A transaction automatically acquires a table lock (**TM lock**) when a table is modified with the following statements: `INSERT`, `UPDATE`, `DELETE`, `MERGE`, and `SELECT` ... `FOR UPDATE`. These DML operations require table locks to reserve DML access to the table on behalf of a transaction and to prevent DDL operations that would conflict with the transaction. You can explicitly obtain a table lock using the `LOCK TABLE` statement, as described in "Manual Data Locking".

A table lock can be held in any of the following modes:

- A **row share lock (RS)**, also called a **subshare table lock (SS)**, indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them. An SS lock is the least restrictive mode of table lock, offering the highest degree of concurrency for a table.
- A **row exclusive lock (RX)**, also called a **subexclusive table lock (SX)**, indicates that the transaction holding the lock has updated table rows or issued `SELECT ... FOR UPDATE`. An SX lock allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table. Therefore, SX locks allow multiple transactions to obtain simultaneous SX and SS locks for the same table.
- A **share table lock (S)** held by one transaction allows other transactions to query the table (without using `SELECT ... FOR UPDATE`) but allows updates only if a single transaction holds the share table lock. Multiple transactions may hold a share table lock concurrently, so holding this lock is not sufficient to ensure that a transaction can modify the table.
- A **share row exclusive table lock (SRX)**, also called a **share-subexclusive table lock (SSX)**, is more restrictive than a share table lock. Only one transaction at a time can acquire an SSX lock on a given table. An SSX lock held by a transaction allows other transactions to query the table (except for `SELECT ... FOR UPDATE`) but not to update the table.
- An **exclusive table lock (X)** is the most restrictive mode of table lock, allowing the transaction that holds the lock exclusive write access to the table. Only one transaction can obtain an X lock for a table.

> **See Also:**
>
> "Manual Data Locking"

Locks in DML Operations

Oracle Database automatically obtains row-level and table-level locks on behalf of DML operations. The type of operation determines the locking behavior. Table B-1 summarizes the information in this section.

> **Note:**
>
> The implicit SX locks shown for the DML statements in Table B-1 can sometimes be exclusive (X) locks for a short time owing to side effects from constraints.

Table B-1 Summary of Locks Obtained by DML Statements

SQL Statement	Row Locks	Table Lock Mode	RS	RX	S	SRX	X
`SELECT ... FROM` *table*...	—	none	Y	Y	Y	Y	Y
`INSERT INTO` *table* ...	Yes	SX	Y	Y	N	N	N

Table B-1 (Cont.) Summary of Locks Obtained by DML Statements

SQL Statement	Row Locks	Table Lock Mode	RS	RX	S	SRX	X
`UPDATE` *`table`* ...	Yes	SX	Y[1]	Y[1]	N	N	N
`MERGE INTO` *`table`* ...	Yes	SX	Y	Y	N	N	N
`DELETE FROM` *`table`* ...	Yes	SX	Y[1]	Y[1]	N	N	N
`SELECT` ... `FROM` *`table`* `FOR UPDATE OF` ...	Yes	SX	Y[1]	Y[1]	N	N	N
`LOCK TABLE` *`table`* `IN` ...	—						
`ROW SHARE MODE`		SS	Y	Y	Y	Y	N
`ROW EXCLUSIVE MODE`		SX	Y	Y	N	N	N
`SHARE MODE`		S	Y	N	Y	N	N
`SHARE ROW EXCLUSIVE MODE`		SSX	Y	N	N	N	N
`EXCLUSIVE MODE`		X	N	N	N	N	N

1 Yes, if no conflicting row locks are held by another transaction. Otherwise, waits occur.

Locks When Rows Are Queried

A query can be explicit, as in the `SELECT` statement, or implicit, as in most `INSERT`, `MERGE`, `UPDATE`, and `DELETE` statements. The only DML statement that does not necessarily include a query component is an `INSERT` statement with a `VALUES` clause. Because queries only read data, they are the SQL statements least likely to interfere with other SQL statements.

The following characteristics apply to a query *without* the `FOR UPDATE` clause:

- The query acquires no data locks. Therefore, other transactions can query and update a table being queried, including the specific rows being queried. Because queries without the `FOR UPDATE` clause do not acquire any data locks to block other operations, such queries are often referred to as **nonblocking queries**.
- The query does not have to wait for any data locks to be released. Therefore, the query can always proceed. An exception to this rule is that queries may have to wait for data locks in some very specific cases of pending distributed transactions.

Locks When Rows Are Modified

Some databases use a lock manager to maintain a list of locks in memory. Oracle Database, in contrast, stores lock information in the data block that contains the locked row. Each row lock affects only a single row.

Oracle Database uses a queuing mechanism for acquisition of row locks. If a transaction requires a row lock, and if the row is not already locked, then the transaction acquires a lock in the row's data block. The transaction itself has an entry in the **interested transaction list (ITL)** section of the block header. Each row modified by this transaction points to a copy of the transaction ID stored in the ITL. Thus, 100 rows in the same block modified by a single transaction require 100 row locks, but all 100 rows reference a single transaction ID.

When a transaction ends, the transaction ID remains in the ITL section of the data block header. If a new transaction wants to modify a row, then it uses the transaction ID to determine whether the lock is active. If the lock is active, then the session of the new transaction asks to be notified when the lock is released; otherwise, the new transaction acquires the lock.

The characteristics of `INSERT`, `UPDATE`, `DELETE`, and `SELECT ... FOR UPDATE` statements are as follows:

- A transaction containing a DML statement acquires exclusive row locks on the rows modified by the statement. Therefore, other transactions cannot update or delete the locked rows until the locking transaction either commits or rolls back.
- In addition to these row locks, a transaction containing a DML statement that modifies data also requires at least a subexclusive table lock (SX) on the table that contains the affected rows. If the transaction already holds an S, SRX, or X table lock for the table, which are more restrictive than an SX lock, then the SX lock is not needed and is not acquired. If the containing transaction already holds only an SS lock, however, then Oracle Database automatically converts the SS lock to an SX lock.
- A transaction that contains a DML statement does not require row locks on any rows selected by a subquery or an implicit query.

  In the following sample `UPDATE` statement, the `SELECT` statement in parentheses is a subquery, whereas the `WHERE a > 5` clause is an implicit query:

  ```
  UPDATE t SET x = ( SELECT y FROM t2 WHERE t2.z = t.z ) WHERE a > 5;
  ```

  A subquery or implicit query inside a DML statement is guaranteed to be consistent as of the start of the query and does not see the effects of the DML statement of which it forms a part.
- A query in a transaction can see the changes made by previous DML statements in the same transaction, but not the uncommitted changes of other transactions.

See Also:

Oracle Database Concepts for information on locks in foreign keys

Automatic Locks in DDL Operations

A **data dictionary (DDL) lock** protects the definition of a schema object while it is acted upon or referred to by an ongoing DDL operation. For example, when a user creates a procedure, Oracle Database automatically acquires DDL locks for all schema objects referenced in the procedure definition. The DDL locks prevent these objects from being altered or dropped before procedure compilation is complete.

Oracle Database acquires a DDL lock automatically on behalf of any DDL transaction requiring it. Users cannot explicitly request DDL locks. Only individual schema objects that are modified or referenced are locked during DDL operations. The whole data dictionary is never locked.

DDL operations also acquire DML locks on the schema object to be modified.

Exclusive DDL Locks

An **exclusive DDL lock** prevents other session from obtaining a DDL or DML lock.

Most DDL operations require exclusive DDL locks to prevent destructive interference with other DDL operations that might modify or reference the same schema object. For example, a `DROP TABLE` operation is not allowed to drop a table while an `ALTER TABLE` operation is adding a column to it, and vice versa. However, a query against the table is not blocked.

Exclusive DDL locks last for the duration of DDL statement execution and automatic commit. During the acquisition of an exclusive DDL lock, if another DDL lock is already held on the schema object by another operation, then the acquisition waits until the older DDL lock is released and then proceeds.

Share DDL Locks

A **share DDL lock** for a resource prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.

For example, when a `CREATE PROCEDURE` statement is run, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table.

A share DDL lock lasts for the duration of DDL statement execution and automatic commit. Thus, a transaction holding a share DDL lock is guaranteed that the definition of the referenced schema object is constant for the duration of the transaction.

Breakable Parse Locks

A **parse lock** is held by a SQL statement or PL/SQL program unit for each schema object that it references. Parse locks are acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped. A parse lock is called a **breakable parse lock** because it does not disallow any DDL operation and can be broken to allow conflicting DDL operations.

A **parse lock** is acquired in the shared pool during the parse phase of SQL statement execution. The lock is held as long as the shared SQL area for that statement remains in the shared pool.

Manual Data Locking

Oracle Database always performs locking automatically to ensure data concurrency, data integrity, and statement-level read consistency. However, you can override the Oracle default locking mechanisms. This can be useful in situations such as the following:

- When your application requires consistent data for the duration of the transaction, not reflecting changes by other transactions, you can achieve transaction-level read consistency by using explicit locking, read-only transactions, serializable transactions, or by overriding default locking.
- When your application requires that a transaction have exclusive access to a resource so that the transaction does not have to wait for other transactions to complete, you can explicitly lock the data for the duration of the transaction.

You can override automatic locking at two levels:

- **Transaction**. You can override transaction-level locking with the following SQL statements:
  - `SET TRANSACTION ISOLATION LEVEL`

  - `LOCK TABLE`
  - `SELECT` ... `FOR UPDATE`

  Locks acquired by these statements are released after the transaction commits or rolls back.

- **Session**. A session can set the required transaction isolate level with an `ALTER SESSION SET ISOLATION LEVEL` statement.

> **Note:**
>
> When overriding Oracle default locking, the database administrator or application developer should ensure that data integrity is guaranteed, data concurrency is acceptable, and deadlocks are not possible or, if possible, are appropriately handled. For more information on these criteria, see *Oracle Database Concepts*.

C

Oracle and Standard SQL

This appendix declares Oracle's conformance to the SQL standards established by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO).

The ISO SQL standard consists of nine parts (SQL/Framework, SQL/Foundation, SQL/CLI, SQL/PSM, SQL/MED, SQL/OLB, SQL/Schemata, SQL/JRT, and SQL/XML). The ANSI SQL standard consists of the same nine parts.

The mandatory portion of SQL is known as Core SQL and is found in SQL:2016 Part 2 (Foundation) and Part 11 (Schemata). The Foundation features are analyzed in Annex F of Part 2 in the table "Feature taxonomy and definition for mandatory features." The Schemata features are analyzed in Annex F of Part 11 in the table "Feature taxonomy and definition for mandatory features."

This appendix contains the following sections:

- ANSI Standards
- ISO Standards
- Oracle Compliance to Core SQL
- Oracle Support for Optional Features of SQL/Foundation
- Oracle Compliance with SQL/CLI
- Oracle Compliance with SQL/PSM
- Oracle Compliance with SQL/MED
- Oracle Compliance with SQL/OLB
- Oracle Compliance with SQL/JRT
- Oracle Compliance with SQL/XML
- Oracle Compliance with FIPS 127-2
- Oracle Extensions to Standard SQL
- Oracle Compliance with Older Standards
- Character Set Support

ANSI Standards

The following documents of the American National Standards Institute (ANSI) relate to SQL:

- INCITS/ANSI/ISO/IEC 9075-1:2016, Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework)
- INCITS/ANSI/ISO/IEC 9075-2:2016, Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)
- INCITS/ANSI/ISO/IEC 9075-3:2016, Information technology—Database languages—SQL—Part 3: Call-Level Interface (SQL/CLI)

- INCITS/ANSI/ISO/IEC 9075-4:2016, Information technology—Database languages —SQL—Part 4: Persistent Stored Modules (SQL/PSM)
- INCITS/ANSI/ISO/IEC 9075-9:2016, Information technology—Database languages —SQL—Part 9: Management of External Data (SQL/MED)
- INCITS/ANSI/ISO/IEC 9075-10:2016, Information technology—Database languages—SQL—Part 10: Object Language Bindings (SQL/OLB)
- INCITS/ANSI/ISO/IEC 9075-11:2016, Information technology—Database languages—SQL—Part 11: Information and Definition Schemas (SQL/Schemata)
- INCITS/ANSI/ISO/IEC 9075-13:2016, Information technology—Database languages—SQL—Part 13: SQL Routines and Types using the Java Programming Language (SQL/JRT)
- INCITS/ANSI/ISO/IEC 9075-14:2016, Information technology—Database languages—SQL—Part 14: XML-Related Specifications (SQL/XML)

These standards are identical to the corresponding ISO standards listed in the next section.

You can obtain a copy of ANSI standards from this address:

American National Standards Institute
25 West 43rd Street, fourth floor
New York, NY 10036 USA
Telephone: +1.212.642.4900
Fax: +1.212.398.0023
Web site: `http://www.ansi.org/`

You can also obtain the standards from their Web site:

`http://webstore.ansi.org/default.aspx`

A subset of ANSI standards, including the SQL standard, are INCITS standards. You can obtain these from the InterNational Committee for Information Technology Standards (INCITS) at:

`http://www.incits.org/`

ISO Standards

The following documents of the International Organization for Standardization (ISO) relate to SQL:

- ISO/IEC 9075-1:2016, Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework)
- ISO/IEC 9075-2:2016, Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)
- ISO/IEC 9075-3:2016, Information technology—Database languages—SQL—Part 3: Call-Level Interface (SQL/CLI)
- ISO/IEC 9075-4:2016, Information technology—Database languages—SQL—Part 4: Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9:2016, Information technology—Database languages—SQL—Part 9: Management of External Data (SQL/MED)

- ISO/IEC 9075-10:2016, Information technology—Database languages—SQL—Part 10: Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11:2016, Information technology—Database languages—SQL—Part 11: Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13:2016, Information technology—Database languages—SQL—Part 13: SQL Routines and Types using the Java Programming Language (SQL/JRT)
- ISO/IEC 9075-14:2016, Information technology—Database languages—SQL—Part 14: XML-Related Specifications (SQL/XML)

You can obtain a copy of ISO standards from this address:

International Organization for Standardization
1, ch. de la Voie-Creuse
Case postale 56
CH-1211, Geneva 20, Switzerland
Phone: +41.22.749.0111
Fax: +41.22.733.3430
Web site: http://www.iso.org/

or from their Web store:

`http://www.iso.org/iso/store.htm`

Oracle Compliance to Core SQL

The ANSI and ISO SQL standards require conformance claims to state the type of conformance and the implemented facilities. The minimum claim of conformance is called Core SQL and is defined in Part 2, SQL/Foundation, and Part 11, SQL/Schemata, of the standard. The following products provide full or partial conformance with Core SQL as described in the tables that follow:

- Oracle Database server, release 12.2
- OTT (Oracle Type Translator), release 12.2
- Pro*C/C++, release 12.2
- Pro*COBOL, release 12.2

The SQL standards conformance features can be used either as a guide to portability, or as a guide to functionality. From the standpoint of portability, the user is interested in conformance to both the precise syntax and semantics of the standard feature. From the standpoint of functionality, the user is less concerned about the precise syntax and more concerned with issues of semantics. The tables in this appendix use the following terms regarding support for standard syntax and semantics:

- Full Support: The feature is supported with standard syntax and semantics.
- Partial Support: Some, but not all, of the standard syntax is supported; whatever is supported has standard semantics.
- Enhanced Support: The standard semantics is supported, as well as additional functionality.
- Equivalent Support: The standard semantics is supported using non-standard syntax.
- Similar Support: Neither the standard's syntax nor semantics are supported precisely, but similar functionality is provided.

Oracle's support for the features of Core SQL is listed in Table C-1:

Table C-1 Oracle Support of Core SQL Features

Feature ID, Feature	Support
E011, Numeric data types	Oracle fully supports this feature.
E021, Character data types	Oracle fully supports these subfeatures: • E021-01, `CHARACTER` data type • E021-07, Character concatenation • E021-08, `UPPER` and `LOWER` functions • E021-09, `TRIM` function • E021-10, Implicit casting among character data types Oracle partially supports these subfeatures: • E021-02, `CHARACTER VARYING` data type (Oracle does not distinguish a zero-length `VARCHAR` string from `NULL`) • E021-03, Character literals (Oracle regards the zero-length literal '' as being null) • E021-12, Character comparison (Oracle's rules for padding the shorter of two strings to be compared differs from the standard) Oracle has equivalent functionality for these subfeatures: • E021-04, `CHARACTER_LENGTH` function: use `LENGTH` function instead • E021-05, `OCTET_LENGTH` function: use `LENGTHB` function instead • E021-06, `SUBSTRING` function: use `SUBSTR` function instead • E021-11, `POSITION` function: use `INSTR` function instead
E031, Identifiers	Oracle supports this feature, with the following exceptions: • Oracle does not support the escape sequence to permit a double quote within a quoted identifier • A non-quoted identifier may not be equivalent to an Oracle reserved word (the list of Oracle reserved words differs from the standard's list) • A column name may not be `ROWID`, even as a quoted identifier Oracle extends this feature as follows: • An identifier may be up to 128 characters long • A non-quoted identifier may have dollar sign ($) or pound sign (#)
E051, Basic query specification	Oracle fully supports the following subfeatures: • E051-01, `SELECT DISTINCT` • E051-02, `GROUP BY` clause • E051-04, `GROUP BY` can contain columns not in `SELECT` list • E051-05, `SELECT` list items can be renamed • E051-06, `HAVING` clause • E051-07, Qualified * in `SELECT` list Oracle partially supports the following subfeatures: • E051-08, Correlation names in `FROM` clause (Oracle supports correlation names, but not the optional `AS` keyword) Oracle has equivalent functionality for the following subfeature: • E051-09, Rename columns in the `FROM` clause (column names can be renamed in a subquery in the `FROM` clause)

Table C-1 (Cont.) Oracle Support of Core SQL Features

Feature ID, Feature	Support
E061, Basic predicates and search conditions	Oracle fully supports this feature, except that Oracle comparison of character strings differs from the standard as follows: In the standard, two character strings of unequal length are compared by either padding the shorter string with spaces or a fictitious character that is less than all actual characters. The decision on padding is made on the basis of the character set. In Oracle, the decision is based on whether the comparands are of fixed or varying length.
E071, Basic query expressions	Oracle fully supports the following subfeatures: • E071-01, `UNION DISTINCT` table operator • E071-02, `UNION ALL` table operator • E071-05, Columns combined by table operators need not have exactly the same type • E071-06, table operators in subqueries Oracle has equivalent functionality for the following subfeature: • E071-03, `EXCEPT DISTINCT` table operator: Use `MINUS` instead of `EXCEPT DISTINCT`
E081, Basic privileges	Oracle fully supports all subfeatures of this feature, except E081-09, `USAGE` privileges. In the standard, the `USAGE` privilege permits the user to use domains, collations, character sets, transliterations, user-defined types and sequence generators. Oracle does not support domains or transliterations. No privileges are required to access collations and character sets. The Oracle privilege to use a user-defined type is `EXECUTE`. The Oracle privilege to use a sequence type is `SELECT`.
E091, Set functions	Oracle fully supports this feature.
E101, Basic data manipulation	Oracle fully supports this feature.
E111, Single row `SELECT` statement	Oracle fully supports this feature.
E121, Basic cursor support	Oracle fully supports the following subfeatures: • E121-02, `ORDER BY` columns need not be in `SELECT` list • E121-03, Value expressions in `ORDER BY` clause • E121-04, `OPEN` statement • E121-06, Positioned `UPDATE` statement • E121-07, Positioned `DELETE` statement • E121-08, `CLOSE` statement Oracle provides partial support for the following subfeatures: • E121-01, `DECLARE CURSOR` - fully supported, except for the `FOR READ ONLY` syntax • E121-10 `FETCH` statement, implicit `NEXT` - fully supported, except for the noise word `FROM` Oracle provides enhanced support for the following subfeature: • E121-17, `WITH HOLD` cursors (in the standard, a cursor is not held through a `ROLLBACK`, but Oracle does hold through `ROLLBACK`)

Table C-1 (Cont.) Oracle Support of Core SQL Features

Feature ID, Feature	Support
E131, Null value support	Oracle fully supports this feature, with this exception: In Oracle, a null of character type is indistinguishable from a zero-length character string.
E141, Basic integrity constraints	Oracle fully supports this feature.
E151, Transaction support	Oracle fully supports this feature.
E152, Basic `SET TRANSACTION` statement	Oracle fully supports this feature.
E153, Updatable queries with subqueries	Oracle fully supports this feature.
E161, SQL comments using leading double minus	Oracle fully supports this feature.
E171, SQLSTATE support	Oracle fully supports this feature.
E182, Host language binding	Oracle fully supports this feature through Pro*C/C++ and Pro*COBOL
F021, Basic information schema	Oracle does not have any of the views in this feature. However, Oracle makes the same information available in other metadata views: • Instead of `TABLES`, use `ALL_TABLES`. • Instead of `COLUMNS`, use `ALL_TAB_COLUMNS`. • Instead of `VIEWS`, use `ALL_VIEWS`. However, Oracle's `ALL_VIEWS` does not display whether a user view was defined `WITH CHECK OPTION` or if it is updatable. To see whether a view has `WITH CHECK OPTION`, use `ALL_CONSTRAINTS`, with `TABLE_NAME` equal to the view name and look for `CONSTRAINT_TYPE` equal to '`V`'. • Instead of `TABLE_CONSTRAINTS`, `REFERENTIAL_CONSTRAINTS`, and `CHECK_CONSTRAINTS`, use `ALL_CONSTRAINTS`. However, Oracle's `ALL_CONSTRAINTS` does not display whether a constraint is deferrable or initially deferred.

Table C-1 (Cont.) Oracle Support of Core SQL Features

Feature ID, Feature	Support
F031, Basic schema manipulation	Oracle fully supports these subfeatures: • F031-01, `CREATE TABLE` statement to create persistent base tables • F031-02, `CREATE VIEW` statement • F031-03, `GRANT` statement Oracle provides equivalent support for this subfeature: • F031-04, `ALTER TABLE` statement: `ADD COLUMN` clause (Oracle does not support the optional keyword `COLUMN` in this syntax. Also, Oracle requires the column definition to be enclosed in parentheses, unlike the standard.) Oracle does not support these subfeatures (because Oracle does not support the keyword `RESTRICT`): • F031-13, `DROP TABLE` statement: `RESTRICT` clause • F031-16, `DROP VIEW` statement: `RESTRICT` clause • F031-19, `REVOKE` statement: `RESTRICT` clause (Oracle `DROP` commands enhance the standard by invalidating dependent objects, so that they can be subsequently revalidated without user action, rather than either cascading all drops to dependent objects or prohibiting a drop if there is a dependent object.)
F041, Basic joined table	Oracle fully supports this feature.
F051, Basic date and time	Oracle fully supports this feature, except the following subfeatures are not supported: • F051-02, `TIME` data type • F051-07, `LOCALTIME`
F081, `UNION` and `EXCEPT` in views	Oracle fully supports `UNION` in views. The equivalent in Oracle of the standard's `EXCEPT` is called `MINUS`, which is fully supported in views.
F131, Grouped operations	Oracle fully supports this feature.
F181, Multiple module support	Oracle fully supports this feature.
F201, `CAST` function	Oracle fully supports this feature.
F221, Explicit defaults	Oracle's `DEFAULT ON NULL` capability in a column definition provides equivalent functionality for the `INSERT` statement though not for the `UPDATE` statement.
F261, `CASE` expressions	Oracle fully supports this feature.
F311, Schema definition statement	Oracle fully supports this feature.
F471, Scalar subquery values	Oracle fully supports this feature.
F481, Expanded null predicate	Oracle fully supports this feature.

Table C-1 (Cont.) Oracle Support of Core SQL Features

Feature ID, Feature	Support
F501, Feature and conformance views	Oracle does not support this feature.
F812, Basic flagging	Oracle has a flagger, but it flags SQL-92 compliance rather than SQL:2011 compliance.
S011, Distinct types	Distinct types are strongly typed scalar types. A distinct type can be emulated in Oracle using an object type with only one attribute. The standard's Information Schema view called `USER_DEFINED_TYPES` is equivalent to Oracle's metadata view `ALL_TYPES`.
T321, Basic SQL-invoked routines	Oracle fully supports these subfeatures: • T321-03, function invocation • T321-04, `CALL` statement Oracle supports these subfeatures with syntactic differences: • T321-01, user-defined functions with no overloading • T321-02, user-defined procedures with no overloading The Oracle syntax for `CREATE FUNCTION` and `CREATE PROCEDURE` differs from the standard as follows: • In the standard, the mode of a parameter (`IN`, `OUT`, or `INOUT`) comes before the parameter name, whereas in Oracle it comes after the parameter name. • The standard uses `INOUT`, whereas Oracle uses `IN OUT`. • Oracle requires either `IS` or `AS` after the return type and before the definition of the routine body, while the standard lacks these keywords. • If the routine body is in C (for example), then the standard uses the keywords `LANGUAGE C EXTERNAL NAME` to name the routine, whereas Oracle uses `LANGUAGE C NAME`. • If the routine body is in SQL, then Oracle uses its proprietary procedural extension called PL/SQL. Oracle supports the following subfeature in PL/SQL but not in Oracle SQL: • T321-05, `RETURN` statement Oracle provides equivalent functionality for the following subfeatures: • T321-06, `ROUTINES` view: Use the `ALL PROCEDURES` metadata view. • T321-07, `PARAMETERS` view: Use the `ALL_ARGUMENTS` and `ALL_METHOD_PARAMS` metadata views.
T631, `IN` predicate with one list element	Oracle fully supports this feature.

Oracle Support for Optional Features of SQL/Foundation

Oracle's support for optional features of SQL/Foundation is listed in Table C-2:

Table C-2 Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
B012, Embedded C	Oracle fully supports this feature.
B013, Embedded COBOL	Oracle fully supports this feature.
B021, Direct SQL	Oracle fully supports this feature, as SQL*Plus.
B031, Basic dynamic SQL	Oracle supports dynamic SQL in two styles, documented in the embedded language manuals as "Oracle dynamic SQL" and "ANSI dynamic SQL." ANSI dynamic SQL is an implementation of the standard, with the following restrictions: • Oracle supports a subset of the descriptor items. • For <input using clause>, Oracle only supports <using input descriptor>. • For <output using clause>, Oracle only supports <into descriptor>. • Dynamic parameters are indicated by a colon followed by an identifier rather than a question mark. Oracle dynamic SQL is similar to standard dynamic SQL, with the following modifications: • Parameters are indicated by a colon followed by an identifier, instead of a question mark. • Oracle's `DESCRIBE SELECT LIST FOR` statement replaces the standard's `DESCRIBE OUTPUT`. • Oracle provides `DECLARE STATEMENT` if you want to declare a cursor using a dynamic SQL statement physically prior to the `PREPARE` statement that prepares the dynamic SQL statement.
B032, Extended dynamic SQL	In ANSI dynamic SQL, Oracle only implements the ability to declare global statements and global cursors from this feature; the rest of the feature is not supported. In Oracle dynamic SQL, Oracle's `DESCRIBE BIND VARIABLES` is equivalent to the standard's `DESCRIBE INPUT`; the rest of this feature is not supported.
B122, Routine language C	Oracle supports external routines written in C, though Oracle does not support the standard syntax for creating such routines.
B128, Routine language SQL	Oracle supports routines written in PL/SQL, which is Oracle's equivalent to the standard procedural language SQL/PSM.
F032, `CASCADE` drop behavior	In Oracle, a `DROP` command invalidates all of the dropped object's dependent objects. Invalidated objects are effectively unusable until the dropped object is redefined in such a way to allow successful recompilation of the invalidated object.
F033, `ALTER TABLE` statement: `DROP COLUMN` clause	Oracle provides a `DROP COLUMN` clause, but without the `RESTRICT` or `CASCADE` options found in the standard.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
F034, Extended REVOKE statement	Oracle supports the following parts of this feature: • F034-01, REVOKE statement performed by other than the owner of a schema object • F034-03, REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION Oracle provides equivalent functionality for the following parts of this feature: • CASCADE: In Oracle, a REVOKE invalidates all dependent objects, which become effectively unusable until the metadata is changed through subsequent CREATE and GRANT commands enabling the invalidated object to be successfully recompiled.
F052, Intervals and datetime arithmetic	Oracle only supports the INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND data types.
F111, Isolations levels other than SERIALIZABLE	In addition to SERIALIZABLE, Oracle supports the READ COMMITTED isolation level.
F121, Basic diagnostics management	Much of the functionality of this feature is provided through the SQLCA in embedded languages.
F191, Referential delete actions	Oracle supports ON DELETE CASCADE and ON DELETE SET NULL.
F200, TRUNCATE TABLE	Oracle fully supports this feature, and extends it by permitting truncation of a table that references itself in a referential integrity constraint, and the ability to cascade to child tables with enabled ON DELETE CASCADE referential constraints.
F231, Privilege tables	Oracle makes this information available in the following metadata views: • Instead of TABLE_PRIVILEGES, use ALL_TAB_PRIVS. • Instead of COLUMN_PRIVILEGES, use ALL_COL_PRIVS. • Oracle does not support USAGE privileges so there is no equivalent to USAGE_PRIVILEGES.
F281, LIKE enhancements	Oracle fully supports this feature.

F291, UNIQUE predicate	The IS A SET condition may be used to test whether a multiset is a set; that is, each row is unique. Thus, the equivalent of
	`UNIQUE <table subquery>`
	is
	`CAST (<table subquery> AS MULTISET) IS A SET`

F302, INTERSECT table operator	Oracle supports INTERSECT but not INTERSECT ALL. Syntactically, Oracle differs from the standard in that UNION, INTERSECT, and MINUS have the same precedence.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
F312, MERGE statement	The Oracle MERGE statement is almost the same as the standard, with these exceptions: • Oracle does not support the optional AS keyword before a table alias. • Oracle does not support the ability to rename columns of the table specified in the USING clause with a parenthesized list of column names following the table alias. • Oracle does not support the <override clause>.
F314, MERGE statement with DELETE branch	Oracle has similar functionality, though in Oracle you must first update a row, after which you can delete it if the revised row meets a condition.
F321, User authorization	Oracle provides equivalent functionality for the following subfeatures: • Use SYS_CONTEXT ('USERENV', 'SESSION_USER') instead of SESSION_USER • Use SYS_CONTEXT ('USERENV', 'CURRENT_USER') instead of CURRENT_USER Oracle does not support the following subfeatures: • SYSTEM_USER • SET SESSION AUTHORIZATION statement
F341, Usage tables	Oracle makes this information available in the views ALL_DEPENDENCIES, DBA_DEPENDENCIES, and USER_DEPENDENCIES.
F381, Extended schema manipulation	Oracle fully supports the following element of this feature: • Oracle supports the standard syntax to add a table constraint using ALTER TABLE. Oracle partially supports the following element of this feature: • Oracle supports the standard syntax to drop a table constraint, except that Oracle does not support RESTRICT. Oracle provides equivalent functionality for the following element of this feature: • To alter the default value of a column, use the MODIFY option of ALTER TABLE. Oracle does not support the following parts of this feature: • DROP SCHEMA statement • ALTER ROUTINE statement
F382, Alter column data type	Oracle supports this functionality, though with non-standard syntax. As an extension to the standard, Oracle allows you to reduce the size or precision of a column.
F383, Set column not null clause	Oracle provides equivalent functionality for the two subfeatures of this feature: • To add a NOT NULL constraint to an existing column, use ALTER TABLE ... MODIFY • To drop a NOT NULL constraint, use ALTER TABLE to drop the constraint by name
F384, Drop identity property clause	Oracle provides equivalent functionality using ALTER TABLE ... MODIFY (... DROP IDENTITY)

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
F386, Set identity column generation clause	Oracle provides equivalent functionality. Oracle's syntax and semantics are the same as the standard, with this exception: • Oracle does not support `RESTART`; use `START WITH` instead. When restarting an identity column, the values of the other parameters for the identity column are reset to their defaults unless explicitly set in the `ALTER TABLE` statement. Oracle's `START WITH LIMIT VALUE` option is an extension on the standard.
F391, Long identifiers	Oracle supports identifiers up to 128 characters in length.
F393, Unicode escapes in literals	The Oracle `UNISTR` function supports numeric escape sequences for all Unicode characters.
F394, Optional normal form specification	This feature adds the keywords `NFC`, `NFD`, `NFKC`, and `NKD` to the `NORMALIZE` function and the `IS NORMAL` predicate. Without these keywords, `NFC` is the default (see Feature T061, UCS support). Oracle supports all four normalization forms, with nonstandard syntax, as follows: • For `NFC`, use `COMPOSE` • For `NFD`, use `DECOMPOSE` with the `CANONICAL` option • For `NFKD`, use `DECOMPOSE` with the `COMPATIBILITY` option • For `NFKC`, use `DECOMPOSE` with the `CANONICAL` option followed by `COMPOSE` Oracle does not support the `IS NORMAL` predicate.
F401, Extended joined table	Oracle supports `FULL` outer joins, `CROSS` joins, and `NATURAL` joins.
F402, Named column joins for LOBs, arrays and multisets	Oracle supports named column joins for columns whose declared type is nested table. Oracle does not support named column joins for LOBs or arrays.
F403, Partitioned join tables	Oracle supports this feature, except with `FULL` outer joins.
F411, Time zone specification	Oracle fully supports `TIMESTAMP WITH TIME ZONE`, but does not support `TIME WITH TIME ZONE`.
F421, National character	Oracle fully supports this feature.
F431, Read-only scrollable cursors	Oracle fully supports this feature.
F441, Extended set function support	Oracle supports the following parts of this feature: • The ability in the `WHERE` clause to reference a column that is defined using an aggregate, either in a view or an inline view • `COUNT` without `DISTINCT` of an expression • Aggregates that reference columns that are outer references with respect to the aggregating query. However, Oracle defines the aggregating query as the innermost query containing the aggregate, rather than the innermost query that defines a range variable referenced in the aggregate.
F442, Mixed column references in set functions	Oracle fully supports this feature.
F461, Named character sets	Oracle supports many character sets with Oracle-defined names. Oracle does not support any other aspect of this feature.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
F491, Constraint management	Oracle fully supports this feature.
F492, Optional table constraint enforcement	ENFORCED in the standard is equivalent to ENABLE VALIDATE in Oracle. NOT ENFORCED in the standard is equivalent to DISABLE NOVALIDATE in Oracle. Other combinations of the ENABLE | DISABLE, VALIDATE | NOVALIDATE, and RELY | NORELY options are extensions of the standard.
F531, Temporary tables	Oracle supports GLOBAL TEMPORARY tables.
F555, Enhanced seconds precision	Oracle provides enhanced support for this feature, supporting up to 9 places after the decimal point.
F561, Full value expressions	Oracle fully supports this feature.
F571, Truth value tests	Oracle's LNNVL function is equivalent to the standard's IS NOT TRUE predicate.
F591, Derived tables	Oracle supports <derived table>, with the exception of: • Oracle does not support the optional AS keyword before a table alias. • Oracle does not support <derived column list>.
F641, Row and table constructors	In Oracle, a row constructor may be used in an equality or inequality comparison with another row constructor or with a subquery. Oracle does not support anything else in this feature.
F690, Collation support	Oracle's NLSSORT function may be used to change the collation of character expressions.
F693, SQL-sessions and client module collations	To set a session collation, use ALTER SESSION SET NLS_COMP = 'LINGUISTIC' and also set NLS_SORT to your desired collation. Oracle does not support client module collations.
F695, Translation support	The Oracle CONVERT function can convert between the database character set and the national character set. For other character sets, store the data in the RAW data type and use the PL/SQL package function UTL_RAW.CONVERT. Oracle does not provide the ability to add or drop character set conversions.
F721, Deferrable constraints	Oracle fully supports this feature.
F731, INSERT column privileges	Oracle fully supports this feature.
F761, Session management	Oracle provides the following equivalents for elements of this feature: • The equivalent to the standard's SET SESSION CHARACTERISTICS AS TRANSACTION SERIALIZABLE is ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE. • The equivalent to the standard's SET SCHEMA is ALTER SESSION SET CURRENT_SCHEMA. • The equivalent to the standard's SET COLLATION is ALTER SESSION SET NLS_SORT.
F763, CURRENT_SCHEMA	Oracle's equivalent is SYS_CONTEXT ('USERENV', 'CURRENT_SCHEMA')

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
F771, Connection management	Oracle's `CONNECT` statement provides the same functionality as the standard's `CONNECT` statement, though with different syntax. Instead of using the standard's `SET CONNECTION`, Oracle provides the `AT` clause to indicate which connection a SQL statement should be performed on. Oracle embedded languages let you disconnect from a connection by using the `RELEASE` option of either `COMMIT` or `ROLLBACK`.
F781, Self-referencing operations	Oracle fully supports this feature.
F801, Full set function	Oracle fully supports this feature.
F831, Full cursor update	Oracle supports the combination of `FOR UPDATE` and `ORDER BY` clauses in a query.
F841, `LIKE_REGEX` predicate	Oracle's equivalent is `REGEXP_LIKE`. Oracle's pattern syntax lacks some of the features of the standard's. Oracle's match parameter has the same capabilities as the standard's, though with a few spelling differences.
F842, `OCCURRENCES_REGEX` function	Oracle's equivalent is `REGEXP_COUNT`. Oracle's pattern syntax lacks some of the features of the standard's. Oracle's match parameter has the same capabilities as the standard's, though with a few spelling differences.
F843, `POSITION_REGEX` function	Oracle's equivalent is `REGEXP_INSTR`. Oracle's pattern syntax lacks some of the features of the standard's. Oracle's match parameter has the same capabilities as the standard's, though with a few spelling differences.
F844, `SUBSTRING_REGEX` function	Oracle's equivalent is `REGEXP_SUBSTR`. Oracle's pattern syntax lacks some of the features of the standard's. Oracle's match parameter has the same capabilities as the standard's, though with a few spelling differences.
F845, `TRANSLATE_REGEX` function	Oracle's equivalent is `REGEXP_REPLACE`. Oracle's pattern syntax lacks some of the features of the standard's. Oracle's match parameter has the same capabilities as the standard's, though with a few spelling differences.
F850, Top-level <order by clause> in <query expression>	Oracle fully supports this feature.
F851, <order by clause> in subqueries	Oracle fully supports this feature.
F852, Top-level <order by clause> in views	Oracle fully supports this feature.
F855, Nested <order by clause> in <query expression>	Oracle fully supports this feature.
F856, Nested <fetch first clause> in <query expression>	Oracle fully supports this feature.
F857, Top-level <fetch first clause> in a <query expression>	Oracle fully supports this feature.
F858, <fetch first clause> in subqueries	Oracle fully supports this feature.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
F859, Top-level <fetch first clause> in views	Oracle fully supports this feature.
F860, Dynamic <fetch first row count> in <fetch first clause>	Oracle fully supports this feature.
F861, Top-level <result offset clause> in <query expression>	Oracle fully supports this feature.
F862, <result offset clause> in subqueries	Oracle fully supports this feature.
F863, Nested <result offset clause> in <query expression>	Oracle fully supports this feature.
F864, Top-level <result offset clause> in views	Oracle fully supports this feature.
F865, Dynamic <offset row count> in <result offset clause>	Oracle fully supports this feature.
F866, `FETCH FIRST` clause: `PERCENT` option	Oracle fully supports this feature.
F867, `FETCH FIRST` clause: `WITH TIES` option	Oracle fully supports this feature.
R010, Row pattern recognition: `FROM` clause	Oracle fully supports this feature.
S023, Basic structured types	Oracle's object types are equivalent to structured types in the standard.
S024, Enhanced structured types	Oracle's syntax is non-standard, but provides equivalents for the following: • `NOT INSTANTIABLE` • `STATIC` methods • `RELATIVE`, `MAP`, and `STATE` orderings. The keyword in Oracle for `RELATIVE` orderings is `ORDER`. There is no keyword for `STATE` orderings (this is the default, if no other ordering is defined). Unlike the standard, Oracle does not support `EQUALS ONLY` on non-`STATE` orderings. (See also Feature S251, User-defined orderings.) • `SELF AS RESULT` in the signature of constructor methods
S025, Final structured types	Oracle's final object types are equivalent to final structured types in the standard.
S026, Self-referencing structured types	In Oracle, an object type OT may have a reference that references OT.
S041, Basic reference types	Oracle's reference types are equivalent to reference types in the standard. To dereference a reference, dot notation is used, instead of -> as in the standard.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
S043, Enhanced reference types	Oracle supports the following elements of this feature: • `DEREF` operator to return the object referenced by a reference • `SCOPE` clause as a constraint on columns of tables or materialized views • Adding and dropping the scope of a column • References that are either system-generated or derived from the primary key (but not from any other list of columns, nor from a list of attributes of the type)
S051, Create table of type	Oracle's object tables are equivalent to tables of structured type in the standard.
S081, Subtables	Oracle supports hierarchies of object views, but not of object base tables. To emulate a hierarchy of base tables, create a hierarchy of views on those base tables.
S091, Basic array support	Oracle `VARRAY` types are equivalent to array types in the standard. However, Oracle does not support storage of arrays of LOBs. To access a single element of an array using a subscript, you must use PL/SQL. Oracle supports the following aspects of this feature with nonstandard syntax: • To construct an instance of varray type, including an empty array, use the varray type constructor. • To unnest a varray in the `FROM` clause, use the `TABLE` operator. • To get the cardinality of a varray, use the `COUNT` method in PL/SQL.
S092, Arrays of user-defined types	Oracle supports `VARRAY`s of object types.
S094, Arrays of reference types	Oracle supports `VARRAY`s of references.
S095, Array constructors by query	Oracle supports this using `CAST` (`MULTISET` (`SELECT` ...) `AS` *`varray_type`*`)`. The ability to order the elements of the array using `ORDER BY` is not supported.
S097, Array element assignment	In PL/SQL, you can assign to array elements, using syntax that is similar to the standard (SQL/PSM).
S098, `ARRAY_AGG`	Oracle does not have an aggregate that results in a varray. Instead, the `COLLECT` aggregate may be used to create a multiset, which can be cast to an array of the element type.
S111, `ONLY` in query expressions	Oracle supports the `ONLY` clause for view hierarchies; Oracle does not support hierarchies of base tables.
S151, Type predicate	Oracle fully supports this feature.
S161, Subtype treatment	Oracle fully supports this feature.
S162, Subtype treatment for references	Supported, with a minor syntactic difference: The standard requires parentheses around the referenced type's name; Oracle does not support parentheses in this position.
S201, SQL-invoked routines on arrays	PL/SQL provides the ability to pass arrays as parameters and return arrays as the result of functions. Procedures and functions written in C may pass arrays and return arrays as the result of functions using the Oracle Type Translator (OTT).

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
S202, SQL-invoked routines on multisets	A PL/SQL routine may have nested tables as parameters, and may return a nested table. Routines written in C may pass arrays and return arrays as the result of functions using the Oracle Type Translator.
S232, Array locators	Oracle Type Translator supports descriptors for arrays, which achieve the same purpose as locators.
S233, Multiset locators	Oracle supports locators for nested tables.
S241, Transform functions	The Oracle Type Translator provides the same capability as transforms.
S251, User-defined orderings	Oracle's object type ordering capabilities correspond to the standard's capabilities as follows: • Oracle's `MAP` ordering corresponds to the standard's `ORDER FULL BY MAP` ordering. • Oracle's `ORDER` ordering corresponds to the standard's `ORDER FULL BY RELATIVE` ordering. • If an Oracle object type has neither `MAP` nor `ORDER` declared, then this corresponds to `EQUALS ONLY BY STATE` in the standard. • Oracle does not have unordered object types; you can alter the ordering but you cannot drop it.
S261, Specified type method	The `GetTypeName` method of the `ANYDATA` type may be used to learn the name of a type.
S271, Basic multiset support	Multisets in the standard are supported as nested table types in Oracle. The Oracle nested table data type based on a scalar type ST is equivalent, in standard terminology, to a multiset of rows having a single field of type ST and named *`column_value`*. The Oracle nested table type based on an object type is equivalent to a multiset of structured type in the standard. Oracle supports the following elements of this feature on nested tables using the same syntax as the standard has for multisets: • The `CARDINALITY` function • The `SET` function • The `MEMBER` predicate • The `IS A SET` predicate • The `COLLECT` aggregate All other aspects of this feature are supported with non-standard syntax, as follows: • To create an empty multiset, denoted `MULTISET[]` in the standard, use an empty constructor of the nested table type. • To obtain the sole element of a multiset with one element, denoted `ELEMENT` (<multiset value expression>) in the standard, use a scalar subquery to select the single element from the nested table. • To construct a multiset by enumeration, use the constructor of the nested table type. • To construct a multiset by query, use `CAST` with a multiset argument, casting to the nested table type. • To unnest a multiset, use the `TABLE` operator in the `FROM` clause.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
S272, Multisets of user-defined types	Oracle's nested table type permits a multiset of structured types. Oracle does not have distinct types, so a multiset of distinct types is not supported.
S274, Multisets of reference types	A nested table type can have one or more columns of reference type.
S275, Advanced multiset support	Oracle supports the following elements of this feature on nested tables using the same syntax as the standard has for multisets: • The `MULTISET UNION`, `MULTISET INTERSECTION`, and `MULTISET EXCEPT` operators • The `SUBMULTISET` predicate • = and <> predicates Oracle does not support the `FUSION` or `INTERSECTION` aggregates.
S281, Nested collection types	Oracle permits nesting of its collection types (varray and nested table).
S401, Distinct types based on array types	Oracle's varray types are strongly typed.
S403, `ARRAY_MAX_CARDINALITY`	In PL/SQL, the `LIMIT` method of a varray returns its maximum cardinality.
S404, `TRIM_ARRAY`	In PL/SQL, the `TRIM` method of a varray can be used to trim the varray.
T041, Basic LOB data type support	Oracle supports the following aspects of this feature: • The keywords `BLOB`, `CLOB`, and `NCLOB` • Concatenation, `UPPER`, `LOWER` on `CLOB`s Oracle provides equivalent support for the following aspects of this feature: • Use `INSTR` instead of `POSITION`. • Use `LENGTH` instead of `CHAR_LENGTH`. Oracle does not support the following aspects of this feature: • The keywords `BINARY LARGE OBJECT`, `CHARACTER LARGE OBJECT`, and `NATIONAL CHARACTER LARGE OBJECT` as synonyms for `BLOB`, `CLOB`, and `NCLOB`, respectively • <binary string literal> • The ability to specify an upper bound on the length of a `BLOB` or `CLOB` • Concatenation of `BLOB`s

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T042, Extended LOB support	Oracle fully supports the following element of this feature: • TRIM function on a CLOB argument Oracle provides equivalent functionality for the following elements of this feature: • BLOB and CLOB substring, supported using SUBSTR • SIMILAR predicate, supported using REGEXPR_LIKE to perform pattern matching with a Perl-like syntax The following elements of this feature are not supported: • Comparison predicates with BLOB or CLOB operands • CAST with a BLOB or CLOB operand • OVERLAY (This may be emulated using SUBSTR and string concatenation.) • LIKE predicate with BLOB or CLOB operands
T051, Row types	Oracle object types can be used in place of the standard's row types.
T061, UCS support	Oracle provides equivalent functionality for the following elements of this feature: • Oracle supports the keyword CHAR instead of CHARACTERS, and BYTE instead of OCTETS, in a character data type declaration. • The Oracle COMPOSE function is equivalent to the standard's NORMALIZE function. Oracle does not support the IS NORMALIZED predicate.
T071, BIGINT data type	On many implementations, BIGINT refers to a binary integer type with 64 bits, which supports almost 19 decimal digits. The Oracle NUMBER type supports 39 decimal digits.
T111, Updatable joins, unions and columns	Oracle's updatable join views are similar to the standard's updatable join capabilities. Unlike the standard, Oracle does not require an updatable join view to display the strong candidate key in the SELECT list. Although an updatable join view might have more than one key-preserved table, only one of them may be modified using an UPDATE or DELETE, unlike the standard, which modifies all key-preserved tables of an updatable join.
T121, WITH (excluding RECURSIVE) in query expression	Oracle fully supports this feature.
T122, WITH (excluding RECURSIVE) in subquery	Oracle fully supports this feature.
T131, Recursive query	Oracle supports the use of a WITH clause element that references itself, but without the RECURSIVE keyword. Alternatively, Oracle's START WITH and CONNECT BY clauses can be used to perform many recursive queries.
T132, Recursive query in subquery	Oracle supports the use of a WITH clause element that references itself, but without the RECURSIVE keyword. Alternatively, Oracle's START WITH and CONNECT BY clauses can be used to perform many recursive queries.
T141, SIMILAR predicate	Oracle provides REGEXP_LIKE for pattern matching with a Perl-like syntax.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T172, AS subquery clause in table definition	Oracle's AS subquery feature of CREATE TABLE has substantially the same functionality as the standard, though there are some syntactic differences.
T174, Identity columns	Oracle supports this feature, with the following syntactic differences: • Oracle uses NOMINVALUE and NOMAXVALUE instead of the standard's NO MINVALUE and NO MAXVALUE. • To restart an identity column, in an ALTER TABLE MODIFY statement, use START WITH LIMIT VALUE to restart at the highest value (for an increasing identity column) or the lowest value (for a decreasing identity column); use START WITH number to restart at a specific number. GENERATED BY DEFAULT ON NULL is an Oracle extension.
T175, Generated columns	Oracle supports this feature, with the following restrictions: • Generated columns are not supported in temporary tables. • The data type of a generated column may not be LOB or XML.
T176, Sequence generator support	Oracle's sequences have the same capabilities as the standard's, though with different syntax.
T178, Identity columns: simple restart option	Oracle's START WITH LIMIT VALUE is the same as the standard's simple restart if the identity column has not cycled.
T180, System-versioned tables	Oracle's Flashback capability is substantially the same as the standard's system-versioned tables. Some key differences are: • In Oracle you do not need to designate particular tables for journaling; all tables are journaled. • In Oracle, LOB columns need to be individually designated for journaling, because of the potential for large amounts of data. The standard has no analogous provision. • In Oracle you need a privilege in order to read historical data. • In the standard, journaled tables have columns to record the start and end timestamps for the row. In Oracle, this is provided through pseudocolumns.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T181, Application-time period tables	Oracle supports the following elements of this feature: • Application-time period definition during `CREATE TABLE` • Adding and dropping an application-time period definition using `ALTER TABLE` with a minor syntactic difference: Oracle requires parentheses around the period specification; the standard does not support parentheses in this position. Oracle extends this feature: • With the ability to have more than one application-time period per table. • By making the start time and end time columns optional. In this case, Oracle will create these columns implicitly. • By allowing `NULL` for the start time column to indicate that the row is considered valid for any point in time before the value of the end time column. • By allowing `NULL` for the end time column to indicate that the row is considered valid for any point in time on or after the value of the start time column. • By querying an application-time period table using the flashback query options `VERSIONS PERIOD FOR` and `AS OF PERIOD FOR`.
T201, Comparable data types for referential constraints	Oracle fully supports this feature.
T211, Basic trigger capability	Oracle's triggers differ from the standard as follows: • Oracle does not provide the optional syntax `FOR EACH STATEMENT` for the default case, the statement trigger. • Oracle does not support `OLD TABLE` and `NEW TABLE`; the transition tables specified in the standard (the multiset of before and after images of affected rows) are not available. • The trigger body is written in PL/SQL, which is functionally equivalent to the standard's procedural language PSM, but not the same. • In the trigger body, the new and old transition variables are referenced beginning with a colon. • Oracle's row triggers are executed as the row is processed, instead of buffering them and executing all of them after processing all rows. The standard's semantics are deterministic, but Oracle's in-flight row triggers are more performant. • Oracle's before-row and before-statement triggers can perform DML statements, which is forbidden in the standard. However, Oracle's after-row statements cannot perform DML, while it is permitted in the standard. • When multiple triggers apply, the standard says they are executed in order of definition. In Oracle the execution order is nondeterministic, unless specified using `FOLLOWS`. • Oracle uses the system privileges `CREATE TRIGGER` and `CREATE ANY TRIGGER` to regulate creation of triggers, instead of the standard's `TRIGGER` privilege, which is a table privilege.
T212, Enhanced trigger capability	This feature permits statements triggers, which Oracle supports, as described for feature T211, Basic trigger capability.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T213, `INSTEAD OF` triggers	Oracle supports `INSTEAD OF` triggers on views, with syntax and semantics agreeing with the standard except as noted for feature T211, Basic trigger capability. Oracle permits an `INSTEAD OF` trigger on a view that specified `WITH CHECK OPTION`, unlike the standard.
T241, `START TRANSACTION` statement	Oracle's `SET TRANSACTION` statement starts a transaction making it equivalent to the standard's `START TRANSACTION` rather than the standard's `SET TRANSACTION`. Oracle's `READ ONLY` transactions are at `SERIALIZABLE` isolation level.
T271, Savepoints	Oracle supports this feature, except: • Oracle does not support `RELEASE SAVEPOINT`. • Oracle does not support savepoint levels.
T285, Enhanced derived column names	This feature pertains only to derived columns in a `SELECT` list with no column alias and consisting of a SQL parameter reference. In that case, the column name defaults to the parameter name, the same as in the standard.
T323, Explicit security for external routines	The Oracle syntax `AUTHID` { `CURRENT USER` | `DEFINER` } when used when creating an external function, procedure, or package is equivalent to the standard's `EXTERNAL SECURITY` { `DEFINER` | `INVOKER` }.
T324, Explicit security for SQL routines	Oracle's syntax `AUTHID` { `CURRENT USER` | `DEFINER` } when used when creating a PL/SQL function, procedure, or package is equivalent to the standard's `SQL SECURITY` { `DEFINER` | `INVOKER` }.
T325, Qualified SQL parameter reference	PL/SQL supports the use of a routine name to qualify a parameter name.
T326, Table functions	Oracle provides equivalents for the following elements of this feature: • <multiset value constructor by query> is supported using `CAST` (`MULTISET` (<*query expression*>) `AS` <*nested table type*>) • <table function derived table> is supported using the `TABLE` operator in the `FROM` clause with a varray or nested table as the argument • <collection value expression> is equivalent to an Oracle expression resulting in a varray or nested table • <returns table type> is equivalent to a PL/SQL function that returns a nested table
T331, Basic roles	Oracle supports this feature, except for `REVOKE ADMIN OPTION FOR` <role name>.
T341, Overloading of SQL-invoked functions and procedures	Oracle supports overloading of functions and procedures. However, the rules for handling certain data type combinations are not the same as the standard. For example, the standard permits the coexistence of two functions of the same name differing only in the numeric types of the arguments, whereas Oracle does not permit this.
T351, Bracketed comments	Oracle fully supports this feature.
T431, Extended grouping capabilities	Oracle fully supports this feature.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T432, Nested and concatenated GROUPING SETS	Oracle supports concatenated GROUPING SETS, but not nested GROUPING SETS.
T433, Multiargument function GROUPING	The Oracle GROUP_ID function can be used to conveniently distinguish groups in a grouped query, serving the same purpose as the standard multiargument GROUPING function.
T441, ABS and MOD functions	Oracle supports the ABS function. Oracle's MOD function is similar to the standard, though the behavior is different if the two arguments are of opposite sign.
T471, Result sets return value	PL/SQL ref cursors provide all the functionality of the standard's result set cursors.
T491, LATERAL derived tables	Oracle fully supports this feature.
T501, Enhanced EXISTS predicate	Oracle fully supports this feature.
T511, Transaction counts	Oracle supports the count of transactions committed and rolled back via the system views V$STATNAME and V$SESSTAT.
T521, Named arguments in CALL statement	Oracle fully supports this feature.
T522, Default values for IN parameters of SQL-invoked procedures	Oracle fully supports this feature.
T524, Named arguments in routine invocations other than a CALL statement	Oracle fully supports this feature.
T525, Default values for parameters of SQL-invoked functions	Oracle fully supports this feature.
T571, Array-returning external SQL-invoked function	Oracle table functions returning a varray can be defined in external programming languages. When declaring such functions in SQL, use the CREATE FUNCTION command with the PIPELINED USING clause.
T572, Multiset-returning external SQL-invoked function	Oracle table functions returning a nested table can be defined in external programming languages. When declaring such functions in SQL, use the CREATE FUNCTION command with the PIPELINED USING clause. In the body of the function, use the OCITable interface. The function must be invoked within the TABLE operator in the FROM clause.
T581, Regular expressions substring functions	Oracle provides the REGEXP_SUBSTR function to perform substring operations using regular expression matching.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T591, UNIQUE constraints of possibly null columns	Oracle permits a UNIQUE constraint on one or more nullable columns. If the UNIQUE constraint is on a single column, then the semantics are the same as the standard (the constraint permits any number of rows that are null in the designated column). If the UNIQUE constraint is on two or more columns, then the semantics are nonstandard. Oracle permits any number of rows that are null in all the designated columns. Unlike the standard, if a row is non-null in at least one of the designated columns, then another row having the same values in the non-null columns of the constraint is a constraint violation and not permitted.
T611, Elementary OLAP operations	Oracle fully supports this feature, except that DISTINCT is only supported in conjunction with window partitioning but not with window framing.
T612, Advanced OLAP operations	Oracle supports the following elements of this feature: PERCENT_RANK, CUME_DIST, WIDTH_BUCKET, hypothetical set functions, PERCENTILE_CONT, PERCENTILE_DISC, and ROW_NUMBER. Oracle does not support the following elements of this feature: • window names • EXCLUDE • ROW_NUMBER without ORDER BY
T613, Sampling	Oracle uses the keyword SAMPLE instead of the standard's keyword, TABLESAMPLE. Oracle uses the keyword BLOCK instead of the standard's keyword, SYSTEM. Oracle uses the absence of the keyword BLOCK to indicate a Bernoulli sampling of rows, indicated in the standard by the keyword BERNOULLI. Oracle does not support sampling of derived tables or views that are not key-preserving. Oracle does not permit sampling in a subquery of a DELETE, UPDATE or MERGE statement.
T614, NTILE function	Oracle fully supports this feature.
T615, LEAD and LAG functions	Oracle fully supports this feature.
T616, Null treatment option for LEAD and LAG functions	Oracle fully supports this feature.
T617, FIRST_VALUE and LAST_VALUE functions	Oracle fully supports this feature.
T618, NTH_VALUE function	Oracle fully supports this feature.
T621, Enhanced numeric functions	Oracle fully supports this feature, except for the alternate spelling CEILING of the CEIL function.
T622, Trigonometric functions	Oracle fully supports this feature.
T623, General logarithm function	Oracle fully supports this feature.
T625, LISTAGG	Oracle fully supports this feature, except that the keyword DISTINCT is not supported
T641, Multiple column assignment	The standard syntax to assign to multiple columns is supported if the assignment source is a subquery.
T652, SQL-dynamic statements in SQL routines.	PL/SQL supports dynamic SQL.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T654, SQL-dynamic statements in external routines	Oracle supports dynamic SQL in embedded C, which may be used to create an external routine.
T655, Cyclically dependent routines	PL/SQL supports recursion.
T811, Basic SQL/JSON constructor functions	Oracle fully supports this feature, except for the `JSON_ARRAY` constructor by query.
T812, SQL/JSON: `JSON_OBJECTAGG`	Oracle fully supports this feature.
T813, SQL/JSON: `JSON_ARRAYAGG` with `ORDER BY`	Oracle fully supports this feature.
T821, Basic SQL/JSON query operators	Oracle fully supports this feature.
T822, SQL/JSON: `IS JSON WITH UNIQUE KEYS` predicate	Oracle fully supports this feature.
T823, SQL/JSON: `PASSING` clause	Oracle supports the `PASSING` clause in `JSON_EXISTS`.
T825, SQL/JSON: `ON EMPTY` and `ON ERROR` clauses	Oracle fully supports this feature, except that: • The `ON ERROR` clause for `JSON_EXISTS` does not support `UNKNOWN`. • `JSON_TABLE` does not support a column-level `ON EMPTY` clause.
T828, `JSON_QUERY`	Oracle fully supports this feature.
T829, `JSON_QUERY`: array wrapper options	Oracle fully supports this feature.
T832, SQL/JSON path language: item method	Oracle fully supports the following item methods: • `abs` • `ceiling` • `double` • `floor` Oracle provides the following comparable support: • `date` and `timestamp` are comparable to the standard's `datetime` Oracle extends this feature by supporting the following item methods: • `length` • `lower` • `number` • `string` • `upper`
T833, SQL/JSON path language: multiple subscripts	Oracle fully supports this feature, except that subscripts have to be specified in strictly monotonically increasing order.
T834, SQL/JSON path language: wildcard member accessor	Oracle fully supports this feature.

Table C-2 (Cont.) Oracle Support for Optional Features of SQL/Foundation

Feature ID, Feature	Support
T835, SQL/JSON path language: filter expression	Oracle supports the filter expression as the last step of the SQL/JSON path expression in `JSON_EXISTS`.
T839, Formatted cast of datetimes to/from character strings	Oracle supports this feature with a minor syntactic difference: Oracle uses a comma instead of the keyword `FORMAT`.

Oracle Compliance with SQL/CLI

The Oracle ODBC driver conforms to SQL/CLI.

Oracle Compliance with SQL/PSM

Oracle PL/SQL provides functionality equivalent to SQL/PSM, with minor syntactic differences, such as the spelling or arrangement of keywords.

Oracle Compliance with SQL/MED

Oracle does not comply with SQL/MED.

Oracle Compliance with SQL/OLB

Oracle SQLJ conforms to SQL/OLB:1999 and not yet to SQL/OLB:2016.

Oracle Compliance with SQL/JRT

Oracle fully supports stored routines and SQL types implemented in Java(TM). Oracle provides equivalent support for the creation and maintenance of such types and procedures. Oracle's capabilities are in general a superset of the functionality defined by the standard.

Oracle Compliance with SQL/XML

The XML data type in the standard is `XML`. The Oracle equivalent data type is `XMLType`. A feature of the standard is considered to be fully supported if the only difference between Oracle and the standard is the spelling of the data type name.

Table C-3 describes Oracle's support for the features of SQL/XML.

Table C-3 Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X010, XML type	Oracle fully supports this feature.
X011, Arrays of XML types	Oracle supports this feature using named array types

Table C-3 (Cont.) Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X012, Multisets of XML type	The Oracle equivalent of a multiset of XML type is a nested table with a single column of XML type.
X013, Distinct types of XML	A distinct type can be emulated using an object type with a single attribute.
X014, Attributes of XML type	In Oracle, attributes of object types may be of type `XMLType`, but the syntax for creating object types is nonstandard.
X015, Fields of XML type	Oracle object types may be used instead of row types; Oracle supports object types with attributes of `XMLType`.
X016, Persistent XML values	Oracle fully supports this feature.
X020, `XMLConcat`	Oracle fully supports this feature.
X025, `XMLCast`	Oracle supports this feature, with the following restrictions: • The source expression must be of `XMLType` and the target data type may not be `XMLType`. (Since Oracle has only one XML type, there is no need to cast from XML to XML.) • Oracle does not support <XML passing mechanism>; the behavior is the same as `BY VALUE` in the standard. Oracle extends this feature with the ability to cast to type `REF XMLTYPE`.
X031, `XMLElement`	Oracle fully supports this feature.
X032, `XMLForest`	Oracle fully supports this feature.
X034, `XMLAgg`	Oracle fully supports this feature.
X035, `XMLAgg`: `ORDER BY` option	Oracle fully supports this feature.
X036, `XMLComment`	Oracle fully supports this feature.
X036, `XMLPi`	Oracle fully supports this feature.
X038, `XMLText`	The Oracle `XMLCData` function may be used to create a text node.
X040, Basic table mapping	Oracle table mappings are available through a Java interface and through a package. Oracle table mappings have been generalized to map queries and not just tables. To map only a table: `SELECT * FROM` *`table_name`*. This provides support for the following elements of this feature: • X041, Basic table mapping: null absent • X042, Basic table mapping: null as nil • X043, Basic table mapping: table as forest • X044, Basic table mapping: table as element • X045, Basic table mapping: with target namespace • X046, Basic table mapping: data mapping • X047, Basic table mapping: metadata mapping • X049, Basic table mapping: hex encoding Oracle does not support the following element of this feature: • X048, Basic table mapping: base64 encoding
X041, Basic table mapping: null absent	See X040.

Table C-3 (Cont.) Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X042, Basic table mapping: null as nil	See X040.
X043, Basic table mapping: table as forest	See X040.
X044, Basic table mapping: table as element	See X040.
X045, Basic table mapping: with target namespace	See X040.
X046, Basic table mapping: data mapping	See X040.
X047, Basic table mapping: metadata mapping	See X040.
X049, Basic table mapping: hex encoding	See X040.
X060, `XMLParse`: Character string input and `CONTENT` option	Oracle does not support the {`PRESERVE` | `STRIP`} `WHITESPACE` syntax. The behavior is always `STRIP WHITESPACE`.
X061, `XMLParse`: Character string input and `DOCUMENT` option	Oracle does not support the {`PRESERVE` | `STRIP`} `WHITESPACE` syntax. The behavior is always `STRIP WHITESPACE`.
X069, `XMLSERIALIZE`: `INDENT`	Oracle extends this feature with the ability to specify an indent size.
X070, `XMLSerialize`: Character string serialization and `CONTENT` option	Oracle supports this feature, with this restriction: • In the standard, the choice of `DOCUMENT` or `CONTENT` is optional; in Oracle, you must specify one of these. Oracle extends this feature as follows: the standard requires a target data type; Oracle defaults to `CLOB`.
X071, `XMLSerialize`: Character string serialization and `DOCUMENT` option	Oracle fully supports this feature.
X072, `XMLSerialize`: Character string serialization	Oracle fully supports this feature.
X073, `XMLSerialize`: `BLOB` serialization and `CONTENT` option	Oracle fully supports this feature.
X074, `XMLSerialize`: `BLOB` serialization and `DOCUMENT` option	Oracle fully supports this feature.
X075, `XMLSerialize`: `BLOB` serialization	Oracle fully supports this feature.
X076, `XMLSerialize`: `VERSION` option	Use `XMLRoot` to set the XML version prior to serialization.
X077, `XMLSerialize`: explicit `ENCODING` option	Oracle fully supports this feature.

Table C-3 (Cont.) Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X080, Namespaces in XML publishing	In the Oracle implementation of XMLElement, XMLAttributes are used to define namespaces (XMLNamespaces is not implemented). However, XMLAttributes is not supported for XMLForest.
X086, XML namespace declarations in XMLTable	Oracle fully supports this feature.
X090, XML document predicate	In Oracle, you can test whether an XML value is a document by using the ISFRAGMENT method.
X096, XMLExists	Oracle fully supports this feature, with this exception: Oracle only supports passing by value, so the keywords BY VALUE are optional at the beginning of the PASSING clause, and not supported on individual arguments.
X120, XML parameters in SQL routines	Oracle fully supports this feature.
X121, XML parameters in external routines	Oracle supports XML values passed to external routines using a non-standard interface.
X141, IS VALID predicate: data drive case	The XMLISVALID method is equivalent to the IS VALID predicate, and supports the data-driven case.
X142, IS VALID predicate: ACCORDING TO clause	The XMLISVALID method is equivalent to the IS VALID predicate, and includes the equivalent of the ACCORDING TO clause.
X143, IS VALID predicate: ELEMENT clause	The XMLISVALID method is equivalent to the IS VALID predicate, and includes the equivalent of the ELEMENT clause.
X144, IS VALID predicate: schema location	The XMLISVALID method is equivalent to the IS VALID predicate, and supports the specification of a schema location for a registered XML Schema.
X145, IS VALID predicate outside check constraints	The XMLISVALID method is equivalent to the IS VALID predicate, and may be used outside check constraints.
X151, IS VALID predicate with DOCUMENT option	The XMLISVALID method is equivalent to the IS VALID predicate, and performs validation equivalent to the DOCUMENT clause. (XMLISVALID does not support "content" validation.)
X156, IS VALID predicate: optional NAMESPACE with ELEMENT clause	The XMLISVALID method is equivalent to the IS VALID predicate, and may be used to validate against an element in any namespace.
X157, IS VALID predicate: NO NAMESPACE with ELEMENT clause	The XMLISVALID method is equivalent to the IS VALID predicate, and may be used to validate against an element in the "no name" namespace.
X160, Basic Information Schema for registered XML Schemas	The Oracle static data dictionary view ALL_XML_SCHEMAS provides a list of the registered XML schemas that are accessible to the current user. The ALL_XML_SCHEMAS.SCHEMA_URL column corresponds to the standard XML_SCHEMAS.XML_SCHEMA_LOCATION column. The target namespace of the registered XML Schemas can be learned by examining ALL_XML_SCHEMAS.SCHEMA. Oracle has no equivalents for the other columns of the standard's XML_SCHEMAS.

Table C-3 (Cont.) Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X161, Advanced Information Schema for registered XML Schemas	Oracle does not have static data dictionary views corresponding to XML_SCHEMA_NAMESPACES and XML_SCHEMA_ELEMENTS in the standard. However, all the information about registered XML Schemas may be learned by examining the actual XML Schema, which is found in the ALL_XML_SCHEMAS.SCHEMA column. This may also be examined to learn whether a registered XML Schema is nondeterministic, and which of its namespaces and elements are nondeterministic.
X191, XML(DOCUMENT (XMLSCHEMA)) type	Oracle does not support this syntax. However, a column of a table can be constrained by a registered XML Schema, in which case all values of the column will be of XML(DOCUMENT(XMLSCHEMA)) type.
X200, XMLQuery	Oracle fully supports the following elements of this feature: • X201, XMLQuery: RETURNING CONTENT • X203, XMLQuery: passing a context item • X204, XMLQuery: initializing an XQuery variable • X206, XMLQuery: NULL ON EMPTY option Oracle only supports passing by value, so the keywords BY VALUE are optional at the beginning of the PASSING clause, and not supported on individual arguments.
X201, XMLQuery: RETURNING CONTENT	See X200.
X203, XMLQuery: passing a context item	See X200.
X204, XMLQuery: initializing an XQuery variable	See X200.
X206, XMLQuery: NULL ON EMPTY option	See X200.
X221, XML passing mechanism BY VALUE	Oracle supports the BY VALUE clause in XMLQuery, XMLTable and XMLExists. In these, BY VALUE is supported as optional syntax at the beginning of an argument list, but not as a modifier on an individual argument or column.
X232, XML(CONTENT(ANY)) type	Oracle does not support this syntax as a type modifier, but the Oracle XMLType supports this data type for transient values. Persistent values are of type XML(DOCUMENT(ANY)), which is a subset of XML(CONTENT(ANY)).
X241, RETURNING CONTENT in XML publishing	Oracle does not support this syntax. In Oracle, the behavior of the publishing functions (XMLAgg, XMLComment, XMLConcat, XMLElement, XMLForest, and XMLPi) is always RETURNING CONTENT.
X251, Persistent XML values of XML(DOCUMENT(UNTYPED)) type	Oracle fully supports this feature.
X252, Persistent values of type XML(DOCUMENT(ANY))	Oracle fully supports this feature.

Table C-3 (Cont.) Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X256, Persistent values of XML(DOCUMENT(XMLSCHEMA)) type	Oracle fully supports this feature.
X260, XML type, ELEMENT clause	Oracle does not support this syntax. However, a column of a table may be constrained by a top-level element in a registered XML Schema.
X263, XML type: NO NAMESPACE with ELEMENT clause	Oracle does not support this syntax. However, a column of a table may be constrained by a top-level element in the "no name" namespace of a registered XML Schema.
X264, XML type: schema location	Oracle does not support this syntax. However, a column of a table may be constrained by a registered XML Schema that is identified by a schema location.
X271, XMLValidate: data driven case	The SCHEMAVALIDATE method is equivalent to XMLValidate, and supports the data-driven case.
X272, XMLValidate: ACCORDING TO clause	The SCHEMAVALIDATE method is equivalent to XMLValidate, and may be used to specify a particular registered XML Schema.
X273, XMLValidate: ELEMENT clause	The SCHEMAVALIDATE method is equivalent to XMLValidate, and may be used to specify a particular element of a particular registered XML Schema.
X274, XMLValidate: schema location	The SCHEMAVALIDATE method is equivalent to XMLValidate, and may be used to specify a particular registered XML Schema by its schema location URL.
X281, XMLValidate with DOCUMENT option	The SCHEMAVALIDATE method is equivalent to XMLValidate. SCHEMAVALIDATE performs validation only of XML documents (not content).
X286, XMLValidate: NO NAMESPACE with ELEMENT clause	The SCHEMAVALIDATE method is equivalent to XMLValidate, and may be used to specify a particular element in the "no name" namespace of a particular registered XML Schema.
X300, XMLTable	Oracle does not support reverse axes in the column path expressions. Aside from that restriction, Oracle fully supports the following elements of this feature: • X086, XML namespace declarations in XMLTable • X302, XMLTable with ordinality column • X303, XMLTable: column default option • X304, XMLTable: passing a context item • X305, XMLTable: initializing an XQuery variable Oracle only supports passing by value, so the keywords BY VALUE are optional at the beginning of the PASSING clause, and not supported on individual arguments.
X302, XMLTable with ordinality column	See X300.
X303, XMLTable: column default option	See X300.
X304, XMLTable: passing a context item	See X300.

Table C-3 (Cont.) Oracle Support for Features of SQL/XML

Feature ID, Feature	Support
X305, `XMLTable`: initializing an XQuery variable	See X300.

Oracle Compliance with FIPS 127-2

Oracle complied fully with last Federal Information Processing Standard (FIPS), which was FIPS PUB 127-2. That standard is no longer published. However, for users whose applications depend on information about the sizes of some database constructs that were defined in FIPS 127-2, the details of our compliance are listed in Table C-4.

Table C-4 Sizing for Database Constructs

Database Constructs	FIPS	Oracle Database
Length of an identifier (in bytes)	18	128
Length of `CHARACTER` data type (in bytes)	240	2,000
Decimal precision of `NUMERIC` data type	15	38
Decimal precision of `DECIMAL` data type	15	38
Decimal precision of `INTEGER` data type	9	38
Decimal precision of `SMALLINT` data type	4	38
Binary precision of `FLOAT` data type	20	126
Binary precision of `REAL` data type	20	63
Binary precision of `DOUBLE PRECISION` data type	30	126
Columns in a table	100	1,000
Values in an `INSERT` statement	100	1,000
SET clauses in an `UPDATE` statement (Note 1)	20	1,000
Length of a row (Note2, Note 3)	2,000	2,000,000
Columns in a `UNIQUE` constraint	6	32
Length of a `UNIQUE` constraint (Note 2)	120	(Note 4)
Length of foreign key column list (Note 2)	120	(Note 4)
Columns in a `GROUP BY` clause	6	255 (Note 5)
Length of `GROUP BY` column list	120	(Note 5)
Sort specifications in `ORDER BY` clause	6	255 (Note 5)
Length of `ORDER BY` column list	120	(Note 5)
Columns in a referential integrity constraint	6	32
Tables referenced in a SQL statement	15	No limit
Cursors simultaneously open	10	(Note 6)
Items in a `SELECT` list	100	1,000

Note 1: The number of `SET` clauses in an `UPDATE` statement refers to the number items separated by commas following the `SET` keyword.

Note 2: The FIPS PUB defines the length of a collection of columns to be the sum of: twice the number of columns, the length of each character column in bytes, decimal precision plus 1 of each exact numeric column, binary precision divided by 4 plus 1 of each approximate numeric column.

Note 3: The Oracle limit for the maximum row length is based on the maximum length of a row containing a `LONG` value of length 2 gigabytes and 999 `VARCHAR2` values, each of length 4000 bytes: 2(254) + 231 + (999(4000)).

Note 4: The Oracle limit for a `UNIQUE` key is half the size of an Oracle data block (specified by the initialization parameter `DB_BLOCK_SIZE`) minus some overhead.

Note 5: Oracle places no limit on the number of columns in a `GROUP BY` clause or the number of sort specifications in an `ORDER BY` clause. However, the sum of the sizes of all the expressions in either a `GROUP BY` clause or an `ORDER BY` clause is limited to the size of an Oracle data block (specified by the initialization parameter `DB_BLOCK_SIZE`) minus some overhead.

Note 6: The Oracle limit for the number of cursors simultaneously opened is specified by the initialization parameter `OPEN_CURSORS`. The maximum value of this parameter depends on the memory available on your operating system and exceeds 100 in all cases.

Oracle Extensions to Standard SQL

Oracle supports numerous features that extend beyond standard SQL. If you are concerned with the portability of your applications to other implementations of SQL, then use Oracle's FIPS Flagger to help identify the use of Oracle extensions to Entry SQL-92 in your embedded SQL programs. The FIPS Flagger is part of the Oracle precompilers and the SQL*Module compiler. The FIPS Flagger can also be enabled in SQL*Plus by using `ALTER SESSION SET FLAGGER = ENTRY`. While SQL-92 has been superseded by SQL:2016, there has been no conformance testing authority for any version of SQL since SQL-92; hence, Entry SQL-92 offers you the most assurance of portability.

> **See Also:**
>
> *Pro*COBOL Programmer's Guide* and *Pro*C/C++ Programmer's Guide* for information on how to use the FIPS Flagger

Oracle Compliance with Older Standards

This release of Oracle Database conforms to SQL:2016, the most recent edition of the SQL standard when this guide was published, as itemized in preceding sections of this appendix. Oracle does not formally claim that this release of the database conforms to SQL-92—and in particular, to SQL-92 Entry Level—or to SQL:1999, because those standards have been superseded by SQL:2016. Some, mostly minor, changes between editions of the SQL standard might affect applications. The SQL standard, or a reference discussing that standard, can be consulted to determine the details of any incompatibilities that have been introduced. One important source is Annex E of SQL/Foundation:1999, SQL/Foundation:2003, SQL/Foundation:2008, SQL/Foundation:2011, and SQL/Foundation:2016.

In some cases, this release of Oracle Database might continue to recognize constructs from older editions of SQL. Such recognition is often allowed as a valid vendor extension. It is the general policy of Oracle to keep incompatibilities between versions of the database as few as possible. This policy extends to retention of older forms when that is feasible. In any case, the differences between older SQL and SQL:2016 (as noted above) are relatively inconsequential.

Character Set Support

Oracle supports most national, international, and vendor-specific encoded character set standards. A complete list of character sets supported by Oracle appears in *Oracle Database Globalization Support Guide*.

Unicode is a universal encoded character set that lets you store information from any language using a single character set. Unicode is required by modern standards such as XML, Java, JavaScript, and LDAP. Unicode is compliant with ISO/IEC standard 10646. For information on ISO standards, visit the Web site of the International Organization for Standardization:

`http://www.iso.ch/`

Oracle Database 12*c* Release 2 (12.2.0.1) complies with version 7.0 of the Unicode Standard. For up-to-date information on the Unicode Standard, visit the Web site of the Unicode Consortium:

`http://www.unicode.org`

Oracle supports the UTF-8 encoding scheme of the Unicode Standard through the AL32UTF8 character set, the UTF-16BE encoding scheme through the AL16UTF16 character set, and the UTF-16LE encoding scheme through the AL16UTF16LE character set. AL32UTF8 is valid as the client and database character set on ASCII-based platforms. AL16UTF16 is valid as the national (`NCHAR`) character set on all platforms. AL16UTF16LE is not valid as the client, database, or national character set.

Oracle implements two deprecated Unicode compatibility encoding forms: CESU-8 through the UTF8 character set and UTF-EBCDIC through the UTFE character set. The UTF8 and UTFE character sets are not guaranteed to include updates to the Unicode standard beyond version 3.0. UTF8 is valid as the client and database character set on ASCII-based platforms and as the national (`NCHAR`) character set on all platforms. UTFE is valid as the database character set on EBCDIC-based platforms.

All mentioned Oracle character sets are supported in conversion functions.

Oracle recommends that databases on ASCII-based platforms are created with the AL32UTF8 character set and the AL16UTF16 national (`NCHAR`) character set. Oracle recommends that you avoid the use of the `NCHAR` data types and the associated national character set as they are not supported by some RDBMS components, such as Oracle Text and Oracle XDB.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for details on Oracle character set support

D

Oracle Regular Expression Support

Oracle's implementation of regular expressions conforms with the IEEE Portable Operating System Interface (POSIX) regular expression standard and to the Unicode Regular Expression Guidelines of the Unicode Consortium.

This appendix contains the following sections:

- Multilingual Regular Expression Syntax
- Regular Expression Operator Multilingual Enhancements
- Perl-influenced Extensions in Oracle Regular Expressions

Multilingual Regular Expression Syntax

Table D-1 lists the full set of operators defined in the POSIX standard Extended Regular Expression (ERE) syntax. Oracle follows the exact syntax and matching semantics for these operators as defined in the POSIX standard for matching ASCII (English language) data. For more complete descriptions of the operators, examples of their use, and Oracle multilingual enhancements of the operators, refer to *Oracle Database Development Guide*. Notes following the table provide more complete descriptions of the operators and their functions, as well as Oracle multilingual enhancements of the operators. Table D-2 summarizes Oracle support for and multilingual enhancement of the POSIX operators.

Table D-1 Regular Expression Operators and Metasymbols

Operator	Description
\	The backslash character can have four different meanings depending on the context. It can: • Stand for itself • Quote the next character • Introduce an operator • Do nothing
*	Matches zero or more occurrences
+	Matches one or more occurrences
?	Matches zero or one occurrence
		Alternation operator for specifying alternative matches
^	Matches the beginning of a string by default. In multiline mode, it matches the beginning of any line anywhere within the source string.
$	Matches the end of a string by default. In multiline mode, it matches the end of any line anywhere within the source string.
.	Matches any character in the supported character set except `NULL`

Table D-1 (Cont.) Regular Expression Operators and Metasymbols

Operator	Description
[ ]	Bracket expression for specifying a matching list that should match any one of the expressions represented in the list. A non-matching list expression begins with a circumflex (^) and specifies a list that matches any character except for the expressions represented in the list. To specify a right bracket (]) in the bracket expression, place it first in the list (after the initial circumflex (^), if any). To specify a hyphen in the bracket expression, place it first in the list (after the initial circumflex (^), if any), last in the list, or as an ending range point in a range expression.
( )	Grouping expression, treated as a single subexpression
{m}	Matches exactly m times
{m,}	Matches at least m times
{m,n}	Matches at least m times but no more than n times
\n	The backreference expression (n is a digit between 1 and 9) matches the n^{th} subexpression enclosed between '(' and ')' preceding the \n
[..]	Specifies one collation element, and can be a multicharacter element (for example, [.ch.] in Spanish)
[: :]	Specifies character classes (for example, [:alpha:]). It matches any character within the character class.
[==]	Specifies equivalence classes. For example, [=a=] matches all characters having base letter 'a'.

Regular Expression Operator Multilingual Enhancements

When applied to multilingual data, Oracle's implementation of the POSIX operators extends beyond the matching capabilities specified in the POSIX standard. Table D-2 shows the relationship of the operators in the context of the POSIX standard.

- The first column lists the supported operators.
- The second and third columns indicate whether the POSIX standard (Basic Regular Expression—BRE and Extended Regular Expression—ERE, respectively) defines the operator
- The fourth column indicates whether Oracle's implementation extends the operator's semantics for handling multilingual data.

Oracle lets you enter multibyte characters directly, if you have a direct input method, or you can use functions to compose the multibyte characters. You cannot use the Unicode hexadecimal encoding value of the form '`\xxxx`'. Oracle evaluates the characters based on the byte values used to encode the character, not the graphical representation of the character. All accented characters are considered word characters.

Table D-2 POSIX and Multilingual Operator Relationships

Operator	POSIX BRE syntax	POSIX ERE Syntax	Multilingual Enhancement
\	Yes	Yes	—
*	Yes	Yes	—
+	--	Yes	—
?	—	Yes	—
|	—	Yes	—
^	Yes	Yes	Yes
$	Yes	Yes	Yes
.	Yes	Yes	Yes
[ ]	Yes	Yes	Yes
( )	Yes	Yes	—
{m}	Yes	Yes	—
{m,}	Yes	Yes	—
{m,n}	Yes	Yes	—
\n	Yes	Yes	Yes
[..]	Yes	Yes	Yes
[::]	Yes	Yes	Yes
[==]	Yes	Yes	Yes

Perl-influenced Extensions in Oracle Regular Expressions

Oracle Database regular expression functions and conditions accept a number of Perl-influenced operators that are in common use, although not part of the POSIX standard. Table D-3 lists those operators. For more complete descriptions with examples, refer to *Oracle Database Development Guide*.

Table D-3 Perl-influenced Operators in Oracle Regular Expressions

Operator	Description
\d	A digit character.
\D	A nondigit character.
\w	A word character.
\W	A nonword character.
\s	A whitespace character.
\S	A non-whitespace character.
\A	Matches only at the beginning of a string, or before a newline character at the end of a string.
\Z	Matches only at the end of a string.
*?	Matches the preceding pattern element 0 or more times (nongreedy).

Table D-3 (Cont.) Perl-influenced Operators in Oracle Regular Expressions

Operator	Description
+?	Matches the preceding pattern element 1 or more times (nongreedy).
??	Matches the preceding pattern element 0 or 1 time (nongreedy).
{n}?	Matches the preceding pattern element exactly *n* times (nongreedy).
{n,}?	Matches the preceding pattern element at least *n* times (nongreedy).
{n,m}?	Matches the preceding pattern element at least *n* but not more than *m* times (nongreedy).

E

Oracle SQL Reserved Words and Keywords

This appendix contains the following sections:

- Oracle SQL Reserved Words
- Oracle SQL Keywords

Oracle SQL Reserved Words

This section lists Oracle SQL reserved words. You cannot use Oracle SQL reserved words as nonquoted identifiers. Quoted identifiers can be reserved words, although this is not recommended.

> **Note:**
>
> In addition to the following reserved words, Oracle uses system-generated names beginning with "SYS_" for implicitly generated schema objects and subobjects. Oracle discourages you from using this prefix in the names you explicitly provide to your schema objects and subobjects to avoid possible conflict in name resolution.

The V$RESERVED_WORDS data dictionary view provides additional information on each reserved word, including whether it is always reserved or is reserved only for particular uses. Refer to *Oracle Database Reference* for more information.

Words followed by an asterisk (*) are also ANSI reserved words.

ACCESS
ADD
ALL *
ALTER *
AND *
ANY *
AS *
ASC
AUDIT
BETWEEN *
BY *
CHAR *
CHECK *
CLUSTER
COLUMN *
COLUMN_VALUE (See Note 1 at the end of this list)
COMMENT
COMPRESS

CONNECT *
CREATE *
CURRENT *
DATE *
DECIMAL *
DEFAULT *
DELETE *
DESC
DISTINCT *
DROP *
ELSE *
EXCLUSIVE
EXISTS *
FILE
FLOAT *
FOR *
FROM *
GRANT *
GROUP *
HAVING *
IDENTIFIED
IMMEDIATE
IN *
INCREMENT
INDEX
INITIAL
INSERT *
INTEGER *
INTERSECT *
INTO *
IS *
LEVEL
LIKE *
LOCK
LONG
MAXEXTENTS
MINUS
MLSLABEL
MODE
MODIFY
NESTED_TABLE_ID (See Note 1 at the end of this list)
NOAUDIT
NOCOMPRESS
NOT *
NOWAIT
NULL *
NUMBER
OF *
OFFLINE

ON *
ONLINE
OPTION
OR *
ORDER *
PCTFREE
PRIOR
PUBLIC
RAW
RENAME
RESOURCE
REVOKE *
ROW *
ROWID (See Note 2 at the end of this list)
ROWNUM
ROWS *
SELECT *
SESSION
SET *
SHARE
SIZE
SMALLINT *
START *
SUCCESSFUL
SYNONYM
SYSDATE
TABLE *
THEN *
TO *
TRIGGER *
UID
UNION *
UNIQUE *
UPDATE *
USER *
VALIDATE
VALUES *
VARCHAR *
VARCHAR2
VIEW
WHENEVER *
WHERE *
WITH *

Note 1: This keyword is only reserved for use as an attribute name.

Note 2: You cannot use the uppercase word `ROWID`, either quoted or nonquoted, as a column name. However, you can use the uppercase word as a quoted identifier that is not a column name, and you can use the word with one or more lowercase letters (for example, "Rowid" or "rowid") as any quoted identifier, including a column name.

Oracle SQL Keywords

Oracle SQL keywords are not reserved. However, Oracle uses them internally in specific ways. Therefore, if you use these words as names for objects and object parts, then your SQL statements may be more difficult to read and may lead to unpredictable results.

You can obtain a list of keywords by querying the `V$RESERVED_WORDS` data dictionary view. All keywords in the view that are not listed as always reserved or reserved for a specific use are Oracle SQL keywords. Refer to *Oracle Database Reference* for more information.

F

Extended Examples

The body of the *SQL Language Reference* contains examples for almost every reference topic. This appendix contains lengthy examples that are not appropriate in the context of a single SQL statement. These examples are intended to provide uninterrupted the series of steps that you would use to take advantage of particular Oracle functionality. They do not replace the syntax diagrams and semantics found for each individual SQL statement in the body of the reference. Use the cross-references provided to access additional information, such as privileges required and restrictions, as well as syntax.

This appendix contains the following sections:

- Using Extensible Indexing
- Using XML in SQL Statements

Using Extensible Indexing

This section provides examples of the steps entailed in a simple but realistic extensible indexing scenario.

Suppose you want to rank the salaries in the `HR.employees` table and then find those that rank between 10 and 20. You could use the `DENSE_RANK` function, as follows:

```
SELECT last_name, salary FROM
   (SELECT last_name, DENSE_RANK() OVER
      (ORDER BY salary DESC) rank_val, salary FROM employees)
   WHERE rank_val BETWEEN 10 AND 20;
```

> **See Also:**
>
> DENSE_RANK

This nested query is somewhat complex, and it requires a full scan of the `employees` table as well as a sort. An alternative would be to use extensible indexing to achieve the same goal. The resulting query will be simpler. The query will require only an index scan and a table access by rowid, and will therefore perform much more efficiently.

The first step is to create the implementation type `position_im`, including method headers for index definition, maintenance, and creation. Most of the type body uses PL/SQL, which is shown in italics.

The type must created with the `AUTHID CURRENT_USER` clause because of the `EXECUTE IMMEDIATE` statement inside the function `ODCIINDEXCREATE()`. By default that function runs with the definer rights. When the function is called in the subsequent creation of the domain index, the invoker does not have the same rights.

> **See Also:**
>
> - CREATE TYPE and CREATE TYPE BODY
> - *Oracle Database Data Cartridge Developer's Guide* for complete information on the ODCI routines in this statement

```
CREATE OR REPLACE TYPE position_im AUTHID CURRENT_USER AS OBJECT
(
  curnum  NUMBER,
  howmany NUMBER,
  lower_bound NUMBER,
  upper_bound NUMBER,
/* lower_bound and upper_bound are used for the
index-based functional implementation */
  STATIC FUNCTION ODCIGETINTERFACES(ifclist OUT SYS.ODCIOBJECTLIST) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXCREATE
    (ia SYS.ODCIINDEXINFO, parms VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXTRUNCATE (ia SYS.ODCIINDEXINFO,
                                     env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXDROP(ia SYS.ODCIINDEXINFO,
                                env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXINSERT(ia SYS.ODCIINDEXINFO, rid ROWID,
                                  newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXDELETE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXUPDATE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXSTART(SCTX IN OUT position_im, ia SYS.ODCIINDEXINFO,
                                 op SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO,
                                 strt NUMBER, stop NUMBER, lower_pos NUMBER,
                                 upper_pos NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  MEMBER FUNCTION ODCIINDEXFETCH(SELF IN OUT position_im, nrows NUMBER,
                                 rids OUT SYS.ODCIRIDLIST, env SYS.ODCIEnv)
                                 RETURN NUMBER,
  MEMBER FUNCTION ODCIINDEXCLOSE(env SYS.ODCIEnv) RETURN NUMBER
);
/

CREATE OR REPLACE TYPE BODY position_im
IS
   STATIC FUNCTION ODCIGETINTERFACES(ifclist OUT SYS.ODCIOBJECTLIST)
       RETURN NUMBER IS
   BEGIN
       ifclist := SYS.ODCIOBJECTLIST(SYS.ODCIOBJECT('SYS','ODCIINDEX2'));
       RETURN ODCICONST.SUCCESS;
   END ODCIGETINTERFACES;
 STATIC FUNCTION ODCIINDEXCREATE (ia SYS.ODCIINDEXINFO, parms VARCHAR2, env SYS.ODCIEnv) RETURN
 NUMBER
  IS
   stmt   VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
   stmt := 'Create Table ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
           '_STORAGE_TAB' || '(col_val, base_rowid, constraint pk PRIMARY KEY ' ||
           '(col_val, base_rowid)) ORGANIZATION INDEX AS SELECT ' ||
           ia.INDEXCOLS(1).COLNAME || ', ROWID FROM ' ||
           ia.INDEXCOLS(1).TABLESCHEMA || '.' || ia.INDEXCOLS(1).TABLENAME;
```

```
    EXECUTE IMMEDIATE stmt;
    RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXDROP(ia SYS.ODCIINDEXINFO, env SYS.ODCIEnv) RETURN NUMBER IS
    stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
    stmt := 'DROP TABLE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
    '_STORAGE_TAB';
/* Execute the statement */
    EXECUTE IMMEDIATE stmt;
    RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXTRUNCATE(ia SYS.ODCIINDEXINFO, env SYS.ODCIEnv) RETURN NUMBER IS
    stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
    stmt := 'TRUNCATE TABLE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME || '_STORAGE_TAB';

    EXECUTE IMMEDIATE stmt;
    RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXINSERT(ia SYS.ODCIINDEXINFO, rid ROWID,
                        newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
    stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
    stmt := 'INSERT INTO ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
           '_STORAGE_TAB  VALUES (''' || newval || ''' , ''' || rid || ''' )';
/* Execute the SQL statement */
    EXECUTE IMMEDIATE stmt;
    RETURN ODCICONST.SUCCESS;
  END;

  STATIC FUNCTION ODCIINDEXDELETE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                env SYS.ODCIEnv)
      RETURN NUMBER IS
    stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
    stmt := 'DELETE FROM ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
           '_STORAGE_TAB WHERE col_val = ''' || oldval || ''' AND base_rowid = ''' || rid || '''';
/* Execute the statement */
    EXECUTE IMMEDIATE stmt;
    RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXUPDATE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                        newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
    stmt VARCHAR2(2000);
  BEGIN
/* Construct the SQL statement */
    stmt := 'UPDATE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
           '_STORAGE_TAB SET col_val = ''' || newval || ''' WHERE f2 = '''|| rid ||'''';
/* Execute the statement */
    EXECUTE IMMEDIATE stmt;
    RETURN ODCICONST.SUCCESS;
  END;
  STATIC FUNCTION ODCIINDEXSTART(SCTX IN OUT position_im, ia SYS.ODCIINDEXINFO,
                       op SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO,
                       strt NUMBER, stop NUMBER, lower_pos NUMBER,
                       upper_pos NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
```

```
    rid             VARCHAR2(5072);
    storage_tab_name VARCHAR2(65);
    lower_bound_stmt VARCHAR2(2000);
    upper_bound_stmt VARCHAR2(2000);
    range_query_stmt VARCHAR2(2000);
    lower_bound      NUMBER;
    upper_bound      NUMBER;
    cnum             INTEGER;
    nrows            INTEGER;

  BEGIN
/* Take care of some error cases.
    The only predicates in which position operator can appear are
       op() = 1     OR
       op() = 0     OR
       op() between 0 and 1
*/
    IF (((strt != 1) AND (strt != 0)) OR
        ((stop != 1) AND (stop != 0)) OR
        ((strt = 1) AND (stop = 0))) THEN
      RAISE_APPLICATION_ERROR(-20101,
                              'incorrect predicate for position_between operator');
    END IF;
    IF (lower_pos > upper_pos) THEN
      RAISE_APPLICATION_ERROR(-20101, 'Upper Position must be greater than or
      equal to Lower Position');
    END IF;
    IF (lower_pos <= 0) THEN
      RAISE_APPLICATION_ERROR(-20101, 'Both Positions must be greater than zero');
    END IF;
    storage_tab_name := ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
                        '_STORAGE_TAB';
    upper_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                        storage_tab_name || ') */ DISTINCT ' ||
                        'col_val FROM ' || storage_tab_name || ' ORDER BY ' ||
                        'col_val DESC) WHERE rownum <= ' || lower_pos;
    EXECUTE IMMEDIATE upper_bound_stmt INTO upper_bound;
    IF (lower_pos != upper_pos) THEN
      lower_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                          storage_tab_name || ') */ DISTINCT ' ||
                          'col_val FROM ' || storage_tab_name ||
                          ' WHERE col_val < ' || upper_bound || ' ORDER BY ' ||
                          'col_val DESC) WHERE rownum <= ' ||
                          (upper_pos - lower_pos);
      EXECUTE IMMEDIATE lower_bound_stmt INTO lower_bound;
    ELSE
      lower_bound := upper_bound;
    END IF;
    IF (lower_bound IS NULL) THEN
      lower_bound := upper_bound;
    END IF;
    range_query_stmt := 'Select base_rowid FROM ' || storage_tab_name ||
                        ' WHERE col_val BETWEEN ' || lower_bound || ' AND ' ||
                        upper_bound;
    cnum := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cnum, range_query_stmt, DBMS_SQL.NATIVE);
/* set context as the cursor number */
    SCTX := position_im(cnum, 0, 0, 0);
/* return success */
    RETURN ODCICONST.SUCCESS;
  END;
```

```
  MEMBER FUNCTION ODCIINDEXFETCH(SELF IN OUT position_im, nrows NUMBER,
                               rids OUT SYS.ODCIRIDLIST, env SYS.ODCIEnv)
   RETURN NUMBER IS
    cnum    INTEGER;
    rid_tab DBMS_SQL.Varchar2_table;
    rlist   SYS.ODCIRIDLIST := SYS.ODCIRIDLIST();
    i       INTEGER;
    d       INTEGER;
  BEGIN
    cnum := SELF.curnum;
    IF self.howmany = 0 THEN
      dbms_sql.define_array(cnum, 1, rid_tab, nrows, 1);
      d := DBMS_SQL.EXECUTE(cnum);
    END IF;
    d := DBMS_SQL.FETCH_ROWS(cnum);
    IF d = nrows THEN
      rlist.extend(d);
    ELSE
      rlist.extend(d+1);
    END IF;
    DBMS_SQL.COLUMN_VALUE(cnum, 1, rid_tab);
    for i in 1..d loop
      rlist(i) := rid_tab(i+SELF.howmany);
    end loop;
    SELF.howmany := SELF.howmany + d;
    rids := rlist;
    RETURN ODCICONST.SUCCESS;
  END;
  MEMBER FUNCTION ODCIINDEXCLOSE(env SYS.ODCIEnv) RETURN NUMBER IS
    cnum INTEGER;
  BEGIN
    cnum := SELF.curnum;
    DBMS_SQL.CLOSE_CURSOR(cnum);
    RETURN ODCICONST.SUCCESS;
  END;
END;
/
```

The next step is to create the functional implementation `function_for_position_between` for the operator that will be associated with the indextype. (The PL/SQL blocks are shown in parentheses.)

This function is for use with an index-based function evaluation. Therefore, it takes an index context and scan context as parameters.

See Also:

- *Oracle Database Data Cartridge Developer's Guide* for information on creating index-based functional implementation
- CREATE FUNCTION and *Oracle Database PL/SQL Language Reference*

```
CREATE OR REPLACE FUNCTION function_for_position_between
                           (col NUMBER, lower_pos NUMBER, upper_pos NUMBER,
                            indexctx IN SYS.ODCIIndexCtx,
                            scanctx IN OUT position_im,
                            scanflg IN NUMBER)
```

```
RETURN NUMBER AS
  rid               ROWID;
  storage_tab_name VARCHAR2(65);
  lower_bound_stmt VARCHAR2(2000);
  upper_bound_stmt VARCHAR2(2000);
  col_val_stmt     VARCHAR2(2000);
  lower_bound      NUMBER;
  upper_bound      NUMBER;
  column_value     NUMBER;
BEGIN
  IF (indexctx.IndexInfo IS NOT NULL) THEN
    storage_tab_name := indexctx.IndexInfo.INDEXSCHEMA || '.' ||
                        indexctx.IndexInfo.INDEXNAME || '_STORAGE_TAB';
    IF (scanctx IS NULL) THEN
/* This is the first call. Open a cursor for future calls.
   First, do some error checking
*/
      IF (lower_pos > upper_pos) THEN
        RAISE_APPLICATION_ERROR(-20101,
          'Upper Position must be greater than or equal to Lower Position');
      END IF;
      IF (lower_pos <= 0) THEN
        RAISE_APPLICATION_ERROR(-20101,
          'Both Positions must be greater than zero');
      END IF;
/* Obtain the upper and lower value bounds for the range we're interested in.
*/
      upper_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                        storage_tab_name || ') */ DISTINCT ' ||
                        'col_val FROM ' || storage_tab_name || ' ORDER BY ' ||
                        'col_val DESC) WHERE rownum <= ' || lower_pos;
      EXECUTE IMMEDIATE upper_bound_stmt INTO upper_bound;
      IF (lower_pos != upper_pos) THEN
        lower_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                            storage_tab_name || ') */ DISTINCT ' ||
                            'col_val FROM ' || storage_tab_name ||
                            ' WHERE col_val < ' || upper_bound || ' ORDER BY ' ||
                            'col_val DESC) WHERE rownum <= ' ||
                            (upper_pos - lower_pos);
        EXECUTE IMMEDIATE lower_bound_stmt INTO lower_bound;
      ELSE
        lower_bound := upper_bound;
      END IF;
      IF (lower_bound IS NULL) THEN
        lower_bound := upper_bound;
      END IF;
/* Store the lower and upper bounds for future function invocations for
   the positions.
*/
      scanctx := position_im(0, 0, lower_bound, upper_bound);
    END IF;
/* Fetch the column value corresponding to the rowid, and see if it falls
   within the determined range.
*/
    col_val_stmt := 'Select col_val FROM ' || storage_tab_name ||
                    ' WHERE base_rowid = ''' || indexctx.Rid || '''';
    EXECUTE IMMEDIATE col_val_stmt INTO column_value;
    IF (column_value <= scanctx.upper_bound AND
        column_value >= scanctx.lower_bound AND
        scanflg = ODCICONST.RegularCall) THEN
      RETURN 1;
```

```
    ELSE
      RETURN 0;
    END IF;
  ELSE
    RAISE_APPLICATION_ERROR(-20101, 'A column that has a domain index of' ||
                            'Position indextype must be the first argument');
  END IF;
END;
/
```

Next, create the position_between operator, which uses the function_for_position_between function. The operator takes an indexed NUMBER column as the first argument, followed by a NUMBER lower and upper bound as the second and third arguments.

See Also:

CREATE OPERATOR

```
CREATE OR REPLACE OPERATOR position_between
   BINDING (NUMBER, NUMBER, NUMBER) RETURN NUMBER
   WITH INDEX CONTEXT, SCAN CONTEXT position_im
   USING function_for_position_between;
```

In this CREATE OPERATOR statement, the WITH INDEX CONTEXT, SCAN CONTEXT position_im clause is included so that the index context and scan context are passed in to the functional evaluation, which is index based.

Now create the position_indextype indextype for the position_operator:

See Also:

CREATE INDEXTYPE

```
CREATE INDEXTYPE position_indextype
   FOR position_between(NUMBER, NUMBER, NUMBER)
   USING position_im;
```

The operator position_between uses an index-based functional implementation. Therefore, a domain index must be defined on the referenced column so that the index information can be passed into the functional evaluation. So the final step is to create the domain index salary_index using the position_indextype indextype:

See Also:

CREATE INDEX

```
CREATE INDEX salary_index ON employees(salary)
   INDEXTYPE IS position_indextype;
```

Now you can use the `position_between` operator function to rewrite the original query as follows:

```
SELECT last_name, salary FROM employees
   WHERE position_between(salary, 10, 20)=1
   ORDER BY salary DESC, last_name;

LAST_NAME                     SALARY
------------------------- ----------
Tucker                         10000
King                           10000
Baer                           10000
Bloom                          10000
Fox                             9600
Bernstein                       9500
Sully                           9500
Greene                          9500
Hunold                          9000
Faviet                          9000
McEwen                          9000
Hall                            9000
Hutton                          8800
Taylor                          8600
Livingston                      8400
Gietz                           8300
Chen                            8200
Fripp                           8200
Weiss                           8000
Olsen                           8000
Smith                           8000
Kaufling                        7900
```

Using XML in SQL Statements

This section describes some of the ways you can use `XMLType` data in the database.

XMLType Tables

The sample schema `oe` contains a table `warehouses`, which contains an `XMLType` column `warehouse_spec`. Suppose you want to create a separate table with the `warehouse_spec` information. The following example creates a very simple `XMLType` table with one `CLOB` column:

```
CREATE TABLE xwarehouses OF XMLTYPE
  XMLTYPE STORE AS CLOB;
```

You can insert into such a table using `XMLType` syntax, as shown in the next statement. (The data inserted in this example corresponds to the data in the `warehouse_spec` column of the sample table `oe.warehouses` where `warehouse_id` = 1.)

```
INSERT INTO xwarehouses VALUES
  (xmltype('<?xml version="1.0"?>
  <Warehouse>
    <WarehouseId>1</WarehouseId>
    <WarehouseName>Southlake, Texas</WarehouseName>
```

```
    <Building>Owned</Building>
    <Area>25000</Area>
    <Docks>2</Docks>
    <DockType>Rear load</DockType>
    <WaterAccess>true</WaterAccess>
    <RailAccess>N</RailAccess>
    <Parking>Street</Parking>
    <VClearance>10</VClearance>
  </Warehouse>'));
```

See Also:

Oracle XML DB Developer's Guide for information on `XMLType` and its member methods

You can query this table with the following statement:

```
SELECT e.getClobVal() FROM xwarehouses e;
```

`CLOB` columns are subject to all of the restrictions on LOB columns. To avoid these restrictions, create an XMLSchema-based table. The XMLSchema maps the XML elements to their object-relational equivalents. The following example registers an XMLSchema locally. The XMLSchema (`xwarhouses.xsd`) reflects the same structure as the `xwarehouses` table. (XMLSchema declarations use PL/SQL and the `DBMS_XMLSCHEMA` package, so the example is shown in italics.)

See Also:

Oracle XML DB Developer's Guide for information on creating XMLSchemas

```
begin
 dbms_xmlschema.registerSchema(
  'http://www.example.com/xwarehouses.xsd',
  '<schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.example.com/xwarehouses.xsd"
       xmlns:who="http://www.example.com/xwarehouses.xsd"
       version="1.0">

  <simpleType name="RentalType">
   <restriction base="string">
    <enumeration value="Rented"/>
    <enumeration value="Owned"/>
   </restriction>
  </simpleType>

  <simpleType name="ParkingType">
   <restriction base="string">
    <enumeration value="Street"/>
    <enumeration value="Lot"/>
   </restriction>
  </simpleType>

  <element name = "Warehouse">
```

```
    <complexType>
     <sequence>
      <element name = "WarehouseId"   type = "positiveInteger"/>
      <element name = "WarehouseName" type = "string"/>
      <element name = "Building"      type = "who:RentalType"/>
      <element name = "Area"          type = "positiveInteger"/>
      <element name = "Docks"         type = "positiveInteger"/>
      <element name = "DockType"      type = "string"/>
      <element name = "WaterAccess"   type = "boolean"/>
      <element name = "RailAccess"    type = "boolean"/>
      <element name = "Parking"       type = "who:ParkingType"/>
      <element name = "VClearance"    type = "positiveInteger"/>
     </sequence>
    </complexType>
  </element>
</schema>',
   TRUE, TRUE, FALSE, FALSE);
end;
/
```

Now you can create an XMLSchema-based table, as shown in the following example:

```
CREATE TABLE xwarehouses OF XMLTYPE
   XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
   ELEMENT "Warehouse";
```

By default, Oracle stores this as an object-relational table. Therefore, you can insert into it as shown in the example that follows. (The data inserted in this example corresponds to the data in the `warehouse_spec` column of the sample table `oe.warehouses` where `warehouse_id` = 1.)

```
INSERT INTO xwarehouses VALUES(   xmltype.createxml('<?xml version="1.0"?>
   <who:Warehouse xmlns:who="http://www.example.com/xwarehouses.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.example.com/xwarehouses.xsd
   http://www.example.com/xwarehouses.xsd">
      <WarehouseId>1</WarehouseId>
      <WarehouseName>Southlake, Texas</WarehouseName>
      <Building>Owned</Building>
      <Area>25000</Area>
      <Docks>2</Docks>
      <DockType>Rear load</DockType>
      <WaterAccess>true</WaterAccess>
      <RailAccess>false</RailAccess>
      <Parking>Street</Parking>
      <VClearance>10</VClearance>
      </who:Warehouse>'));
...
```

You can define constraints on an XMLSchema-based table. To do so, you use the `XMLDATA` pseudocolumn to refer to the appropriate attribute within the `Warehouse` XML element:

```
ALTER TABLE xwarehouses ADD (PRIMARY KEY(XMLDATA."WarehouseId"));
```

Because the data in `xwarehouses` is stored object relationally, Oracle rewrites queries to this `XMLType` table to go to the underlying storage when possible. Therefore the following queries would use the index created by the primary key constraint in the preceding example:

```
SELECT * FROM xwarehouses x
   WHERE EXISTSNODE(VALUE(x), '/Warehouse[WarehouseId="1"]',
   'xmlns:who="http://www.example.com/xwarehouses.xsd"') = 1;

SELECT * FROM xwarehouses x
   WHERE EXTRACTVALUE(VALUE(x), '/Warehouse/WarehouseId',
   'xmlns:who="http://www.example.com/xwarehouses.xsd"') = 1;
```

You can also explicitly create indexes on XMLSchema-based tables, which greatly enhance the performance of subsequent queries. You can create object-relational views on `XMLType` tables, and you can create `XMLType` views on object-relational tables.

See Also:

- XMLDATA Pseudocolumn for information on the `XMLDATA` pseudocolumn
- "Creating an XMLType View: Example"
- Creating an Index on an XMLType Table: Example

XMLType Columns

The sample table `oe.warehouses` was created with a `warehouse_spec` column of type `XMLType`. The examples in this section create a shortened form of the `oe.warehouses` table, using two different types of storage.

The first example creates a table with an `XMLType` table stored as a `CLOB`. This table does not require an XMLSchema, so the content structure is not predetermined:

```
CREATE TABLE xwarehouses (
   warehouse_id        NUMBER,
   warehouse_spec      XMLTYPE)
   XMLTYPE warehouse_spec STORE AS CLOB
   (TABLESPACE example
    STORAGE (INITIAL 6144)
    CHUNK 4000
    NOCACHE LOGGING);
```

The following example creates a similar table, but stores the `XMLType` data in an object-relational `XMLType` column whose structure is determined by the specified XMLSchema:

```
CREATE TABLE xwarehouses (
   warehouse_id    NUMBER,
   warehouse_spec  XMLTYPE)
   XMLTYPE warehouse_spec STORE AS OBJECT RELATIONAL
      XMLSCHEMA "http://www.example.com/xwarehouses.xsd"
      ELEMENT "Warehouse";
```

Index

Symbols

+ (plus sign) in Oracle Automatic Storage Management filenames, *8-32*

Numerics

20th century, *2-78*
21st century, *2-78*
3GL functions and procedures, calling, *14-1*

A

ABORT LOGICAL STANDBY clause
  of ALTER DATABASE, *10-81*
ABS function, *7-22*
ACCESSED GLOBALLY clause
  of CREATE CONTEXT, *13-43*
ACCOUNT LOCK clause
  of ALTER USER. *See* CREATE USER, *12-197*
  of CREATE USER, *15-176*
ACCOUNT UNLOCK clause
  of ALTER USER. *See* CREATE USER, *12-197*
  of CREATE USER, *15-176*
ACOS function, *7-22*
ACTIVATE STANDBY DATABASE clause
  of ALTER DATABASE, *10-76*
AD and A.D. datetime format elements, *2-77*
ADD clause
  of ALTER DIMENSION, *10-95*
  of ALTER INDEXTYPE, *10-160*
  of ALTER TABLE, *12-29*
  of ALTER VIEW, *12-208*
ADD DATAFILE clause
  of ALTER TABLESPACE, *12-171*
ADD LOGFILE clause
  of ALTER DATABASE, *10-46*
ADD LOGFILE GROUP clause
  of ALTER DATABASE, *10-71*
ADD LOGFILE INSTANCE clause
  of ALTER DATABASE, *10-70*
ADD LOGFILE MEMBER clause
  of ALTER DATABASE, *10-46*, *10-71*
ADD LOGFILE THREAD clause
  of ALTER DATABASE, *10-70*
ADD OVERFLOW clause
  of ALTER TABLE, *12-29*
ADD PARTITION clause
  of ALTER TABLE, *12-29*
ADD PRIMARY KEY clause
  of ALTER MATERIALIZED VIEW LOG, *11-41*
ADD ROWID clause
  of ALTER MATERIALIZED VIEW, *11-41*
ADD SUPPLEMENTAL LOG DATA clause
  of ALTER DATABASE, *10-73*
ADD SUPPLEMENTAL LOG GROUP clause
  of ALTER TABLE, *12-85*
ADD TEMPFILE clause
  of ALTER TABLESPACE, *12-171*
ADD VALUES clause
  of ALTER TABLE ... MODIFY PARTITION, *12-123*, *12-124*
ADD_MONTHS function, *7-23*
adding a constraint to a table, *12-113*
ADMIN USER clause
  of CREATE PLUGGABLE DATABASE, *14-78*
ADMINISTER ANY SQL TUNING SET system privilege, *18-41*
ADMINISTER KEY MANAGEMENT statement, *10-5*
ADMINISTER KEY MANAGEMENT system privilege, *18-44*
ADMINISTER SQL MANAGEMENT OBJECT system privilege, *18-41*
ADMINISTER SQL TUNING SET system privilege, *18-41*
advanced index compression
  definition, *13-123*
  disabling, *13-123*
  enabling, *10-148*
  of index rebuild, *10-148*
Advanced Row Compression, *15-70*
ADVISE clause
  of ALTER SESSION, *11-89*
aggregate functions, *7-12*
alias
  for a column, *9-2*
  for an expressions in a view query, *15-185*

alias *(continued)*
specifying in queries and subqueries, *19-39*
ALL clause
of SELECT, *19-59*
of SET CONSTRAINTS, *19-125*
of SET ROLE, *19-128*
ALL operator, *6-4*
ALL PRIVILEGES clause
of GRANT, *18-37*
of REVOKE, *19-29*
ALL PRIVILEGES shortcut
of AUDIT, *12-226*
ALL shortcut
of AUDIT, *12-226*
ALL_COL_COMMENTS data dictionary view, *12-248*
ALL_INDEXTYPE_COMMENTS data dictionary view, *12-249*
ALL_MVIEW_COMMENTS data dictionary view, *12-249*
ALL_OPERATOR_COMMENTS data dictionary view, *12-249*
ALL_ROWS hint, *2-94*
ALL_TAB_COMMENTS data dictionary view, *12-248*
all-column wildcard, *19-59*
ALLOCATE EXTENT clause
of ALTER CLUSTER, *10-36*, *10-37*
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW, *11-21*
of ALTER TABLE, *12-85*
ALLOW CORRUPTION clause
of ALTER DATABASE ... RECOVER, *10-59*
ALTER ANALYTIC VIEW statement, *10-29*
ALTER ANY SQL PROFILE system privilege, *18-41*
ALTER ATTRIBUTE DIMENSION statement, *10-30*
ALTER AUDIT POLICY statement, *10-31*
ALTER CLUSTER statement, *10-36*
ALTER DATABASE LINK system privilege, *18-42*
ALTER DATABASE statement, *10-40*
ALTER DIMENSION statement, *10-92*
ALTER DISKGROUP statement, *10-96*
ALTER FLASHBACK ARCHIVE statement, *10-130*
ALTER FUNCTION statement, *10-133*
ALTER HIERARCHY statement, *10-134*
ALTER INDEX statement, *10-135*
ALTER INDEXTYPE statement, *10-159*
ALTER INMEMORY JOIN GROUP statement, *10-161*
ALTER JAVA CLASS statement, *10-163*
ALTER JAVA SOURCE statement, *10-163*
ALTER LIBRARY statement, *11-1*
ALTER LOCKDOWN PROFILE statement, *11-2*
ALTER MATERIALIZED VIEW LOG statement, *11-35*
ALTER MATERIALIZED VIEW statement, *11-14*
ALTER MATERIALIZED ZONEMAP statement, *11-44*
ALTER object privilege
on a SQL translation profile, *18-56*
ALTER OPERATOR statement, *11-48*
ALTER OUTLINE statement, *11-51*
ALTER PACKAGE statement, *11-52*
ALTER PLUGGABLE DATABASE statement, *11-54*
ALTER PROCEDURE statement, *11-73*
ALTER PROFILE statement, *11-75*
ALTER PUBLIC DATABASE LINK system privilege, *18-42*
ALTER RESOURCE COST statement, *11-78*
ALTER ROLE statement, *11-80*
ALTER ROLLBACK SEGMENT statement, *11-82*
ALTER SEQUENCE statement, *11-85*
ALTER SESSION statement, *11-87*
ALTER SNAPSHOT
See ALTER MATERIALIZED VIEW
ALTER SNAPSHOT LOG
See ALTER MATERIALIZED VIEW LOG
ALTER SYSTEM statement, *12-3*
ALTER TABLE statement, *12-29*
ALTER TABLESPACE SET statement, *12-189*
ALTER TABLESPACE statement, *12-171*
ALTER TRIGGER statement, *12-191*
ALTER TYPE statement, *12-193*
ALTER USER statement, *12-195*
ALTER VIEW statement, *12-206*
alter_external_table clause
of ALTER TABLE, *12-56*
altering storage
of PDBs, *11-62*
AM and A.M. datetime format elements, *2-77*
American National Standards Institute (ANSI), *1-1*
data types, *2-34*
conversion to Oracle data types, *2-35*
standards, *1-1*, *C-1*
supported data types, *2-1*
analytic functions, *7-14*
analytic view measure expression, *5-6*
analytic view simple expression, *5-16*
analytic views
altering, *10-29*
creating, *13-6*
dropping, *15-206*
granting system privileges for, *18-30*
ANALYZE CLUSTER statement, *12-209*
ANALYZE INDEX statement, *12-209*

ANALYZE TABLE statement, *12-209*
ANCILLARY TO clause
of CREATE OPERATOR, *14-60*
AND condition, *6-9*
AND DATAFILES clause
of DROP TABLESPACE, *18-7*
ANSI
See American National Standards Institute (ANSI)
antijoins, *9-15*
ANY operator, *6-4*
APPEND hint, *2-94*
APPEND_VALUES hint, *2-95*
APPENDCHILDXML function, *7-24*
application servers
allowing connection as user, *12-195*
applications
allowing connection as user, *12-195*
securing, *13-42*
validating, *13-42*
APPROX_COUNT_DISTINCT function, *7-25*
APPROX_COUNT_DISTINCT_AGG function, *7-26*
APPROX_COUNT_DISTINCT_DETAIL function, *7-27*
APPROX_MEDIAN function, *7-30*
APPROX_PERCENTILE function, *7-33*
APPROX_PERCENTILE_AGG function, *7-36*
APPROX_PERCENTILE_DETAIL function, *7-37*
ARCHIVE LOG clause
of ALTER SYSTEM, *12-3*
archive mode
specifying, *13-60*
archived redo logs
location, *10-57*
ARCHIVELOG clause
of ALTER DATABASE, *10-46*
of CREATE CONTROLFILE, *13-50*
of CREATE DATABASE, *13-60*
arguments
of operators, *4-1*
arithmetic
with DATE values, *2-24*
arithmetic operators, *4-2*
AS CLONE clause
of CREATE PLUGGABLE DATABASE, *14-89*
AS source_char clause
of CREATE JAVA, *13-148*
AS subquery clause
of CREATE MATERIALIZED VIEW, *14-4*
of CREATE TABLE, *15-121*
of CREATE VIEW, *15-189*
ASC clause
of CREATE INDEX, *13-122*
ASCII function, *7-41*
ASCIISTR function, *7-42*
ASIN function, *7-42*
ASSOCIATE STATISTICS statement, *12-218*
asterisk
all-column wildcard in queries, *19-59*
asynchronous commit, *13-4*
ATAN function, *7-43*
ATAN2 function, *7-44*
ATTRIBUTE clause
of ALTER DIMENSION, *10-94*
of CREATE DIMENSION, *13-75*, *13-76*
attribute clustering, *15-112*
attribute dimensions
altering, *10-30*
creating, *13-14*
dropping, *15-207*
granting system privileges for, *18-30*
attributes
adding to a dimension, *10-95*
dropping from a dimension, *10-95*
maximum number of in object type, *15-53*
of dimensions, defining, *13-75*
of disk groups, *10-96*, *13-88*
audit policies
comments on, *12-248*
creating, *13-23*
dropping, *15-207*
modifying, *10-31*
AUDIT statement, *12-222*
for unified auditing, *12-238*
locks, *B-5*
auditing
options
for SQL statements, *12-222*
policies
value-based, *12-222*
SQL statements, *12-223*
by a proxy, *12-224*
by a user, *12-224*
on a directory, *12-224*
on a schema, *12-224*
stopping, *19-11*
system privileges, *12-223*
AUTHENTICATED BY clause
of CREATE DATABASE LINK, *13-69*
AUTHENTICATED clause
of ALTER USER, *12-204*
AUTHENTICATION REQUIRED clause
of ALTER USER, *12-204*
AUTHID CURRENT_USER clause
of ALTER JAVA, *10-163*
of CREATE JAVA, *13-148*, *13-149*
AUTHID DEFINER clause
of ALTER JAVA, *10-163*
of CREATE JAVA, *13-148*, *13-149*

AUTOALLOCATE clause
of CREATE TABLESPACE, *15-141*
AUTOEXTEND clause
of ALTER DATABASE, *10-46*
of CREATE DATABASE, *13-54*
automatic segment-space management, *15-155*
automatic undo mode, *11-83*, *13-52*
AVG function, *7-45*

B

BACKUP CONTROLFILE clause
of ALTER DATABASE, *10-48*, *10-75*
backups, *10-85*
band joins, *9-13*
basic table compression, *15-70*
BC and B.C. datetime format elements, *2-77*
BECOME USER system privilege, *18-51*
BEGIN BACKUP clause
of ALTER DATABASE, *10-63*
of ALTER TABLESPACE, *12-179*
BEQUEATH clause
of CREATE VIEW, *15-188*
BETWEEN condition, *6-32*
BFILE
data type, *2-30*
locators, *2-30*
BFILENAME function, *7-46*
BIN_TO_NUM function, *7-47*
binary large objects
See BLOB
binary operators, *4-1*
binary XML format, *15-17*
binary XML storage, *15-17*
bindings
adding to an operator, *11-48*
dropping from an operator, *11-50*
bit vectors
converting to numbers, *7-47*
BITAND function, *7-48*
BITMAP clause
of CREATE INDEX, *13-115*
bitmap indexes, *13-115*
creating join indexes, *13-107*
blank padding
specifying in format models, *2-79*
suppressing, *2-79*
BLOB data type, *2-31*
BLOCKSIZE clause
of CREATE TABLESPACE, *15-150*
bottom-N reporting, *7-107*, *7-269*, *7-301*
buffer cache
flushing, *12-11*
BUFFER_POOL parameter
of STORAGE clause, *8-58*
BUILD DEFERRED clause
of CREATE MATERIALIZED VIEW, *14-4*
BUILD IMMEDIATE clause
of CREATE MATERIALIZED VIEW, *14-4*
BY ACCESS clause
of AUDIT, *12-229*
BY SESSION clause
of AUDIT, *12-229*
BYTE character semantics, *2-10*, *2-13*
BYTE length semantics, *12-101*

C

CACHE clause
of ALTER MATERIALIZED VIEW, *11-27*
of ALTER MATERIALIZED VIEW LOG, *11-41*
of ALTER TABLE, *15-114*
of CREATE CLUSTER, *13-40*
of CREATE MATERIALIZED VIEW, *14-22*
of CREATE MATERIALIZED VIEW LOG, *14-43*
CACHE hint, *2-95*
CACHE parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-85*
of CREATE SEQUENCE, *15-5*
CACHE READS clause
of ALTER TABLE, *12-98*
of CREATE TABLE, *15-115*
cached cursors
execution plan for, *18-17*
calculated measure expression, *5-4*, *5-16*
call spec
See call specifications
call specifications,
in procedures, *14-93*
CALL statement, *12-243*
calls
limiting CPU time for, *14-99*
limiting data blocks read, *14-99*
CARDINALITY function, *7-50*
Cartesian products, *9-13*
CASCADE clause
of CREATE TABLE, *15-118*
of DROP PROFILE, *17-15*
of DROP USER, *18-14*
CASCADE CONSTRAINTS clause
of DROP CLUSTER, *15-210*
of DROP TABLE, *18-4*
of DROP TABLESPACE, *18-7*
of DROP VIEW, *18-16*
of REVOKE, *19-30*
CASE expressions, *5-18*
searched, *5-19*
simple, *5-19*

CAST function, 7-51
CATSEARCH condition, 6-2
CATSEARCH operator, 4-1
CDBs,
  creating, 13-67
  modifying, 10-52
CEIL function, 7-56
chained rows
  listing, 12-216
  of clusters, 12-209
CHANGE CATEGORY clause
  of ALTER OUTLINE, 11-52
CHANGE NOTIFICATION system privilege, 18-51
CHANGE_DUPKEY_ERROR_INDEX hint, 2-96
changing state
  of a PDB, 11-65
  of multiple PDBs, 11-68
CHAR character semantics, 2-10, 2-13
CHAR data type, 2-10
  converting to VARCHAR2, 2-68
CHAR length semantics, 12-101
character functions
  returning character values, 7-5
  returning number values, 7-5
character large objects
  See CLOB
character length semantics, 12-101
character literal
  See text
character set
  changing, 10-82
character set functions, 7-6
CHARACTER SET parameter
  of CREATE CONTROLFILE, 13-51
  of CREATE DATABASE, 13-57
character sets
  database, specifying, 13-57
  multibyte characters, 2-142
  specifying for database, 13-57
character strings
  comparison rules, 2-45
  exact matching, 2-79
  fixed-length, 2-10
  national character set, 2-11
  variable-length, 2-11, 2-18
CHARTOROWID function, 7-56
CHECK clause
  of constraints, 8-3
  of CREATE TABLE, 15-17
check constraints, 8-3
CHECK DATAFILES clause
  of ALTER SYSTEM, 12-10
CHECKPOINT clause
  of ALTER SYSTEM, 12-9
checkpoints
  forcing, 12-9
CHR function, 7-57
CHUNK clause
  of ALTER TABLE, 12-99
  of CREATE TABLE, 15-86
CLEAR LOGFILE clause
  of ALTER DATABASE, 10-46, 10-69
CLOB data type, 2-31
clone databases
  mounting, 10-54
CLOSE DATABASE LINK clause
  of ALTER SESSION, 11-89
CLUSTER clause
  of ANALYZE, 12-209
  of CREATE INDEX, 13-105
  of CREATE TABLE, 15-83
  of TRUNCATE, 19-133
CLUSTER hint, 2-96
CLUSTER_DETAILS function, 7-59
CLUSTER_DISTANCE function, 7-62
CLUSTER_ID function, 7-65
CLUSTER_PROBABILITY function, 7-67
CLUSTER_SET function, 7-70
CLUSTERING hint, 2-97
clusters
  assigning tables to, 15-83
  caching retrieved blocks, 13-40
  cluster indexes, 13-105
  collecting statistics on, 12-209
  creating, 13-33
  deallocating unused extents, 10-37
  degree of parallelism
    changing, 10-37, 10-39
    when creating, 13-33
  dropping tables, 15-210
  extents, allocating, 10-36, 10-37
  granting system privileges for, 18-30
  hash, 13-38
    single-table, 13-38
    sorted, 13-36, 15-54
  indexed, 13-38
  key values
    allocating space for, 13-37
    modifying space for, 10-38
  migrated and chained rows in, 12-209, 12-216
  modifying, 10-36
  physical attributes
    changing, 10-37
    specifying, 13-33
  releasing unused space, 10-36
  removing from the database, 15-209
  SQL examples, 15-209

clusters *(continued)*
storage attributes
changing, *10-37*
storage characteristics
changing, *10-36*
tablespace in which created, *13-37*
validating structure, *12-214*
COALESCE clause
for partitions, *12-29*
of ALTER INDEX, *10-151*
of ALTER TABLE, *12-91*, *12-93*, *12-122*
of ALTER TABLESPACE, *12-178*
COALESCE function, *7-72*
as a variety of CASE expression, *7-72*
COALESCE SUBPARTITION clause
of ALTER TABLE, *12-122*
COLLATE operator, *4-3*
COLLATION function, *7-74*
collation functions, *7-6*
COLLECT function, *7-75*
collection functions, *7-8*
collection types
multilevel, *15-91*
collection-typed values
converting to data types, *7-51*
collections
inserting rows into, *18-61*
modifying, *12-29*
modifying retrieval method, *12-47*
nested tables, *2-37*
testing for empty, *6-13*
treating as a table, *18-61*, *19-39*, *19-138*
unnesting, *19-39*
examples, *19-120*
varrays, *2-37*
column expressions, *5-20*
column REF constraints, *8-3*
of CREATE TABLE, *15-17*
column values
unpivoting into rows, *19-69*
COLUMN_VALUE pseudocolumn, *3-7*
columns
adding, *12-29*
aliases for, *9-2*
altering storage, *12-97*
associating statistics types with, *12-218*
basing an index on, *13-105*
comments on, *12-248*
creating comments about, *12-247*
defining, *15-17*
disassociating statistics types from, *15-204*
dropping from a table, *12-105*
LOB
storage attributes, *12-29*
maximum number of, *15-53*
columns *(continued)*
modifying existing, *12-99*
parent-child relationships between, *13-75*
properties, altering, *12-48*, *12-97*
qualifying names of, *9-2*
REF
describing, *8-3*
renaming, *12-29*
restricting values for, *8-3*
specifying
as primary key, *8-3*
constraints on, *15-17*
default values, *15-55*
storage properties, *15-83*
substitutable, identifying type, *7-346*
virtual
adding to a table, *12-96*
creating, *15-17*
modifying, *12-96*
COLUMNS clause
of ASSOCIATE STATISTICS, *12-218*, *12-220*
of DISASSOCIATE STATISTICS, *15-204*
COMMENT clause
of COMMIT, *13-3*
COMMENT statement, *12-247*
comments, *2-86*
adding to objects, *12-247*
associating with a transaction, *13-5*
dropping from objects, *12-247*
in SQL statements, *2-86*
on editions, *12-248*
on indextypes, *12-249*
on mining models, *12-249*
on operators, *12-249*
on schema objects, *2-87*
on table columns, *12-248*
on tables, *12-249*
on unified audit policies, *12-248*
removing from the data dictionary, *12-247*
specifying, *2-86*
viewing, *12-248*
commit
asynchronous, *13-4*
automatic, *13-2*
COMMIT IN PROCEDURE clause
of ALTER SESSION, *11-89*
COMMIT statement, *13-1*
COMMIT TO SWITCHOVER clause
of ALTER DATABASE, *10-79*
comparison conditions, *6-4*
comparison functions, *7-7*
comparison semantics
of character strings, *2-45*
COMPILE clause
of ALTER DIMENSION, *10-95*

COMPILE clause *(continued)*
of ALTER JAVA SOURCE, *10-165*
of ALTER MATERIALIZED VIEW, *11-33*
of ALTER VIEW, *12-208*
of CREATE JAVA, *13-149*
COMPOSE function, *7-76*
composite foreign keys, *8-3*
composite partitioning
range-list, *12-29*, *15-108*
when creating a table, *15-43*, *15-105*
composite primary keys, *8-3*
composite range partitions, *15-105*
COMPOSITE_LIMIT parameter
of ALTER PROFILE, *11-75*
of CREATE PROFILE, *14-99*
compound conditions, *6-31*
compound expressions, *5-3*
COMPRESS clause
of ALTER INDEX ... REBUILD, *13-123*
of CREATE TABLE, *15-79*
compression
of index keys, *10-138*
of tables, *15-69*
of tablespaces, *15-69*
CON_DBID_TO_ID function, *7-77*
CON_GUID_TO_ID function, *7-78*
CON_NAME_TO_ID function, *7-79*
CON_UID_TO_ID function, *7-79*
CONCAT function, *7-80*
concatenation operator, *4-4*
conditions
BETWEEN, *6-32*
comparison, *6-4*
compound, *6-31*
EXISTS, *6-22*, *6-33*
floating-point, *6-8*
group comparison, *6-7*
IN, *6-33*
in SQL syntax, *6-1*
interval, *6-32*
IS ANY, *6-10*
IS JSON, *6-24*
IS OF type, *6-36*
IS PRESENT, *6-11*
JSON_EXISTS, *6-26*
JSON_TEXTCONTAINS, *6-29*
LIKE, *6-15*
logical, *6-9*
MEMBER, *6-14*
membership, *6-14*, *6-33*
model, *6-10*
multiset, *6-12*
IS [NOT] EMPTY, *6-13*
null, *6-21*
pattern matching, *6-15*
conditions *(continued)*
range, *6-32*
REGEXP_LIKE, *6-19*
SET, *6-12*
simple comparison, *6-5*
SQL/JSON, *6-23*
SUBMULTISET, *6-14*
UNDER_PATH, *6-23*
XML, *6-22*
CONNECT BY clause
of queries and subqueries, *19-39*
of SELECT, *9-4*, *19-39*
CONNECT clause
of SELECT and subqueries, *19-47*
CONNECT TO clause
of CREATE DATABASE LINK, *13-72*
CONNECT_BY_ISCYCLE pseudocolumn, *3-1*
CONNECT_BY_ISLEAF pseudocolumn, *3-2*
CONNECT_BY_ROOT operator, *4-6*
CONNECT_TIME parameter
of ALTER PROFILE, *11-75*
of ALTER RESOURCE COST, *11-79*
connection qualifier, *2-148*
CONSIDER FRESH clause
of ALTER MATERIALIZED VIEW, *11-33*
constant values
See literals
CONSTRAINT(S) session parameter, *11-96*
constraints,
adding to a table, *12-113*
altering, *12-47*
check, *8-3*
checking
at end of transaction, *8-3*
at start of transaction, *8-3*
at the end of each DML statement, *8-3*
column REF, *8-3*
deferrable, *8-3*, *19-125*
enforcing, *11-96*
defining, *8-3*, *15-17*
for a table, *15-17*
on a column, *15-17*
disabling, *15-17*
after table creation, *12-153*
cascading, *15-118*
during table creation, *15-48*
dropping, *12-47*, *12-114*, *18-7*
enabling, *15-17*, *15-118*
after table creation, *12-153*
during table creation, *15-48*
foreign key, *8-3*
modifying existing, *12-29*
on views
dropping, *12-208*, *18-16*
partitioning referential, *12-114*, *15-109*

constraints *(continued)*
primary key, *8-3*
attributes of index, *8-3*
enabling, *15-118*
referential integrity, *8-3*
renaming, *12-114*
restrictions, *8-7*
setting state for a transaction, *19-125*
storing rows in violation, *12-142*
table REF, *8-3*
unique
attributes of index, *8-3*
enabling, *15-118*
CONTAINER hint, *2-97*
CONTAINS condition, *6-2*
CONTAINS operator, *4-1*
context namespaces
accessible to instance, *13-44*
associating with package, *13-42*
initializing using OCI, *13-43*
initializing using the LDAP directory, *13-43*
removing from the database, *16-1*
contexts
creating namespaces for, *13-42*
granting system privileges for, *18-30*
control file clauses
of ALTER DATABASE, *10-48*
control files
allowing reuse, *13-47*, *13-57*
backing up, *10-75*
force logging mode, *13-50*
re-creating, *13-45*
standby, creating, *10-74*
CONTROLFILE REUSE clause
of CREATE DATABASE, *13-57*
conversion
functions, *7-7*
rules, string to date, *2-82*
CONVERT function, *7-81*
COPY clause
of CREATE PLUGGABLE DATABASE, *14-91*
CORR function, *7-83*
CORR_K function, *7-86*
CORR_S function, *7-86*
correlated subqueries, *9-16*
correlation functions
Kendall's tau-b, *7-84*
Pearson's, *7-83*
Spearman's rho, *7-84*
correlation names
in DELETE, *15-195*
in SELECT, *19-39*
COS function, *7-87*
COSH function, *7-87*
COUNT function, *7-88*
COVAR_POP function, *7-90*
COVAR_SAMP function, *7-92*
CPU_PER_CALL parameter
of ALTER PROFILE, *11-75*
of CREATE PROFILE, *14-99*
CPU_PER_SESSION parameter
of ALTER PROFILE, *11-75*
of ALTER RESOURCE COST, *11-79*
of CREATE PROFILE, *14-98*
CREATE ANALYTIC VIEW statement, *13-6*
CREATE ANY SQL PROFILE system privilege, *18-41*
CREATE ATTRIBUTE DIMENSION statement, *13-14*
CREATE AUDIT POLICY statement, *13-23*
CREATE CLUSTER statement, *13-33*
CREATE CONTEXT statement, *13-42*
CREATE CONTROLFILE statement, *13-44*
CREATE DATABASE LINK statement, *13-69*
CREATE DATABASE statement, *13-52*
CREATE DATAFILE clause
of ALTER DATABASE, *10-44*, *10-64*
CREATE DIMENSION statement, *13-75*
CREATE DIRECTORY statement, *13-81*
CREATE DISKGROUP statement, *13-84*
CREATE FLASHBACK ARCHIVE statement, *13-96*
CREATE FUNCTION statement, *13-99*
CREATE HIERARCHY statement, *13-101*
CREATE INDEX statement, *13-105*
CREATE INDEXTYPE statement, *13-142*
CREATE INMEMORY JOIN GROUP statement, *13-146*
CREATE JAVA statement, *13-148*
CREATE LIBRARY statement, *14-1*
CREATE LOCKDOWN PROFILE statement, *14-3*
CREATE MATERIALIZED VIEW LOG statement, *14-38*
CREATE MATERIALIZED VIEW statement, *14-4*
CREATE MATERIALIZED ZONEMAP statement, *14-49*
CREATE OPERATOR statement, *14-58*
CREATE OUTLINE statement, *14-62*
CREATE PACKAGE BODY statement, *14-67*
CREATE PACKAGE statement, *14-65*
locks, *B-5*
CREATE PFILE statement, *14-69*
CREATE PLUGGABLE DATABASE statement, *14-71*
CREATE PLUGGABLE DATABASE system privilege, *18-47*
CREATE PROCEDURE statement, *14-93*
locks, *B-5*
CREATE PROFILE statement, *14-95*
CREATE RESTORE POINT statement, *14-103*

CREATE ROLE statement, *14-106*
CREATE ROLLBACK SEGMENT statement, *14-110*
CREATE SCHEMA statement, *14-113*
CREATE SEQUENCE statement, *15-1*
CREATE SPFILE statement, *15-8*
CREATE STANDBY CONTROLFILE clause
    of ALTER DATABASE, *10-48*, *10-74*
CREATE SYNONYM statement, *15-12*
CREATE TABLE statement, *15-17*
CREATE TABLESPACE SET statement, *15-162*
CREATE TABLESPACE statement, *15-141*
CREATE TRIGGER statement, *15-164*
CREATE TYPE BODY statement, *15-168*
CREATE TYPE statement, *15-166*
CREATE USER statement, *15-169*
CREATE VIEW statement, *15-179*
cross joins, *19-71*
CUBE clause
    of SELECT statements, *19-77*
CUBE_TABLE function, *7-93*
cubes
    extracting data, *7-93*
CUME_DIST function, *7-95*
cumulative distributions, *7-95*
currency
    group separators, *2-69*
currency symbol
    ISO, *2-69*
    local, *2-69*
    union, *2-70*
CURRENT_DATE function, *7-97*
CURRENT_SCHEMA session parameter, *11-96*
CURRENT_TIMESTAMP function, *7-97*
CURRENT_USER clause
    of CREATE DATABASE LINK, *13-72*
CURRVAL pseudocolumn, *3-3*, *15-2*
CURSOR expressions, *5-21*
CURSOR_SHARING_EXACT hint, *2-98*
cursors
    cached, *18-17*
CV function, *7-98*
CYCLE parameter
    of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-85*
    of CREATE SEQUENCE, *15-5*

D

data
    aggregation
        composite columns of GROUP BY, *19-77*
        concatenated grouping sets of **GROUP BY**, *19-77*
        grouping sets, *19-77*
    analyzing a subset, *7-232*
    caching frequently used, *15-114*
    independence, *15-12*
    integrity checking on input, *2-14*
    locks on, *B-1*
    pivoting, *19-68*
    retrieving, *9-1*
    specifying as temporary, *15-49*
    undo
        preserving, *12-171*, *15-141*
    unpivoting, *19-69*
data cartridge functions, *7-21*
data conversion, *2-49*
    between character data types, *2-51*
    implicit
        disadvantages, *2-49*
    implicit versus explicit, *2-49*
    when performed implicitly, *2-49*, *2-52*
    when specified explicitly, *2-52*
data definition language
    locks, *B-4*
data definition language (DDL), *10-2*
    statements, *10-2*
        and implicit commit, *10-2*
        causing recompilation, *10-2*
        PL/SQL support, *10-2*
    statements requiring exclusive access, *10-2*
data dictionary
    adding comments to, *12-247*
    locks, *B-4*
data files
    bringing online, *10-40*
    changing size of, *10-65*
    creating new, *10-64*
    defining for a tablespace, *15-142*, *15-146*, *15-147*
    defining for the database, *13-55*
    designing media recovery, *10-56*
    dropping, *12-181*, *18-7*
    enabling autoextend, *8-32*
    end online backup of, *10-65*, *12-180*
    extending automatically, *8-32*
    online backup of, *12-179*
    online, updating information on, *12-10*
    putting online, *10-65*
    re-creating lost or damaged, *10-64*
    recover damaged, *10-56*
    recovering, *10-59*
    renaming, *10-64*
    resizing, *10-40*
    reusing, *8-32*
    size of, *8-32*

data files *(continued)*
specifying, *8-32*
for a tablespace, *15-148*
for database, *13-61*
system generated, *10-64*
taking offline, *10-40*, *10-65*
temporary
shrinking, *12-181*
data manipulation language (DML), *10-3*
allowing during indexing, *10-135*
operations
during index creation, *13-126*
during index rebuild, *12-148*
restricting, *12-14*
parallelizing, *15-17*
retrieving rows affected by, *15-202*, *18-71*, *19-146*
statements, *10-3*
PL/SQL support, *10-3*
data mining functions, *7-9*
data redaction
granting system privileges for, *18-30*
data types, *2-1*
"Any" types, *2-38*
ANSI-supported, *2-1*
BFILE, *2-30*
BLOB, *2-31*
built-in, *2-6*
CHAR, *2-10*
character, *2-9*
CLOB, *2-31*
comparison rules, *2-44*
converting to collection-typed values, *7-51*
converting to other data types, *7-51*
DATE, *2-21*
datetime, *2-20*
interval, *2-20*
INTERVAL DAY TO SECOND, *2-24*
INTERVAL YEAR TO MONTH, *2-24*
length semantics, *2-10*, *2-13*
LONG, *2-18*
LONG RAW, *2-28*
media types, *2-42*
NCHAR, *2-11*
NCLOB, *2-31*
NUMBER, *2-14*
numeric, *2-14*
NVARCHAR2, *2-13*
Oracle-supplied types, *2-38*
RAW, *2-28*
ROWID, *2-33*
SDO_TOPO_GEOMETRY, *2-42*
spatial types, *2-41*
TIMESTAMP, *2-22*
data types *(continued)*
TIMESTAMP WITH LOCAL TIME ZONE, *2-23*
TIMESTAMP WITH TIME ZONE, *2-22*
UROWID, *2-34*
user-defined, *2-36*
VARCHAR, *2-12*
VARCHAR2, *2-11*
XML types, *2-39*
database links, *9-18*
altering, *10-91*
closing, *11-89*
creating, *2-147*, *13-69*
creating synonyms with, *15-12*
current user, *13-72*
granting system privileges for, *18-30*
naming, *2-147*
public, *13-71*
dropping, *16-3*
referring to, *2-148*
removing from the database, *16-3*
shared, *13-71*
syntax, *2-147*
updating passwords, *10-91*
username and password, *2-148*
database objects
dropping, *18-14*
nonschema, *2-139*
schema, *2-138*
Database Smart Flash Cache, *8-51*
database triggers
See triggers
databases
accounts
creating, *15-169*
allowing changes to, *11-87*
allowing generation of redo logs, *10-40*
allowing reuse of control files, *13-57*
allowing unlimited resources to users, *14-98*
archive mode, specifying, *13-60*
beginning backup of, *10-63*
blocks
specifying size, *15-150*
cancel-based recovery
terminating, *10-60*
changing characteristics, *13-45*
changing global name, *10-84*
changing name, *13-45*, *13-47*
character set, specifying, *13-57*
committing to standby status, *10-79*
connect strings, *2-148*
controlling use, *10-88*
create script for, *10-40*
creating, *13-52*

databases *(continued)*
data files
modifying, *10-40*
specifying, *13-61*
default edition, setting, *10-82*
designing media recovery, *10-56*
dropping, *16-2*
ending backup of, *10-63*
erasing all data from, *13-52*
flashing back, *18-20*
granting system privileges for, *18-30*
in FLASHBACK mode, *10-40*
in FORCE LOGGING mode, *10-68*, *13-50*, *13-60*
instances of, *13-57*
limiting resources for users, *14-95*
log files
modifying, *10-40*
specifying, *13-58*
managed recovery, *10-43*
modifying, *10-40*
mounting, *10-54*, *13-52*
moving a subset to a different database, *12-29*
namespaces, *2-143*
naming, *10-53*
national character set, specifying, *13-58*
no-data-loss mode, *10-77*
online
adding log files, *10-70*
opening, *10-54*, *13-52*
prepare to re-create, *10-40*
preventing changes to, *10-40*
protection mode of, *10-77*
quiesced state, *12-14*
re-creating control file for, *13-45*
read-only, *10-54*
read/write, *10-54*
reconstructing damaged, *10-56*
recovering, *10-40*, *10-57*
recovery
allowing corrupt blocks, *10-59*
testing, *10-59*
using a storage snapshot, *10-58*
with backup control file, *10-40*
remote
accessing, *9-18*
authenticating users to, *13-69*
connecting to, *13-72*
inserting into, *18-61*
service name of, *13-69*
table locks on, *18-80*
restoring earlier version of, *10-40*, *12-171*, *15-141*

databases *(continued)*
restricting users to read-only transactions, *10-40*
resuming activity, *12-14*
returning to a past time, *18-20*
standby
adding log files, *10-70*
suspending activity, *12-14*
system user passwords, *13-56*
temp files
modifying, *10-40*
time zone
determining, *7-101*
setting, valid values for, *10-87*, *13-52*
DATAFILE clause
of CREATE DATABASE, *13-61*
DATAFILE clauses
of ALTER DATABASE, *10-44*, *10-65*
DATAFILE OFFLINE clause
of ALTER DATABASE, *10-40*
DATAFILE ONLINE clause
of ALTER DATABASE, *10-40*
DATAFILE RESIZE clause
of ALTER DATABASE, *10-40*
DATAOBJ_TO_MAT_PARTITION function, *7-100*
DATAOBJ_TO_PARTITION function, *7-100*
DATE columns
converting to datetime columns, *12-101*
DATE data type, *2-21*
julian, *2-21*
date format models, *2-71*, *2-73*
long, *2-73*
punctuation in, *2-72*
short, *2-74*
text in, *2-72*
date functions, *7-6*
dates
arithmetic, *2-24*
comparison rules, *2-44*
datetime arithmetic, *2-24*
boundary cases, *11-96*
calculating daylight saving time, *2-26*
datetime columns
creating from DATE columns, *12-101*
datetime data types, *2-20*
daylight saving time, *2-26*
datetime expressions, *5-23*
datetime field
extracting from a datetime or interval value, *7-114*
datetime format elements, *2-72*
and Globalization Support, *2-77*
capitalization, *2-72*
ISO standard, *2-77*
RR, *2-78*

datetime format elements *(continued)*
 suffixes, *2-79*
datetime functions, *7-6*
datetime literals, *2-60*
DAY datetime format element, *2-77*
daylight saving time, *2-26*
 boundary cases, *2-26*
 going into or coming out of effect, *2-26*
DB2 data types, *2-34*
 restrictions on, *2-35*
DBA_2PC_PENDING data dictionary view, *11-89*
DBA_COL_COMMENTS data dictionary view, *12-248*
DBA_INDEXTYPE_COMMENTS data dictionary view, *12-249*
DBA_MVIEW_COMMENTS data dictionary view, *12-249*
DBA_OPERATOR_COMMENTS data dictionary view, *12-249*
DBA_ROLLBACK_SEGS data dictionary view, *17-17*
DBA_TAB_COMMENTS data dictionary view, *12-248*
DBMS_ROWID package
 and extended rowids, *2-34*
DBTIMEZONE function, *7-101*
DDL
 See data definition language (DDL)
DEALLOCATE UNUSED clause
 of ALTER CLUSTER, *10-36*, *10-37*
 of ALTER INDEX, *10-136*
 of ALTER TABLE, *12-86*
debugging
 granting system privileges for, *18-30*
decimal characters
 specifying, *2-69*
DECODE function, *7-102*
decoding functions, *7-11*
DECOMPOSE function, *7-103*
DEFAULT clause
 of ALTER TABLE, *12-95*
 of CREATE TABLE, *15-17*, *15-55*
DEFAULT COST clause
 of ASSOCIATE STATISTICS, *12-218*, *12-219*
default index, suppressing, *14-23*
DEFAULT profile
 assigning to users, *17-15*
DEFAULT ROLE clause
 of ALTER USER, *12-200*
DEFAULT SELECTIVITY clause
 of ASSOCIATE STATISTICS, *12-218*, *12-219*
default tablespace, *13-63*
DEFAULT TABLESPACE clause
 of ALTER DATABASE, *10-83*
 of ALTER PLUGGABLE DATABASE, *11-61*
DEFAULT TABLESPACE clause *(continued)*
 of ALTER USER, *12-199*
 of ALTER USER. *See* CREATE USER, *12-197*
 of CREATE USER, *15-174*
default tablespaces
 specifying for a user, *12-199*
DEFAULT TEMPORARY TABLESPACE clause
 of ALTER DATABASE, *10-83*
 of ALTER PLUGGABLE DATABASE, *11-61*
 of CREATE DATABASE, *13-54*
DEFERRABLE clause
 of constraints, *8-3*
deferrable constraints, *19-125*
DEFERRED clause
 of SET CONSTRAINTS, *19-126*
definer's rights views, *15-188*
DELETE statement, *15-195*
 error logging, *15-195*
DELETE STATISTICS clause
 of ANALYZE, *12-216*
DELETEXML function, *7-105*
DENSE_RANK function, *7-106*
DEPTH function, *7-108*
DEREF function, *7-108*
DESC clause
 of CREATE INDEX, *13-122*
dictionaries
 granting system privileges for, *18-42*
dimensional objects
 extracting data, *7-93*
dimensions
 attributes
  adding, *10-95*
  changing, *10-92*
  defining, *13-75*
  dropping, *10-95*
 compiling invalidated, *10-95*
 creating, *13-75*
 defining levels, *13-76*
 examples, *13-75*
 extracting data, *7-93*
 granting system privileges for, *18-30*
 hierarchies
  adding, *10-95*
  changing, *10-92*
  defining, *13-75*
  dropping, *10-95*
 levels
  adding, *10-95*
  defining, *13-75*
  dropping, *10-95*
  parent-child hierarchy, *13-77*
 removing from the database, *16-4*
direct-path INSERT, *2-94*, *2-95*, *18-61*

directories
    See directory objects
directory objects,
    as aliases for operating system directories, *13-81*
    auditing, *12-228*, *12-229*
    creating, *13-81*
    granting system privileges for, *18-30*
    redefining, *13-82*
    removing from the database, *16-5*
DISABLE ALL TRIGGERS clause
    of ALTER TABLE, *12-154*
DISABLE clause
    of ALTER INDEX, *10-150*
    of CREATE TABLE, *15-17*
DISABLE DISTRIBUTED RECOVERY clause
    of ALTER SYSTEM, *12-10*
DISABLE PARALLEL DML clause
    of ALTER SESSION, *11-90*
DISABLE QUERY REWRITE clause
    of ALTER MATERIALIZED VIEW, *11-31*
    of CREATE MATERIALIZED VIEW, *14-31*
DISABLE RESTRICTED SESSION clause
    of ALTER SYSTEM, *12-17*
DISABLE RESUMABLE clause
    of ALTER SESSION, *11-91*
DISABLE ROW MOVEMENT clause
    of ALTER TABLE, *12-29*
    of CREATE TABLE, *15-17*, *15-33*
DISABLE STORAGE IN ROW clause
    of ALTER TABLE, *12-98*
    of CREATE TABLE, *15-86*
DISABLE TABLE LOCK clause
    of ALTER TABLE, *12-153*
DISABLE_PARALLEL_DML hint, *2-98*
DISASSOCIATE STATISTICS statement, *15-204*
DISCONNECT SESSION clause
    of ALTER SYSTEM, *12-12*
disk group files
    changing permission settings, *10-122*
    Intelligent Data Placement, *10-116*
    setting owner or user group, *10-122*
disk groups
    altering, *10-96*
    creating, *13-84*
        a tablespace in, *15-148*
        failure groups, *10-107*, *13-87*
        files in, *8-32*
    dropping, *16-6*
    managing Oracle ADVM volumes, *10-118*
    rebalancing, *10-96*
    setting attributes, *10-96*, *13-88*
    specifying files in, *8-32*
    specifying files in control files, *13-49*
disks
    bringing online, *10-110*
    QUORUM, *13-86*
    regions, *10-116*
    REGULAR, *13-86*
    replacing, *10-109*
    taking offline, *10-111*
dispatcher processes
    creating additional, *12-25*
    terminating, *12-25*
DISTINCT clause
    of SELECT, *19-59*
distinct queries, *19-59*
distributed queries, *9-18*
    restrictions on, *9-18*
distribution
    hints for, *2-127*
DML
    See data manipulation language (DML)
domain indexes, *13-105*, *13-142*
    and LONG columns, *12-101*
    associating statistics types with, *12-218*
    creating, prerequisites, *13-131*
    determining user-defined CPU and I/O costs, *18-16*
    disassociating statistics types from, *15-204*, *16-12*
    example, *F-1*
    invoking drop routines for, *18-1*
    local partitioned, *13-132*
    modifying, *10-148*
    parallelizing creation of, *13-132*
    rebuilding, *10-135*
    removing from the database, *16-12*
    system managed, *13-145*
domain_index_clause
    of CREATE INDEX, *13-110*
DOWNGRADE clause
    of ALTER DATABASE, *10-55*
DROP ANALYTIC VIEW statement, *15-206*
DROP ANY SQL PROFILE system privilege, *18-41*
DROP ATTRIBUTE DIMENSION statement, *15-207*
DROP AUDIT POLICY statement, *15-207*
DROP clause
    of ALTER DIMENSION, *10-95*
    of ALTER INDEXTYPE, *10-160*
DROP CLUSTER statement, *15-209*
DROP COLUMN clause
    of ALTER TABLE, *12-105*
DROP constraint clause
    of ALTER VIEW, *12-208*
DROP CONSTRAINT clause
    of ALTER TABLE, *12-114*

DROP CONTEXT statement, *16-1*
DROP DATABASE LINK statement, *16-3*
DROP DATABASE statement, *16-2*
DROP DIMENSION statement, *16-4*
DROP DIRECTORY statement, *16-5*
DROP DISKGROUP statement, *16-6*
DROP FLASHBACK ARCHIVE statement, *16-9*
DROP FUNCTION statement, *16-10*
DROP HIERARCHY statement, *16-11*
DROP INDEX statement, *16-12*
DROP INDEXTYPE statement, *16-14*
DROP INMEMORY JOIN GROUP statement, *16-15*
DROP JAVA statement, *16-16*
DROP LIBRARY statement, *17-1*
DROP LOCKDOWN PROFILE statement, *17-2*
DROP LOGFILE clause
  of ALTER DATABASE, *10-46*, *10-72*
DROP LOGFILE MEMBER clause
  of ALTER DATABASE, *10-46*, *10-72*
DROP MATERIALIZED VIEW LOG statement, *17-5*
DROP MATERIALIZED VIEW statement, *17-3*
DROP MATERIALIZED ZONEMAP statement, *17-7*
DROP OPERATOR statement, *17-8*
DROP OUTLINE statement, *17-9*
DROP PACKAGE BODY statement, *17-11*
DROP PACKAGE statement, *17-10*
DROP PARTITION clause
  of ALTER INDEX, *10-135*
  of ALTER TABLE, *12-130*
DROP PLUGGABLE DATABASE statement, *17-11*
DROP PRIMARY constraint clause
  of ALTER TABLE, *12-114*
DROP PROCEDURE statement, *17-13*
DROP PROFILE statement, *17-14*
DROP RESTORE POINT statement, *17-15*
DROP ROLE statement, *17-16*
DROP ROLLBACK SEGMENT statement, *17-17*
DROP SEQUENCE statement, *17-18*
DROP SUPPLEMENTAL LOG DATA clause
  of ALTER DATABASE, *10-74*
DROP SUPPLEMENTAL LOG GROUP clause
  of ALTER TABLE, *12-85*
DROP SYNONYM statement, *17-19*
DROP TABLE statement, *18-1*
DROP TABLESPACE SET statement, *18-8*
DROP TABLESPACE statement, *18-5*
DROP TRIGGER statement, *18-9*
DROP TYPE BODY statement, *18-12*
DROP TYPE statement, *18-10*
DROP UNIQUE constraint clause
  of ALTER TABLE, *12-114*
DROP USER statement, *18-13*
DROP VALUES clause
  of ALTER TABLE ... MODIFY PARTITION, *12-123*, *12-124*
DROP VIEW statement, *18-15*
DUAL dummy table, *2-141*, *9-18*
DUMP function, *7-109*
DY datetime format element, *2-77*
DYNAMIC_SAMPLING hint, *2-99*

E

editioning views, *15-182*
editions
  comments on, *12-248*
  creating, *13-93*
  dropping, *16-8*
  granting system privileges for, *18-30*
  setting default for a PDB, *11-61*
  setting default for database, *10-82*
  setting for a session, *11-93*
embedded SQL, *10-4*
  precompiler support, *10-4*
EMPTY_BLOB function, *7-111*
EMPTY_CLOB function, *7-111*
ENABLE ALL TRIGGERS clause
  of ALTER TABLE, *12-153*
ENABLE clause
  of ALTER INDEX, *10-150*
  of ALTER TRIGGER, *12-193*
  of CREATE TABLE, *15-17*
ENABLE DISTRIBUTED RECOVERY clause
  of ALTER SYSTEM, *12-10*
ENABLE NOVALIDATE constraint state, *8-3*
ENABLE PARALLEL DML clause
  of ALTER SESSION, *11-90*
ENABLE QUERY REWRITE clause
  of ALTER MATERIALIZED VIEW, *11-31*
  of CREATE MATERIALIZED VIEW, *14-31*
ENABLE RESTRICTED SESSION clause
  of ALTER SYSTEM, *12-17*
ENABLE RESUMABLE clause
  of ALTER SESSION, *11-91*
ENABLE ROW MOVEMENT clause
  of ALTER TABLE, *12-29*
  of CREATE TABLE, *15-17*, *15-33*
ENABLE STORAGE IN ROW clause
  of ALTER TABLE, *12-98*
  of CREATE TABLE, *15-86*
ENABLE TABLE LOCK clause
  of ALTER TABLE, *12-153*
ENABLE VALIDATE constraint state, *8-3*
ENABLE_PARALLEL_DML hint, *2-100*
encoding functions, *7-11*

encryption, *15-57*
 of tablespaces, *8-51*
encryption keys
 generating, *12-18*
 managing, *10-5*
END BACKUP clause
 of ALTER DATABASE, *10-63*
 of ALTER DATABASE ... DATAFILE, *10-40*
 of ALTER TABLESPACE, *12-180*
enterprise users
 allowing connection as database users, *12-195*
environment functions, *7-11*
equality test, *6-4*
equijoins, *9-12*
 defining for a dimension, *13-75*
equivalency tests, *6-33*
error logging
 of DELETE operations, *15-195*
 of INSERT operations, *18-74*
 of MERGE operations, *19-1*
ERROR_ON_OVERLAP_TIME session parameter, *11-96*
EXCEPTIONS INTO clause
 of ALTER TABLE, *12-142*
EXCHANGE PARTITION clause
 of ALTER TABLE, *12-29*, *12-69*
EXCHANGE SUBPARTITION clause
 of ALTER TABLE, *12-29*, *12-69*
exchanging partitions
 restrictions on, *12-143*
EXCLUDING NEW VALUES clause
 of ALTER MATERIALIZED VIEW LOG, *11-42*
 of CREATE MATERIALIZED VIEW LOG, *14-46*
EXCLUSIVE lock mode, *18-83*
exclusive locks
 row locks (TX), *B-1*
 table locks (TM), *B-1*
EXECUTE object privilege
 on a directory, *18-54*
execution plans
 determining, *18-16*
 dropping outlines for, *17-9*
 saving, *14-62*
EXISTS condition, *6-22*, *6-33*
EXISTSNODE function, *7-112*
EXP function, *7-113*
EXPLAIN PLAN statement, *18-16*
explicit data conversion, *2-49*, *2-52*
expressions
 analytic view measure, *5-6*
 analytic view simple, *5-16*
 calculated measure, *5-4*, *5-16*
 CASE, *5-18*
expressions *(continued)*
 changing declared type of, *7-385*
 column, *5-20*
 comparing, *7-102*
 compound, *5-3*
 computing with the DUAL table, *9-18*
 CURSOR, *5-21*
 datetime, *5-23*
 in SQL syntax, *5-1*
 interval, *5-25*
 lists of, *5-33*
 model, *5-28*
 object access, *5-30*
 placeholder, *5-30*
 scalar subqueries as, *5-31*
 simple, *5-3*
 single row function, *5-16*
 type constructor, *5-31*
extended rowids
 base 64, *2-34*
 not directly available, *2-34*
extensible indexing
 example, *F-1*
EXTENT MANAGEMENT clause
 of CREATE DATABASE, *13-54*
 of CREATE TABLESPACE, *15-141*, *15-146*
EXTENT MANAGEMENT DICTIONARY clause
 of CREATE TABLESPACE, *15-154*
EXTENT MANAGEMENT LOCAL clause
 of CREATE DATABASE, *13-61*
extents
 allocating for partitions, *12-85*
 allocating for subpartitions, *12-85*
 allocating for tables, *12-85*
 restricting access by instances, *10-135*
 specifying maximum number for an object, *8-56*
 specifying number allocated upon object creation, *8-55*
 specifying the first for an object, *8-54*
 specifying the percentage of size increase, *8-55*
 specifying the second for an object, *8-54*
external functions, *13-100*, *14-93*
external LOBs, *2-29*
external procedures, *14-93*
external tables, *15-77*
 access drivers, *15-81*
 altering, *12-29*
 creating, *15-17*
 ORACLE_DATAPUMP access driver, *15-81*
 ORACLE_HDFS access driver, *15-81*
 ORACLE_HIVE access driver, *15-81*
 ORACLE_LOADER access driver, *15-81*
 restrictions on, *15-81*

external users, *14-108*, *15-173*
EXTRACT (datetime) function, *7-114*
EXTRACT (XML) function, *7-116*
EXTRACTVALUE function, *7-117*

F

FACT hint, *2-100*
FAILED_LOGIN_ATTEMPTS parameter
 of ALTER PROFILE, *11-76*
 of CREATE PROFILE, *14-100*
failure groups
 creating for a disk group, *10-107*, *13-87*
fast refresh, *14-38*
FEATURE_DETAILS function, *7-121*
FEATURE_ID function, *7-124*
FEATURE_SET function, *7-126*
FEATURE_VALUE function, *7-129*
files
 specifying as a redo log file group, *8-32*
 specifying as data files, *8-32*
 specifying as temp files, *8-32*
FIPS
 compliance, *C-32*
 flagging, *11-96*
FIRST function, *7-132*
FIRST_ROWS(n) hint, *2-100*
FIRST_VALUE function, *7-134*
FLAGGER session parameter, *11-96*
flash cache, *8-51*
FLASH_CACHE parameter
 of STORAGE clause, *8-51*
FLASHBACK ARCHIVE object privilege, *18-54*
flashback data archives
 creating, *13-96*
 dropping, *16-9*
 modifying, *10-130*
 privileges for, *18-30*
 specifying for a table, *12-88*, *15-119*
FLASHBACK DATABASE statement, *18-20*
flashback queries, *19-39*
 pseudocolumns for, *3-6*
 using with inserts, *18-61*, *19-138*
FLASHBACK TABLE statement, *18-23*
floating-point conditions, *6-8*
floating-point numbers, *2-16*
 converting to, *7-355*, *7-357*
 handling NaN, *7-207*
FLOOR function, *7-136*
FLUSH BUFFER_CACHE clause
 of ALTER SYSTEM, *12-11*
FLUSH GLOBAL CONTEXT clause
 of ALTER SYSTEM, *12-11*
FLUSH REDO clause
 of ALTER SYSTEM, *12-11*
FLUSH SHARED_POOL clause
 of ALTER SYSTEM, *12-10*
FM format model modifier, *2-79*
FOR clause
 of CREATE INDEXTYPE, *13-144*
 of EXPLAIN PLAN, *18-18*, *18-23*
FOR UPDATE clause
 of SELECT, *19-39*, *19-51*
FORCE clause
 of COMMIT, *13-5*
 of CREATE VIEW, *15-182*
 of DISASSOCIATE STATISTICS, *15-205*
 of DROP INDEX, *16-13*
 of DROP INDEXTYPE, *16-15*
 of DROP OPERATOR, *17-9*
 of DROP TYPE, *18-11*
 of REVOKE, *19-30*
 of ROLLBACK, *18-23*, *19-37*
force full database caching, *10-85*
FORCE LOGGING clause
 of ALTER DATABASE, *10-68*
 of ALTER TABLESPACE, *12-182*
 of CREATE CONTROLFILE, *13-50*
 of CREATE DATABASE, *13-60*
 of CREATE TABLESPACE, *15-151*
FORCE PARALLEL DML clause
 of ALTER SESSION, *11-90*
foreign key constraints, *8-3*
foreign tables
 rowids of, *2-34*
format models, *2-67*
 changing the return format, *2-81*
 date, *2-71*
  changing, *2-71*
  default format, *2-71*
  format elements, *2-72*
  maximum length, *2-71*
 modifiers, *2-79*
 number, *2-68*
 number, elements of, *2-69*
 specifying, *2-80*
 XML, *2-83*
formats
 for dates and numbers. *See* format models, *2-67*
 of return values from the database, *2-67*
 of values stored in the database, *2-67*
free lists
 specifying for a table, partition, cluster, or index, *8-57*
 specifying for LOBs, *15-87*
FREELIST GROUPS parameter
 of STORAGE clause, *8-57*
FREELISTS parameter
 of STORAGE clause, *8-57*

FREEPOOLS parameter
of LOB storage, 15-87
FRESH_MV hint, 2-101
FROM clause
of CREATE PLUGGABLE DATABASE, 14-84
of queries, 9-13
FROM COLUMNS clause
of DISASSOCIATE STATISTICS, 15-205
FROM FUNCTIONS clause
of DISASSOCIATE STATISTICS, 15-205
FROM INDEXES clause
of DISASSOCIATE STATISTICS, 15-205
FROM INDEXTYPES clause
of DISASSOCIATE STATISTICS, 15-205
FROM PACKAGES clause
of DISASSOCIATE STATISTICS, 15-205
FROM TYPES clause
of DISASSOCIATE STATISTICS, 15-205
FROM_TZ function, 7-137
FULL hint, 2-101
full indexes, 13-105
full outer joins, 19-39
function expressions
built-in, 5-24
user-defined, 5-24
function-based indexes, 13-105
creating, 13-105
disabling, 10-135, 10-150
enabling, 10-135, 10-150
refreshing, 10-81
functions, 7-431
3GL,calling, 14-1
associating statistics types with, 12-218
avoiding run-time compilation, 10-133
built_in
as expressions, 5-24
calling, 12-243
changing the declaration of, 13-101
changing the definition of, 13-101
defining an index on, 13-105
disassociating statistics types from, 15-204
executing, 12-243
external, 13-100, 14-93
inverse distribution, 7-237, 7-239
issuing COMMIT or ROLLBACK statements, 11-89
linear regression, 7-292
naming rules, 2-144
OLAP, 7-21
re-creating, 13-101, 13-149
recompiling invalid, 10-133
removing from the database, 16-10
statistics, assigning default cost, 12-219
statistics, defining default selectivity, 12-219
stored, 13-99
functions *(continued)*
storing return value of, 12-243
synonyms for, 15-12
user-defined, 7-431
as expressions, 5-24
XML, 7-9
See also SQL functions
FUNCTIONS clause
of ASSOCIATE STATISTICS, 12-218, 12-220
of DISASSOCIATE STATISTICS, 15-204
FX format model modifier, 2-79

G

GATHER_OPTIMIZER_STATISTICS hint, 2-102
general comparison functions, 7-7
general recovery clause
of ALTER DATABASE, 10-42, 10-56
geoimaging, 2-41
global indexes
See indexes, globally partitioned
GLOBAL parameter
of CREATE SEQUENCE, 15-7
GLOBAL PARTITION BY HASH clause
of CREATE INDEX, 13-127
GLOBAL PARTITION BY RANGE clause
of CREATE INDEX, 13-111, 13-127
global sequences, 15-7
GLOBAL TEMPORARY clause
of CREATE TABLE, 15-49
global users, 14-108, 15-173
GLOBAL_TOPIC_ENABLED system parameter, 12-24
globally partitioned indexes, 13-105, 13-127
GRANT CONNECT THROUGH clause
of ALTER USER, 12-195, 12-197
GRANT statement
locks, B-5
GRAPHIC data type
DB2, 2-35
SQL/DS, 2-35
greater than or equal to tests, 6-4
greater than tests, 6-4
GREATEST function, 7-138
GROUP BY clause
CUBE extension, 19-77
identifying duplicate groupings, 7-139
of SELECT and subqueries, 19-39, 19-47
ROLLUP extension of, 19-76
group comparison conditions, 6-7
group separator
specifying, 2-69
GROUP_ID function, 7-139
GROUPING, 2-102
GROUPING function, 7-140

GROUPING Hint, 2-102
grouping sets, 19-77
GROUPING SETS clause
of SELECT and subqueries, 19-77
GROUPING_ID function, 7-141
groupings
filtering out duplicate, 7-139
GUARD ALL clause
of ALTER DATABASE, 10-87
GUARD clause
of ALTER DATABASE, 10-40
overriding, 11-87
GUARD NONE clause
of ALTER DATABASE, 10-88
GUARD STANDBY clause
of ALTER DATABASE, 10-87

H

hash clusters
creating, 13-38
range-partitioned, 13-40
single-table, creating, 13-38
specifying hash function for, 13-33
HASH hint, 2-103
HASH IS clause
of CREATE CLUSTER, 13-33
hash partitioning clause
of CREATE TABLE, 15-17, 15-47
hash partitions
adding, 12-29
coalescing, 12-122
HASHKEYS clause
of CREATE CLUSTER, 13-38
HAVING condition
of GROUP BY clause, 19-78
heap-organized tables
creating, 15-17
hexadecimal value
returning, 2-70
HEXTORAW function, 7-142
hierarchical functions, 7-8
hierarchical queries, 9-2, 19-39
child rows, 3-2, 9-4
illustrated, 3-3
leaf rows, 3-2
operators in, 4-5
CONNECT_BY_ROOT, 4-6
PRIOR, 4-5
ordering, 19-83
parent rows, 3-2, 9-4
pseudocolumns in, 3-1
CONNECT_BY_ISCYCLE, 3-1
CONNECT_BY_ISLEAF, 3-2
LEVEL, 3-2
hierarchical queries *(continued)*
retrieving root and node values, 7-334
hierarchical query clause
of SELECT and subqueries, 19-47
hierarchies
adding to a dimension, 10-95
altering, 10-134
creating, 13-101
dropping, 16-11
dropping from a dimension, 10-95
granting system privileges for, 18-30
of dimensions, defining, 13-75
HIERARCHY clause
of CREATE DIMENSION, 13-75, 13-76
high water mark
of clusters, 10-36
of indexes, 10-135
of tables, 12-86, 12-211
hints, 9-2
ALL_ROWS, 2-94
APPEND, 2-94
APPEND_VALUES, 2-95
CACHE, 2-95
CLUSTER, 2-96
CLUSTERING, 2-97
CONTAINER, 2-97
CURSOR_SHARING_EXACT, 2-98
DISABLE_PARALLEL_DML, 2-98
DYNAMIC_SAMPLING, 2-99
ENABLE_PARALLEL_DML, 2-100
FACT, 2-100
FIRST_ROWS(n), 2-100
FRESH_MV, 2-101
FULL, 2-101
GATHER_OPTIMIZER_STATISTICS, 2-102
HASH, 2-103
in SQL statements, 2-87
INDEX, 2-104
INDEX_ASC, 2-104
INDEX_COMBINE, 2-105
INDEX_DESC, 2-105
INDEX_FFS, 2-106
INDEX_JOIN, 2-106
INDEX_SS, 2-106
INDEX_SS_ASC, 2-107
INDEX_SS_DESC, 2-107
INMEMORY, 2-108
INMEMORY_PRUNING, 2-108
LEADING, 2-108
location syntax, 2-87
MERGE, 2-109
MODEL_MIN_ANALYSIS, 2-109
MONITOR, 2-110
NO_CLUSTERING, 2-111
NO_EXPAND, 2-111

hints *(continued)*
NO_FACT, *2-112*
NO_GATHER_OPTIMIZER_STATISTICS, *2-112*
NO_INDEX, *2-112*
NO_INDEX_FFS, *2-113*
NO_INDEX_SS, *2-113*
NO_INMEMORY, *2-114*
NO_INMEMORY_PRUNING, *2-114*
NO_MERGE, *2-114*
NO_MONITOR, *2-115*
NO_PARALLEL, *2-115*
NO_PARALLEL_INDEX, *2-116*
NO_PQ_CONCURRENT_UNION, *2-116*
NO_PQ_SKEW, *2-117*
NO_PUSH_PRED, *2-117*
NO_PUSH_SUBQ, *2-117*
NO_PX_JOIN_FILTER, *2-118*
NO_QUERY_TRANSFORMATION, *2-118*
NO_RESULT_CACHE, *2-118*
NO_REWRITE, *2-118*
NO_STAR_TRANSFORMATION, *2-119*
NO_STATEMENT_QUEUING, *2-119*
NO_UNNEST, *2-120*
NO_USE_BAND, *2-120*
NO_USE_CUBE, *2-120*
NO_USE_HASH, *2-120*
NO_USE_MERGE, *2-121*
NO_USE_NL, *2-121*
NO_XML_QUERY_REWRITE, *2-121*
NO_XMLINDEX_REWRITE, *2-122*
NO_ZONEMAP, *2-122*
NOCACHE, *2-111*
NOPARALLEL, *2-115*
NOPARALLEL_INDEX, *2-116*
NOREWRITE, *2-118*
OPT_PARAM, *2-123*
ORDERED, *2-123*
PARALLEL, *2-124*
PARALLEL_INDEX, *2-126*
passing to the optimizer, *19-138*
PQ_CONCURRENT_UNION, *2-127*
PQ_DISTRIBUTE, *2-127*
PQ_FILTER, *2-130*
PQ_SKEW, *2-130*
PUSH_PRED, *2-130*
PUSH_SUBQ, *2-131*
PX_JOIN_FILTER, *2-131*
QB_NAME, *2-131*
REWRITE, *2-133*
specifying a query block, *2-87*
STAR_TRANSFORMATION, *2-133*
STATEMENT_QUEUING, *2-134*
syntax, *2-91*
UNNEST, *2-134*

hints *(continued)*
USE_BAND, *2-135*
USE_CONCAT, *2-135*
USE_CUBE, *2-136*
USE_HASH, *2-136*
USE_MERGE, *2-136*
USE_NL, *2-137*
USE_NL_WITH_INDEX, *2-137*
histograms
creating equiwidth, *7-403*
Hybrid Columnar Compression, *15-70*

I

IDENTIFIED BY clause
of ALTER ROLE. *See* CREATE ROLE, *11-80*
of CREATE DATABASE LINK, *13-73*
IDENTIFIED EXTERNALLY clause
of ALTER ROLE. *See* CREATE ROLE, *11-80*, *14-108*
of ALTER USER. *See* CREATE USER, *15-173*
of CREATE ROLE, *14-108*
of CREATE USER, *15-173*
IDENTIFIED GLOBALLY clause
of ALTER ROLE. *See* CREATE ROLE, *11-80*
of CREATE ROLE, *14-108*
of CREATE USER, *15-173*
identifier functions, *7-11*
identity column, *15-56*
IDLE_TIME parameter
of ALTER PROFILE, *11-75*
IEEE754
floating-point arithmetic, *2-17*
Oracle conformance with, *2-17*
IGNORE_ROW_ON_DUPKEY_INDEX hint, *2-103*
IMMEDIATE clause
of SET CONSTRAINTS, *19-126*
implicit data conversion, *2-49*, *2-52*
IN conditions, *6-33*
in-doubt transactions
forcing, *13-5*
forcing commit of, *13-5*
forcing rollback, *18-23*, *19-37*
rolling back, *19-36*
INCLUDING CONTENTS clause
of DROP TABLESPACE, *18-7*
INCLUDING DATAFILES clause
of ALTER DATABASE TEMPFILE DROP clause, *10-67*
INCLUDING NEW VALUES clause
of ALTER MATERIALIZED VIEW LOG, *11-42*
of CREATE MATERIALIZED VIEW LOG, *14-46*

INCLUDING TABLES clause
  of DROP CLUSTER, *15-210*
incomplete object types, *15-166*
  creating, *15-166*
INCREMENT BY clause
  of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-86*
INCREMENT BY parameter
  of CREATE SEQUENCE, *15-4*
incremental
  and block change tracking, *10-85*
INDEX clause
  of ANALYZE, *12-212*
  of CREATE CLUSTER, *13-38*
INDEX hint, *2-104*
index keys
  compression, *10-138*
index partitions
  creating subpartitions, *13-114*
index subpartitions, *13-114*
INDEX_ASC hint, *2-104*
INDEX_COMBINE hint, *2-105*
INDEX_DESC hint, *2-105*
INDEX_FFS hint, *2-106*
INDEX_JOIN hint, *2-106*
INDEX_SS hint, *2-106*
INDEX_SS_ASC hint, *2-107*
INDEX_SS_DESC hint, *2-107*
index-organized tables
  bitmap indexes on, creating, *15-79*
  creating, *15-17*
  mapping tables, *12-149*
    creating, *15-79*
    moving, *12-125*
  merging contents of index blocks, *12-93*
  modifying, *12-29*, *12-93*
  moving, *12-149*
  overflow segments
    specifying storage, *12-92*, *15-104*
  partitioned, updating secondary indexes, *10-154*
  PCT_ACCESS_DIRECT statistics, *12-211*
  primary key indexes
    coalescing, *12-91*
  rebuilding, *12-29*
  rowids of, *2-34*
  secondary indexes, updating, *10-152*
indexed clusters
  creating, *13-38*
indexes, *10-135*
  advanced index compression of, *13-123*
  advanced index compression, enabling, *10-148*
  allocating new extents for, *10-135*
  application-specific, *13-142*

indexes *(continued)*
  ascending, *13-122*
  B-tree, *13-105*
  based on indextypes, *13-105*
  bitmap, *13-115*
  bitmap join, *13-105*
  changing attributes, *10-135*
  changing parallelism of, *10-135*
  collecting statistics on, *12-212*
  creating, *13-105*
  creating as usable or unusable, *13-134*
  creating on a cluster, *13-107*
  creating on a table, *13-107*
  deallocating unused space from, *10-135*
  descending, *13-122*
    and query rewrite, *13-122*
    as function-based indexes, *13-122*
  direct-path inserts, logging, *10-135*
  domain, *13-105*, *13-142*
  domain, example, *F-1*
  dropping index partitions, *16-12*
  examples, *13-105*
  full, *13-105*
  full fast scans, *2-106*
  function-based, *13-105*
    creating, *13-105*
  global partitioned, creating, *13-111*
  globally partitioned, *13-105*, *13-127*
    updating, *12-29*
  granting system privileges for, *18-30*
  invisible to the optimizer, *10-151*, *13-125*
  join, bitmap, *13-105*
  local domain, *13-132*
  locally partitioned, *13-105*
  logging rebuild operations, *10-135*
  marking as USABLE or UNUSABLE, *10-151*
  merging block contents, *10-135*
  merging contents of index blocks, *10-151*
  merging contents of index partition blocks, *10-154*
  modifying attributes, *10-135*
  moving, *10-135*
  on clusters, *13-105*
  on composite-partitioned tables, *13-105*
  on composite-partitioned tables, creating, *13-114*
  on hash-partitioned tables, *13-105*
    creating, *13-114*
  on index-organized tables, *13-105*
  on list-partitioned tables
    creating, *13-113*
  on nested table storage tables, *13-105*
  on partitioned tables, *13-105*
  on range-partitioned tables, *13-105*
  on range-partitioned tables, creating, *13-113*

indexes *(continued)*
on scalar typed object attributes, *13-105*
on table columns, *13-105*
on XMLType tables, *13-136*
online, *13-126*
parallelizing creation of, *13-127*
partial, *13-105*
partitioned, *2-149*, *13-105*
user-defined, *13-127*
partitioning, *13-127*
partitions, *13-127*
adding hash, *10-135*
adding new, *10-135*
changing default attributes, *10-135*
changing physical attributes, *10-135*
changing storage characteristics, *10-135*
coalescing hash partitions, *10-135*
deallocating unused space from, *10-135*
dropping, *10-135*
marking UNUSABLE, *10-135*, *12-144*
modifying the real characteristics, *10-135*
preventing use of, *10-151*
re-creating, *10-135*
rebuilding, *10-135*
rebuilding unusable, *12-144*
removing, *10-135*
renaming, *10-135*
specifying tablespace for, *10-135*, *10-147*
splitting, *10-135*
prefix compression of, *13-123*
prefix compression, enabling, *10-148*
preventing use of, *10-151*
purging from the recycle bin, *19-20*
re-creating, *10-135*
rebuilding, *10-135*
removing from the database, *16-12*
renaming, *10-135*, *10-151*
reverse, *10-135*, *10-147*, *10-148*, *13-125*
specifying tablespace for, *10-135*, *10-147*
statistics on usage, *10-152*
subpartitions
allocating extents for, *10-135*
changing default attributes, *10-135*
changing physical attributes, *10-135*
changing storage characteristics, *10-135*
deallocating unused space from, *10-135*
marking UNUSABLE, *10-135*
modifying, *10-135*
moving, *10-135*
preventing use of, *10-151*
re-creating, *10-135*
rebuilding, *10-135*
renaming, *10-135*

indexes *(continued)*
subpartitions *(continued)*
specifying tablespace for, *10-135*, *10-147*
tablespace containing, *13-123*
unique, *13-115*
unsorted, *13-125*
used to enforce constraints, *12-115*, *15-17*
validating structure, *12-214*
INDEXES clause
of ASSOCIATE STATISTICS, *12-218*, *12-220*
of DISASSOCIATE STATISTICS, *15-204*
indexing property, *15-17*
INDEXTYPE clause
of CREATE INDEX, *13-105*, *13-110*
indextypes
adding operators, *10-159*
altering, *10-159*
associating statistics types with, *12-218*
changing implementation type, *10-159*
comments on, *12-249*
creating, *13-142*
disassociating statistics types from, *15-204*, *16-14*
drop routines, invoking, *16-12*
granting system privileges for, *18-30*
indexes based on, *13-105*
instances, *13-105*
removing from the database, *16-14*
INDEXTYPES clause
of ASSOCIATE STATISTICS, *12-218*, *12-220*
of DISASSOCIATE STATISTICS, *15-204*
inequality test, *6-4*
INHERIT PRIVILEGES object privilege
on a user, *18-57*
INITCAP function, *7-142*
INITIAL parameter
of STORAGE clause, *8-54*
initialization parameters
changing session settings, *11-87*
setting using ALTER SESSION, *11-95*
INITIALIZED EXTERNALLY clause
of CREATE CONTEXT, *13-43*
INITIALIZED GLOBALLY clause
of CREATE CONTEXT, *13-43*
INITIALLY DEFERRED clause
of constraints, *8-3*
INITIALLY IMMEDIATE clause
of constraints, *8-3*
INITRANS parameter
of ALTER CLUSTER, *10-36*
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW LOG, *11-37*
of ALTER TABLE, *12-29*

INITRANS parameter *(continued)*
of CREATE INDEX. *See* CREATE TABLE, *13-122*
of CREATE MATERIALIZED VIEW LOG. *See* CREATE TABLE, *14-38*
of CREATE MATERIALIZED VIEW. *See* CREATE TABLE, *14-4*
of CREATE TABLE, *8-49*
inline constraints
of ALTER TABLE, *12-29*
of CREATE TABLE, *15-17*
inline views, *9-16*
lateral, *19-61*
INMEMORY hint, *2-108*
INMEMORY_PRUNING hint, *2-108*
inner joins, *9-13*, *19-39*
inner-N reporting, *7-301*
INSERT
direct-path versus conventional, *18-61*
INSERT clause
of MERGE, *19-3*
INSERT statement, *18-61*
append, *2-94*, *2-95*
error logging, *18-74*
INSERTCHILDXML function, *7-143*
INSERTCHILDXMLAFTER function, *7-145*
INSERTCHILDXMLBEFORE function, *7-146*
inserts
and simultaneous update, *19-1*
conditional, *18-61*
conventional, *18-61*
direct-path, *18-61*
multitable, *18-61*
examples, *18-77*
single-table, *18-61*
using MERGE, *19-3*
INSERTXMLAFTER function, *7-147*
INSERTXMLBEFORE function, *7-148*
instance recovery
continue after interruption, *10-56*
INSTANCE session parameter, *11-96*
instances
making index extents available to, *10-135*
setting parameters for, *12-21*
INSTR function, *7-149*
INSTR2 function, *7-149*
INSTR4 function, *7-149*
INSTRB function, *7-149*
INSTRC function, *7-149*
integers
generating unique, *15-1*
in SQL syntax, *2-57*
precision of, *2-58*
syntax of, *2-57*
integrity constraints
See constraints
internal LOBs, *2-29*
International Organization for Standardization (ISO), *1-1*
standards, *1-1*, *C-1*
INTERSECT set operator, *4-6*
interval
arithmetic, *2-24*
data types, *2-20*
literals, *2-64*
interval conditions, *6-32*
INTERVAL DAY TO SECOND data type, *2-24*
INTERVAL expressions, *5-25*
interval partitioning, *12-118*, *15-98*
changing the interval, *12-118*
INTERVAL YEAR TO MONTH data type, *2-24*
INTO clause
of EXPLAIN PLAN, *18-18*
of INSERT, *18-61*
INVALIDATE GLOBAL INDEXES clause
of ALTER TABLE, *12-29*
inverse distribution functions, *7-237*, *7-239*
invoker rights
altering for a Java class, *10-163*
defining for a Java class, *13-148*, *13-149*
invoker's rights views, *15-188*
IS [NOT] EMPTY conditions, *6-13*
IS ANY condition, *6-10*
IS JSON condition, *6-24*
IS OF type condition, *6-36*
IS PRESENT condition, *6-11*
ISO
See International Organization for Standardization (ISO)
ITERATION_NUMBER function, *7-151*

J

Java
class
creating, *13-148*, *13-150*
dropping, *16-16*
resolving, *10-163*, *13-149*
Java source schema object
creating, *13-149*
resource
creating, *13-148*, *13-150*
dropping, *16-16*
schema object
name resolution of, *13-151*
source
compiling, *10-163*, *13-149*
creating, *13-148*
dropping, *16-16*
job scheduler object privileges, *18-30*

JOIN clause
of CREATE DIMENSION, 13-76
join groups
altering, 10-161
creating, 13-146
dropping, 16-15
JOIN KEY clause
of ALTER DIMENSION, 10-94
of CREATE DIMENSION, 13-75
join views
example, 15-193
making updatable, 15-190
modifying, 15-201, 18-67, 19-141
joins, 9-12
antijoins, 9-15
band, 9-13
conditions
defining, 9-12
cross, 19-71
equijoins, 9-12
full outer, 19-39
inner, 9-13, 19-39
left outer, 19-39
natural, 19-72
outer, 9-14
and data densification, 9-14
on grouped tables, 9-14
restrictions, 9-14
parallel, 2-127
right outer, 19-39
self, 9-13
semijoins, 9-16
without join conditions, 9-13
JSON_ARRAY function, 7-153
JSON_ARRAYAGG function, 7-154
JSON_DATAGUIDE function, 7-156
JSON_EXISTS condition, 6-26
JSON_OBJECT function, 7-157
JSON_OBJECTAGG function, 7-159
JSON_QUERY function, 7-161
JSON_TABLE function, 7-167
JSON_TEXTCONTAINS condition, 6-29
JSON_VALUE function, 7-176
Julian dates, 2-21

K

KEEP DATAFILES clause
of DROP PLUGGABLE DATABASE, 17-13
KEEP keyword
of FIRST function, 7-133
of LAST function, 7-133
with aggregate functions, 7-13
KEEP parameter
of CREATE SEQUENCE, 15-6
KEEP SEQUENCE object privilege
on a sequence, 18-56
key compression
See prefix compression
key management framework
granting system privileges for, 18-30
managing, 10-5
key-preserved tables, 15-190
keys, eliminating repetition, 10-135
keywords, 2-141
in object names, 2-141
optional, A-3
required, A-2
KILL SESSION clause
of ALTER SYSTEM, 12-12

L

LAG function, 7-181
large object functions, 7-8
large objects
See LOB data types
LAST function, 7-182
LAST_DAY function, 7-183
LAST_VALUE function, 7-184
lateral inline views, 19-61
LEAD function, 7-187
LEADING hint, 2-108
LEAST function, 7-188
left outer joins, 19-39
LENGTH function, 7-189
LENGTH2 function, 7-189
LENGTH4 function, 7-189
LENGTHB function, 7-189
LENGTHC function, 7-189
less than tests, 6-4
LEVEL clause
of ALTER DIMENSION, 10-93
of CREATE DIMENSION, 13-75, 13-76
level columns
specifying default values, 15-17
LEVEL pseudocolumn, 3-2, 19-39
levels
adding to a dimension, 10-95
dropping from a dimension, 10-95
of dimensions, defining, 13-75
libraries
creating, 14-1
granting system privileges for, 18-30
re-creating, 14-2
removing from the database, 17-1
library units
See Java schema objects
LIKE conditions, 6-15
linear regression functions, 7-292

LIST CHAINED ROWS clause
of ANALYZE, *12-216*
list partitioning
adding default partition, *12-29*
adding partitions, *12-29*
adding values, *12-123*, *12-124*
creating a default partition, *15-17*
creating partitions, *15-17*
dropping values, *12-123*, *12-124*
merging default with nondefault partitions, *12-29*
splitting default partition, *12-134*
list subpartitions
adding, *12-29*
LISTAGG function, *7-190*
listeners
registering, *12-20*
literals, *2-55*
datetime, *2-60*
interval, *2-64*
LN function, *7-194*
LNNVL function, *7-195*
LOB columns
adding, *12-29*
compressing, *15-88*
creating from LONG columns, *2-18*, *12-101*
deduplication, *15-88*
defining properties
for materialized views, *14-12*
encrypting, *15-89*
modifying, *12-99*
modifying storage, *12-29*
restricted in joins, *9-12*
restrictions on, *2-29*
storage characteristics of materialized views, *11-14*
LOB data types, *2-29*
LOB storage clause
for partitions, *12-29*
of ALTER MATERIALIZED VIEW, *11-14*, *11-19*
of ALTER TABLE, *12-29*, *12-51*
of CREATE MATERIALIZED VIEW, *14-4*, *14-12*, *14-14*
of CREATE TABLE, *15-17*, *15-29*
LOBs
attributes, initializing, *2-30*
columns
difference from LONG and LONG RAW, *2-29*
populating, *2-30*
external, *2-29*
internal, *2-29*
locators, *2-29*
logging attribute, *15-17*
LOBs *(continued)*
modifying physical attributes, *12-29*
number of bytes manipulated in, *15-86*
saving old versions, *15-86*, *15-87*
saving values in a cache, *12-98*, *15-115*
specifying directories for, *13-81*
storage
attributes, *15-17*
characteristics, *8-47*
in-line, *15-17*
tablespace for
defining, *15-69*
LOCAL clause
of CREATE INDEX, *13-105*, *13-113*
local users, *14-108*, *15-172*
locale independent, *2-73*
locally managed tablespaces
altering, *12-176*
storage attributes, *8-54*
locally partitioned indexes, *13-105*
LOCALTIMESTAMP function, *7-196*
location transparency, *15-12*
LOCK TABLE statement, *18-80*
locking, overriding automatic, *18-80*
locks, *18-80*
data, *B-1*
dictionary, *B-4*
row (TX), *B-1*
table (TM), *B-1*
See also table locks
log data
collection during update operations, *10-73*
log file clauses
of ALTER DATABASE, *10-46*
log files
adding, *10-40*
dropping, *10-40*
modifying, *10-40*
registering, *10-78*
renaming, *10-64*
specifying for the database, *13-58*
LOG function, *7-197*
log groups
adding, *12-85*
dropping, *12-85*
LOGFILE clause
OF CREATE DATABASE, *13-58*
LOGFILE GROUP clause
of CREATE CONTROLFILE, *13-44*
logging
and redo log size, *8-43*
specifying minimal, *8-43*
supplemental
dropping, *10-74*
supplemental, adding log groups, *12-29*

logging *(continued)*
supplemental, dropping log groups, *12-29*
LOGGING clause,
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW, *11-27*
of ALTER MATERIALIZED VIEW LOG, *11-35*
of ALTER TABLE, *12-80*
of ALTER TABLESPACE, *12-171*
of CREATE MATERIALIZED VIEW, *14-4*
of CREATE MATERIALIZED VIEW LOG, *14-42*
of CREATE TABLE, *15-17*
of CREATE TABLESPACE, *15-141*
logical conditions, *6-9*
logical standby database
aborting, *10-81*
activating, *10-76*
stopping, *10-81*
LOGICAL_READS_PER_CALL parameter
of ALTER PROFILE, *11-75*
LOGICAL_READS_PER_SESSION parameter
of ALTER PROFILE, *11-75*
of ALTER RESOURCE COST, *11-79*
LogMiner
granting system privileges for, *18-30*
supplemental logging, *12-29*, *15-17*
LONG columns
and domain indexes, *12-101*
converting to LOB, *2-18*, *12-101*
restrictions on, *2-18*
to store text strings, *2-18*
to store view definitions, *2-18*
where referenced from, *2-18*
LONG data type, *2-18*
in triggers, *2-19*
LONG RAW data type, *2-28*
converting from CHAR data, *2-29*
LONG VARGRAPHIC data type
DB2, *2-35*
SQL/DS, *2-35*
LOWER function, *7-197*
LPAD function, *7-198*
LTRIM function, *7-199*

M

MAKE_REF function, *7-200*
managed recovery
of database, *10-43*
managed standby recovery
as background process, *10-61*
create a logical standby from the physical standby, *10-62*
overriding delays, *10-61*
returning control during, *10-61*, *10-62*
managed standby recovery *(continued)*
terminating existing, *10-62*, *10-63*
MANAGED STANDBY RECOVERY clause
of ALTER DATABASE, *10-60*
MAPPING TABLE clause
of ALTER TABLE, *12-125*, *12-149*
mapping tables
of index-organized tables, *12-149*, *15-79*
modifying, *12-29*
master databases, *14-4*
master tables, *14-4*
MATCHES condition, *6-2*
MATCHES operator, *4-1*
materialized join views, *14-38*
materialized view logs, *14-38*
creating, *14-38*
excluding new values from, *11-42*
logging changes to, *11-35*
object ID based, *11-41*
parallelizing creation, *14-38*
partition attributes, changing, *11-35*
partitioned, *14-38*
physical attributes
changing, *11-35*
specifying, *14-38*
purging, *11-43*, *14-46*
refreshing, *11-43*, *14-47*
removing from the database, *17-5*
required for fast refresh, *14-38*
required for synchronous refresh, *14-38*
rowid based, *11-42*
saving new values in, *11-42*
saving old values in, *14-46*
staging logs, *14-38*
storage attributes
specifying, *14-38*
materialized views, *14-23*
changing from rowid-based to primary-key-based, *11-30*
changing to primary-key-based, *11-41*
complete refresh, *11-29*, *14-25*
compression of, *11-25*, *14-20*
constraints on, *8-3*
creating, *14-4*
creating comments about, *12-247*
degree of parallelism, *11-14*, *11-35*
during creation, *14-4*
enabling and disabling query rewrite, *14-31*
examples, *14-4*, *14-38*
fast refresh, *11-28*, *14-24*, *14-25*
for data warehousing, *14-5*
for replication, *14-4*
forced refresh, *11-29*
granting system privileges for, *18-30*

materialized views *(continued)*
index characteristics
changing, *11-26*
indexes that maintain, *14-23*
join, *14-38*
LOB storage attributes, *11-14*
logging changes to, *11-27*
master table, dropping, *17-5*
object type, creating, *14-4*
partitions, *11-14*
compression of, *11-25*, *14-20*
physical attributes, *14-4*
changing, *11-14*, *11-44*
primary key, *14-27*
recording values in master table, *11-41*
query rewrite
eligibility for, *8-3*
enabling and disabling, *11-31*
re-creating during refresh, *11-29*
refresh, *10-81*
after DML on master table, *11-29*, *14-25*
mode, changing, *11-14*
on next COMMIT, *11-29*, *14-25*
using trusted constraints, *14-29*
refresh, time, changing, *11-14*
refreshing, *10-81*
removing from the database, *17-3*
restricting scope of, *14-4*
retrieving data from, *19-39*
revalidating, *11-33*
rowid, *14-4*
rowid values
recording in master table, *11-41*
saving blocks in a cache, *11-27*
storage attributes, *14-4*
changing, *11-14*, *11-44*
subquery, *14-4*
suppressing creation of default index, *14-23*
synonyms for, *15-12*
when to populate, *14-4*
MAX function, *7-200*
MAXDATAFILES parameter
of CREATE CONTROLFILE, *13-50*
of CREATE DATABASE, *13-57*
MAXEXTENTS parameter
of STORAGE clause, *8-56*
MAXINSTANCES parameter
of CREATE CONTROLFILE, *13-50*
OF CREATE DATABASE, *13-57*
MAXLOGFILES parameter
of CREATE CONTROLFILE, *13-49*
of CREATE DATABASE, *13-59*
MAXLOGHISTORY parameter
of CREATE CONTROLFILE, *13-49*
of CREATE DATABASE, *13-59*
MAXLOGMEMBERS parameter
of CREATE CONTROLFILE, *13-49*
of CREATE DATABASE, *13-59*
MAXSIZE clause
of ALTER DATABASE, *10-46*
MAXTRANS parameter
of physical_attributes_clause, *8-50*
MAXVALUE parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-86*
of CREATE SEQUENCE, *15-5*
media recovery
avoid on startup, *10-65*
designing, *10-56*
disabling, *10-63*
from specified redo logs, *10-56*
of data files, *10-56*
of database, *10-56*
of standby database, *10-56*
of tablespaces, *10-56*
performing ongoing, *10-60*
preparing for, *10-68*
restrictions, *10-56*
sustained standby recovery, *10-60*
MEDIAN function, *7-202*
median values, *7-239*
MEMBER conditions, *6-14*
membership conditions, *6-14*, *6-33*
MERGE ANY VIEW system privilege, *18-50*
MERGE hint, *2-109*
MERGE PARTITIONS clause
of ALTER TABLE, *12-29*
MERGE statement, *19-1*
deletes during, *19-1*
error logging, *19-1*
inserts during, *19-1*
updates during, *19-1*
MERGE VIEW object privilege on a view, *18-58*
merge_insert_clause
of MERGE, *19-1*
migrated rows
listing, *12-216*
of clusters, *12-209*
MIN function, *7-204*
MINEXTENTS parameter
of STORAGE clause, *8-55*
MINIMIZE RECORDS PER BLOCK clause
of ALTER TABLE, *12-87*
MINIMUM EXTENT clause
of ALTER TABLESPACE, *12-177*
of CREATE TABLESPACE, *15-150*
mining models
auditing, *12-228*
comments on, *12-249*
MINUS set operator, *4-6*

MINVALUE parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-85*
of CREATE SEQUENCE, *15-5*
MOD function, *7-206*
MODE clause
of LOCK TABLE, *18-80*
MODEL clause
of SELECT, *19-39*, *19-48*
model conditions, *6-10*
IS ANY, *6-10*
IS PRESENT, *6-11*
model expression, *5-28*
model functions, *7-21*
MODEL_MIN_ANALYSIS hint, *2-109*
MODIFY clause
of ALTER TABLE, *12-99*
MODIFY CONSTRAINT clause
of ALTER TABLE, *12-29*, *12-47*
of ALTER VIEW, *12-208*
MODIFY DEFAULT ATTRIBUTES clause
of ALTER INDEX, *10-135*, *10-139*
of ALTER TABLE, *12-29*
MODIFY LOB storage clause
of ALTER MATERIALIZED VIEW, *11-14*, *11-20*
of ALTER TABLE, *12-29*
MODIFY NESTED TABLE clause
of ALTER TABLE, *12-29*, *12-47*
MODIFY PARTITION clause
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW, *11-26*
of ALTER TABLE, *12-119*
MODIFY scoped_table_ref_constraint clause
of ALTER MATERIALIZED VIEW, *11-14*
MODIFY SUBPARTITION clause
of ALTER INDEX, *10-135*
MODIFY VARRAY clause
of ALTER TABLE, *12-29*, *12-54*
MON datetime format element, *2-77*
MONITOR hint, *2-110*
MONITORING USAGE clause
of ALTER INDEX, *10-152*
MONTH datetime format element, *2-77*
MONTHS_BETWEEN function, *7-207*
MOUNT clause
of ALTER DATABASE, *10-54*
MOVE clause
of ALTER TABLE, *12-29*, *12-75*
of CREATE PLUGGABLE DATABASE, *14-91*
MOVE ONLINE clause
of ALTER TABLE, *12-148*
MOVE SUBPARTITION clause
of ALTER TABLE, *12-29*
MTS
See shared server
multi-threaded server
See shared server
multilevel collections, *15-91*
multiset conditions, *6-12*
MULTISET EXCEPT operator, *4-7*
MULTISET INTERSECT operator, *4-8*
MULTISET keyword
of CAST function, *7-51*
multiset operators, *4-6*
MULTISET EXCEPT, *4-7*
MULTISET INTERSECT, *4-8*
MULTISET UNION, *4-9*
MULTISET UNION operator, *4-9*
multitable inserts, *18-61*
conditional, *18-61*
examples, *18-77*
unconditional, *18-61*
multitenant container databases
See CDBs

N

NAME clause
of SET TRANSACTION, *19-131*
NAMED clause
of CREATE JAVA, *13-150*
namespaces
and object naming rules, *2-142*
database, *2-143*
for nonschema objects, *2-143*
for schema objects, *2-142*
NANVL function, *7-207*
national character set
changing, *10-82*
multibyte character data, *2-31*
variable-length strings, *2-13*
NATIONAL CHARACTER SET parameter
of CREATE DATABASE, *13-58*
natural joins, *19-72*
NCHAR data type, *2-11*
NCHR function, *7-208*
NCLOB data type, *2-31*
nested subqueries, *9-16*
NESTED TABLE clause
of ALTER TABLE, *12-29*, *12-49*
of CREATE TABLE, *15-17*, *15-28*
nested tables, *2-37*, *6-12*, *7-242*, *7-308*
changing returned value, *12-29*
combining, *4-6*
compared with varrays, *2-48*
comparison rules, *2-48*
creating, *15-166*
creating from existing columns, *7-75*
defining as index-organized tables, *12-29*

nested tables *(continued)*
determining hierarchy, *6-14*
dropping the body of, *18-12*
dropping the specification of, *18-10*
in materialized views, *14-12*, *14-13*
indexing columns of, *13-105*
modifying, *12-29*
modifying column properties, *12-49*
multilevel, *15-91*
partitioned nested table columns, *12-134*
storage characteristics of, *12-29*, *15-17*
NEW_TIME function, *7-209*
NEXT clause
of ALTER MATERIALIZED VIEW ... REFRESH, *11-30*
NEXT parameter
of STORAGE clause, *8-54*
NEXT_DAY function, *7-210*
NEXTVAL pseudocolumn, *3-3*, *15-2*
NLS_CHARSET_DECL_LEN function, *7-211*
NLS_CHARSET_ID function, *7-211*
NLS_CHARSET_NAME function, *7-212*
NLS_COLLATION_ID function, *7-212*
NLS_COLLATION_NAME function, *7-213*
NLS_DATE_LANGUAGE initialization parameter, *2-77*
NLS_INITCAP function, *7-214*
NLS_LANGUAGE initialization parameter, *2-77*, *9-11*
NLS_LOWER function, *7-216*
NLS_SORT initialization parameter, *9-11*
NLS_TERRITORY initialization parameter, *2-77*
NLS_UPPER function, *7-216*
NLSSORT function, *7-217*
NO FORCE LOGGING clause
of ALTER DATABASE, *10-68*
of ALTER TABLESPACE, *12-182*
NO_CLUSTERING hint, *2-111*
NO_EXPAND hint, *2-111*
NO_FACT hint, *2-112*
NO_GATHER_OPTIMIZER_STATISTICS hint, *2-112*
NO_INDEX hint, *2-112*
NO_INDEX_FFS hint, *2-113*
NO_INDEX_SS hint, *2-113*
NO_INMEMORY hint, *2-114*
NO_INMEMORY_PRUNING hint, *2-114*
NO_MERGE hint, *2-114*
NO_MONITOR hint, *2-115*
NO_PARALLEL hint, *2-115*
NO_PARALLEL_INDEX, *2-116*
NO_PQ_CONCURRENT_UNION hint, *2-116*
NO_PQ_SKEW hint, *2-117*
NO_PUSH_PRED hint, *2-117*
NO_PUSH_SUBQ hint, *2-117*
NO_PX_JOIN_FILTER hint, *2-118*
NO_QUERY_TRANSFORMATION hint, *2-118*
NO_RESULT_CACHE hint, *2-118*
NO_REWRITE hint, *2-118*
NO_STAR_TRANSFORMATION hint, *2-119*
NO_STATEMENT_QUEUING hint, *2-119*
NO_UNNEST hint, *2-120*
NO_USE_BAND hint, *2-120*
NO_USE_CUBE hint, *2-120*
NO_USE_HASH hint, *2-120*
NO_USE_MERGE hint, *2-121*
NO_USE_NL hint, *2-121*
NO_XML_QUERY_REWRITE hint, *2-121*
NO_XMLINDEX_REWRITE hint, *2-122*
NO_ZONEMAP hint, *2-122*
NOARCHIVELOG clause
of ALTER DATABASE, *10-46*
of CREATE CONTROLFILE, *13-50*
OF CREATE DATABASE, *10-56*, *13-60*
NOAUDIT statement, *19-10*
for unified auditing, *19-15*
locks, *B-5*
NOCACHE clause
of ALTER MATERIALIZED VIEW, *11-27*
of ALTER MATERIALIZED VIEW LOG, *11-41*
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-86*
of ALTER TABLE, *15-114*
of CREATE CLUSTER, *13-40*
of CREATE MATERIALIZED VIEW, *14-22*
of CREATE MATERIALIZED VIEW LOG, *14-43*
of CREATE SEQUENCE, *15-6*
NOCACHE hint, *2-111*
NOCOMPRESS clause
of ALTER INDEX ... REBUILD, *13-123*
of CREATE TABLE, *15-79*
NOCOPY clause
of CREATE PLUGGABLE DATABASE, *14-91*
NOCYCLE parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-85*
of CREATE SEQUENCE, *15-5*
NOFORCE clause
of CREATE JAVA, *13-149*
of CREATE VIEW, *15-182*
NOKEEP parameter
of CREATE SEQUENCE, *15-6*
NOLOGGING mode
and force logging mode, *8-42*
for nonpartitioned objects, *8-42*
for partitioned objects, *8-42*
NOMAXVALUE parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-85*

NOMAXVALUE parameter *(continued)*
of CREATE SEQUENCE, *15-5*
NOMINIMIZE RECORDS PER BLOCK clause
of ALTER TABLE, *12-87*
NOMINVALUE parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-85*
of CREATE SEQUENCE, *15-5*
NOMONITORING USAGE clause
of ALTER INDEX, *10-152*
NONE clause
of SET ROLE, *19-128*
nonempty subsets of, *7-242*
nonequivalency tests, *6-33*
nonschema objects
list of, *2-139*
namespaces, *2-143*
NOORDER parameter
of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-86*
of CREATE SEQUENCE, *15-6*
NOPARALLEL clause
of CREATE INDEX, *8-46*, *15-116*
NOPARALLEL hint, *2-115*
NOPARALLEL_INDEX hint, *2-116*
NORELY clause
of constraints, *8-3*
NORESETLOGS clause
of CREATE CONTROLFILE, *13-49*
NOREVERSE parameter
of ALTER INDEX ... REBUILD, *10-147*, *10-148*
NOREWRITE hint, *2-118*
NOROWDEPENDENCIES clause
of CREATE CLUSTER, *13-40*
of CREATE TABLE, *15-117*
NOSORT clause
of ALTER INDEX, *13-125*
NOT condition, *6-9*
NOT DEFERRABLE clause
of constraints, *8-3*
NOT IDENTIFIED clause
of ALTER ROLE. *See* CREATE ROLE, *11-80*
of CREATE ROLE, *14-108*
NOT IN subqueries
converting to NOT EXISTS subqueries, *7-195*
NOT NULL clause
of CREATE TABLE, *15-17*
NOWAIT clause
of LOCK TABLE, *18-83*
NTH_VALUE function, *7-220*
NTILE function, *7-222*
null, *2-84*
difference from zero, *2-84*
null *(continued)*
in conditions, *2-85*
table of, *2-85*
in functions, *7-2*
with comparison conditions, *2-85*
null conditions, *6-21*
NULL-related functions, *7-11*
NULLIF function, *7-223*
as a form of CASE expression, *7-223*
NUMBER data type, *2-14*
converting to VARCHAR2, *2-68*
precision, *2-14*
scale, *2-14*
number format models, *2-68*
number functions, *7-4*
numbers
comparison rules, *2-44*
floating-point, *2-14*, *2-16*
in SQL syntax, *2-57*
precision of, *2-59*
spelling out, *2-79*
syntax of, *2-58*
numeric data type, *2-14*
numeric functions, *7-4*
numeric precedence, *2-18*
NUMTODSINTERVAL function, *7-224*
NUMTOYMINTERVAL function, *7-225*
NVARCHAR2 data type, *2-13*
NVL function, *7-226*
NVL2 function, *7-227*

O

object access expressions, *5-30*
OBJECT IDENTIFIER clause
of CREATE TABLE, *15-17*
object identifiers, *2-37*
contained in REFs, *2-37*
primary key, *15-17*
specifying, *15-17*
specifying an index on, *15-17*
system-generated, *15-17*
object instances
types of, *6-36*
object privileges
granting, *14-106*
multiple, *14-113*
on specific columns, *18-30*
on a database object
revoking, *19-24*
revoking, *19-26*
from a role, *19-24*, *19-29*
from a user, *19-24*, *19-29*
from PUBLIC, *19-29*
object reference functions, *7-21*

object tables
- adding rows to, *18-61*
- as part of hierarchy, *15-17*
- creating, *15-17*, *15-19*
- querying, *15-17*
- system-generated column name, *15-123*, *15-125*, *15-186*, *15-187*
- updating to latest version, *12-29*
- upgrading, *12-29*

object type columns
- defining properties
  - for materialized views, *14-12*, *14-13*
- in a type hierarchy, *15-17*
- membership in hierarchy, *12-29*
- modifying properties
  - for tables, *12-29*, *12-48*
- substitutability, *12-29*

object type materialized views
- creating, *14-4*

object types, *2-36*
- associating statistics types with, *12-218*
- attributes, *2-151*
  - in a type hierarchy, *15-17*
  - membership in hierarchy, *12-29*
  - substitutability, *12-29*
- bodies
  - creating, *15-168*
  - re-creating, *15-169*
- comparison rules, *2-48*
  - MAP function, *2-48*
  - ORDER function, *2-48*
- components of, *2-36*
- creating, *15-166*
- defining member methods of, *15-168*
- disassociating statistics types from, *15-204*, *18-10*
- dropping the body of, *18-12*
- dropping the specification of, *18-10*
- granting system privileges for, *18-30*
- identifiers, *3-8*
- incomplete, *15-166*
- methods, *2-151*
- privileges on subtypes, *18-39*
- references to. *See* REFs, *2-37*
- statistics types, *12-218*
- values of, *3-8*

object views, *15-179*
- base tables
  - adding rows, *18-61*
- creating, *15-186*
- defining, *15-179*
- querying, *15-179*

OBJECT_ID pseudocolumn, *3-8*, *15-123*, *15-125*, *15-186*, *15-187*

OBJECT_VALUE pseudocolumn, *3-8*

objects
- *See* object types or database objects

ODCIIndexInsert method
- indextype support of, *10-159*, *13-142*

OF clause
- of CREATE VIEW, *15-186*

of CREATE OPERATOR, *14-58*

OFFLINE clause
- of ALTER TABLESPACE, *12-183*
- of CREATE TABLESPACE, *15-153*

OIDINDEX clause
- of CREATE TABLE, *15-17*

OIDs
- *See* object identifiers

OLAP functions, *7-21*

ON clause
- of CREATE OUTLINE, *14-64*

ON COMMIT clause
- of CREATE TABLE, *15-66*

ON DEFAULT clause
- of AUDIT, *12-228*
- of NOAUDIT, *19-10*

ON DELETE CASCADE clause
- of constraints, *8-3*

ON DELETE SET NULL clause
- of constraints, *8-3*

ON DIRECTORY clause
- of AUDIT, *12-228*, *12-229*
- of NOAUDIT, *19-10*

ON MINING MODEL clause
- of AUDIT, *12-228*

ON object clause
- of NOAUDIT, *19-10*
- of REVOKE, *19-24*

ON PREBUILT TABLE clause
- of CREATE MATERIALIZED VIEW, *14-18*

online backup
- of tablespaces, ending, *12-180*

ONLINE clause
- of ALTER TABLESPACE, *12-183*
- of CREATE INDEX, *13-126*
- of CREATE TABLESPACE, *15-153*

online indexes, *13-126*
- rebuilding, *12-148*

online redo logs
- reinitializing, *10-69*

OPEN clause
- of ALTER DATABASE, *10-54*

OPEN READ ONLY clause
- of ALTER DATABASE, *10-40*

OPEN READ WRITE clause
- of ALTER DATABASE, *10-40*

operands, *4-1*

operating system files
- dropping, *18-7*

operating system files *(continued)*
  removing, *10-67*
operators, *4-1*
  adding to indextypes, *10-160*
  altering, *11-48*
  arithmetic, *4-2*
  binary, *4-1*
  COLLATE, *4-3*
  comments on, *12-249*
  concatenation, *4-4*
  CONNECT_BY_ROOT, *4-6*
  dropping from indextypes, *10-160*
  granting system privileges for, *18-30*
  MULTISET EXCEPT, *4-7*
  MULTISET INTERSECT, *4-8*
  MULTISET UNION, *4-9*
  precedence, *4-2*
  PRIOR, *4-5*
  set, *4-6*
  specifying implementation of, *14-59*
  unary, *4-1*
  user-defined, *4-10*
    binding to a function, *11-48*, *14-58*
    compiling, *11-48*
    creating, *14-58*
    dropping, *17-8*
    how bindings are implemented, *14-60*
    implementation type, *14-60*
OPT_PARAM hint, *2-123*
OPTIMAL parameter
  of STORAGE clause, *8-58*
OR condition, *6-10*
OR REPLACE clause
  of CREATE CONTEXT, *13-43*
  of CREATE DIRECTORY, *13-82*
  of CREATE FUNCTION, *13-101*, *13-149*
  of CREATE LIBRARY, *14-2*
  of CREATE OUTLINE, *14-63*
  of CREATE PACKAGE, *14-66*
  of CREATE PACKAGE BODY, *14-68*
  of CREATE PROCEDURE, *14-94*
  of CREATE TRIGGER, *15-165*
  of CREATE TYPE, *15-167*
  of CREATE TYPE BODY, *15-169*
  of CREATE VIEW, *15-182*
ORA_DST_AFFECTED function, *7-230*
ORA_DST_CONVERT function, *7-230*
ORA_DST_ERROR function, *7-231*
ORA_HASH function, *7-232*
ORA_INVOKING_USER function, *7-233*
ORA_INVOKING_USERID function, *7-233*
ORA_ROWSCN pseudocolumn, *3-9*
Oracle ADVM volumes, *10-118*
Oracle Automatic Storage Management
  migrating nodes in a cluster, *12-15*
Oracle Call Interface, *1-3*
Oracle reserved words, *E-1*
Oracle Text
  built-in conditions, *6-2*
  CATSEARCH, *6-2*
  CONTAINS, *6-2*
  creating domain indexes, *13-131*
  MATCHES, *6-2*
  operators, *4-1*
    CATSEARCH, *4-1*
    CONTAINS, *4-1*
    MATCHES, *4-1*
    SCORE, *4-1*
Oracle Tools
  support of SQL, *1-3*
ORDAudio data type, *2-42*
ORDDicom data type, *2-42*
ORDDoc data type, *2-43*
ORDER BY clause
  of queries, *9-11*
  of SELECT, *9-11*, *19-39*, *19-50*
    with ROWNUM, *3-11*
ORDER clause
  of ALTER SEQUENCE. *See* CREATE SEQUENCE, *11-86*
ORDER parameter
  of CREATE SEQUENCE, *15-6*
ORDER SIBLINGS BY clause
  of SELECT, *19-83*
ORDERED hint, *2-123*
ORDImage data type, *2-43*
ordinal numbers
  specifying, *2-79*
  spelling out, *2-79*
ORDVideo data type, *2-43*
ORGANIZATION EXTERNAL clause
  of CREATE TABLE, *15-17*, *15-77*
ORGANIZATION HEAP clause
  of CREATE TABLE, *15-77*
ORGANIZATION INDEX clause
  of CREATE TABLE, *15-77*
out-of-line constraints
  of CREATE TABLE, *15-17*
outer joins, *9-14*
  restrictions, *9-14*
outlines
  assign to a different category, *11-52*
  assigning to a different category, *11-51*, *11-52*
  copying, *14-64*
  creating, *14-62*
  creating on statements, *14-64*
  dropping from the database, *17-9*
  enabling and disabling dynamically, *14-63*
  for use by current session, *14-63*

outlines *(continued)*
- for use by PUBLIC, *14-63*
- granting system privileges for, *18-30*
- private, use by the optimizer, *11-96*
- rebuilding, *11-51*, *11-52*
- recompiling, *11-52*
- renaming, *11-51*, *11-52*
- replacing, *14-63*
- storing groups of, *14-64*
- use by the optimizer, *12-24*
- use to generate execution plans, *11-96*
- used to generate execution plans, *14-62*

OVER clause
- of analytic functions, *7-14*

OVERFLOW clause
- of ALTER INDEX, *10-141*
- of ALTER TABLE, *12-29*
- of CREATE TABLE, *15-79*

P

P.M. datetime format element, *2-77*

package bodies
- creating, *14-67*
- re-creating, *14-68*
- removing from the database, *17-11*

packaged procedures
- dropping, *17-13*

packages
- associating statistics types with, *12-218*
- creating, *14-65*
- disassociating statistics types from, *15-204*, *17-10*
- redefining, *14-66*
- removing from the database, *17-10*
- synonyms for, *15-12*

PACKAGES clause
- of ASSOCIATE STATISTICS, *12-218*, *12-220*
- of DISASSOCIATE STATISTICS, *15-204*

PARALLEL clause
- of ALTER CLUSTER, *10-37*, *10-39*
- of ALTER INDEX, *10-135*
- of ALTER MATERIALIZED VIEW, *11-14*, *11-21*
- of ALTER MATERIALIZED VIEW LOG, *11-35*, *11-38*
- of ALTER TABLE, *12-29*
- of CREATE CLUSTER, *13-33*
- of CREATE INDEX, *13-127*
- of CREATE MATERIALIZED VIEW, *14-4*, *14-15*
- of CREATE MATERIALIZED VIEW LOG, *14-38*, *14-41*
- of CREATE TABLE, *15-17*, *15-47*

parallel execution, *8-44*
- hints, *2-124*
- of DDL statements, *11-90*
- of DML statements, *11-90*

PARALLEL hint, *2-124*

PARALLEL_INDEX hint, *2-126*

parameter files
- creating, *14-69*
  - from memory, *14-70*

parameters
- in syntax
  - optional, *A-3*
  - required, *A-2*

PARAMETERS clause
- of CREATE INDEX, *13-132*, *13-133*

partial indexes, *13-105*

PARTITION ... LOB storage clause
- of ALTER TABLE, *12-29*

PARTITION BY HASH clause
- of CREATE TABLE, *15-17*, *15-37*

PARTITION BY LIST clause
- of CREATE TABLE, *15-17*, *15-38*

PARTITION BY RANGE clause
- of CREATE TABLE, *15-17*, *15-36*

PARTITION BY REFERENCE clause
- of CREATE TABLE, *15-39*, *15-109*

PARTITION clause
- of ANALYZE, *12-212*
- of CREATE INDEX, *13-128*
- of CREATE TABLE, *15-99*
- of DELETE, *15-199*
- of INSERT, *18-68*
- of LOCK TABLE, *18-80*
- of UPDATE, *19-142*

partition-extended table names
- in DML statements, *2-150*
- restrictions on, *2-149*
- syntax, *2-149*

partitioned index-organized tables
- secondary indexes, updating, *10-154*

partitioned indexes, *2-149*, *13-105*
- local, creating, *13-113*
- user-defined, *13-127*

partitioned tables, *2-149*

partitioning
- by hash, *15-17*, *15-37*
- by list, *15-17*, *15-38*
- by range, *15-17*, *15-36*
- by reference, *15-39*, *15-109*
- clauses
  - of ALTER INDEX, *10-139*
  - of ALTER TABLE, *12-116*
- interval, *15-98*
- of materialized view logs, *11-35*, *14-38*
- of materialized views, *11-14*, *14-4*, *14-8*

partitioning *(continued)*
range with interval partitions, *15-98*
referential constraint, *12-114*, *15-109*
system, *15-110*
partitions
adding, *12-116*
adding rows to, *18-61*
allocating extents for, *12-85*
based on literal values, *15-17*
composite
specifying, *15-105*
converting into nonpartitioned tables, *12-29*
deallocating unused space from, *12-86*
dropping, *12-130*
exchanging with tables, *12-69*
extents
allocating for an index, *10-135*
hash
adding, *12-29*
coalescing, *12-29*
specifying, *15-17*
index, *13-127*
inserting rows into, *18-68*
list, adding, *12-29*
LOB storage characteristics of, *12-29*
locking, *18-80*
logging attribute, *15-17*
logging insert operations, *12-80*
merging, *12-29*
modifying, *12-116*, *12-119*
physical attributes
changing, *12-79*
range
adding, *12-29*
specifying, *15-17*
removing rows from, *12-29*, *15-199*
renaming, *12-29*
revising values in, *19-142*
splitting, *12-134*
storage characteristics, *8-47*
tablespace for
defining, *15-69*
PASSWORD EXPIRE clause
of ALTER USER. *See* CREATE USER, *12-197*
of CREATE USER, *15-176*
PASSWORD_GRACE_TIME parameter
of ALTER PROFILE, *11-76*
of CREATE PROFILE, *14-101*
PASSWORD_LIFE_TIME parameter
of ALTER PROFILE, *11-76*
of CREATE PROFILE, *14-100*
PASSWORD_LOCK_TIME parameter
of ALTER PROFILE, *11-76*
of CREATE PROFILE, *14-101*
PASSWORD_REUSE_MAX parameter
of ALTER PROFILE, *11-76*
of CREATE PROFILE, *14-100*
PASSWORD_REUSE_TIME parameter
of ALTER PROFILE, *11-76*
of CREATE PROFILE, *14-100*
PASSWORD_VERIFY_FUNCTION parameter
of ALTER PROFILE, *11-76*
of CREATE PROFILE, *14-101*
passwords
expiration of, *15-176*
grace period, *14-95*
guaranteeing complexity, *14-95*
limiting use and reuse, *14-95*
locking, *14-95*
making unavailable, *14-95*
parameters
of CREATE PROFILE, *14-97*
special characters in, *15-172*
PATH function, *7-234*
PATH_VIEW, *6-22*, *6-23*
pattern-matching conditions, *6-15*
PCT_ACCESS_DIRECT statistics
for index-organized tables, *12-211*
PCTFREE parameter
of ALTER CLUSTER, *10-36*
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW LOG, *11-37*
of ALTER TABLE, *12-29*
of CREATE MATERIALIZED VIEW LOG. *See* CREATE TABLE., *14-38*
of CREATE MATERIALIZED VIEW. *See* CREATE TABLE., *14-4*
of CREATE TABLE, *8-48*
PCTINCREASE parameter
of STORAGE clause, *8-55*
PCTTHRESHOLD parameter
of CREATE TABLE, *15-17*
PCTUSED parameter
of ALTER CLUSTER, *10-36*
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW LOG, *11-37*
of ALTER TABLE, *12-29*
of CREATE INDEX. *See* CREATE TABLE, *13-122*
of CREATE MATERIALIZED VIEW LOG. *See* CREATE TABLE., *14-38*
of CREATE MATERIALIZED VIEW. *See* CREATE TABLE., *14-4*
of CREATE TABLE, *8-48*
PCTVERSION parameter
of LOB storage, *15-86*
of LOB storage clause, *12-112*
PDBs, *14-71*
administrative user, *14-78*

PDBs *(continued)*
backup, *11-65*
changing
global name, *11-61*
state, *11-65*, *11-68*
storage limits, *11-62*
cloning, *14-84*
creating
by cloning a source PDB, *14-84*
using the seed database, *14-78*
default edition, setting, *11-61*
examples
creating, *14-71*
dropping, *17-11*
modifying, *11-54*
generating file names, *14-79*
granting system privileges for, *18-30*
modifying data files, *11-62*
modifying temporary files, *11-62*
plugging into a CDB, *14-89*
recovery, *11-65*
setting the time zone of, *11-62*
storage limits, *14-79*
unplugging, *11-60*
XML file for plugging in, *14-89*
PERCENT_RANK function, *7-235*
PERCENTILE_CONT function, *7-237*
PERCENTILE_DISC function, *7-239*
PERMANENT clause
of ALTER TABLESPACE, *12-184*
physical attributes clause
of ALTER CLUSTER, *10-37*
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW LOG, *11-37*
of ALTER TABLE, *12-79*
of CREATE CLUSTER, *13-34*
of CREATE MATERIALIZED VIEW, *14-10*
of CREATE TABLE, *15-17*, *15-34*
physical standby database
activating, *10-76*
converting to snapshot standby database, *10-81*
pivot operations, *19-68*
examples, *19-112*
syntax, *19-43*
placeholder expressions, *5-30*
plan management
granting system privileges for, *18-30*
plan stability, *14-62*
PLAN_TABLE sample table, *18-16*
pluggable databases
See PDBs
PM datetime format element, *2-77*
POSIX regular expression standard, *D-1*
POWER function, *7-241*
POWERMULTISET function, *7-242*
POWERMULTISET_BY_CARDINALITY function, *7-243*
PQ_CONCURRENT_UNION hint, *2-127*
PQ_DISTRIBUTE hint, *2-127*
PQ_FILTER hint, *2-130*
PQ_SKEW hint, *2-130*
precedence
of conditions, *6-3*
of numbers, *2-18*
of operators, *4-2*
precision
number of digits of, *2-59*
of NUMBER data type, *2-14*
precompilers, *1-3*
predefined roles, *18-30*
PREDICTION function, *7-244*
PREDICTION_BOUNDS function, *7-248*
PREDICTION_COST function, *7-250*
PREDICTION_DETAILS function, *7-254*
PREDICTION_PROBABILITY function, *7-258*
PREDICTION_SET function, *7-261*
prefix compression, *15-79*
definition, *13-123*
disabling, *13-123*
enabling, *10-148*
of index rebuild, *10-148*
of index-organized tables, *15-79*
PREPARE TO SWITCHOVER clause
of ALTER DATABASE, *10-79*
PRESENTNNV function, *7-265*
PRESENTV function, *7-266*
pretty-printing of XML output, *7-425*
PREVIOUS function, *7-267*
primary database
converting to physical standby database, *10-81*
PRIMARY KEY clause
of constraints, *8-3*
of CREATE TABLE, *15-17*
primary key constraints, *8-3*
enabling, *15-118*
index on, *15-17*
primary keys
generating values for, *15-1*
PRIOR clause
of hierarchical queries, *9-2*
PRIOR operator, *4-5*
PRIVATE clause
of CREATE OUTLINE, *14-63*
private outlines
use by the optimizer, *11-96*
PRIVATE_SGA parameter
of ALTER PROFILE, *11-75*
of ALTER RESOURCE COST, *11-79*

privileges, 14-106
on subtypes of object types, 18-39
revoking from a grantee, 19-26
See also system privileges or object privileges
procedures
3GL,calling, 14-1
calling, 12-243
creating, 14-93
executing, 12-243
external, 14-93
granting system privileges for, 18-30
invalidating local objects dependent on, 17-13
issuing COMMIT or ROLLBACK statements, 11-89
naming rules, 2-144
re-creating, 14-94
recompiling, 11-73
removing from the database, 17-13
synonyms for, 15-12
PROFILE clause
of ALTER USER. *See* CREATE USER, 12-197
of CREATE USER, 15-176
profiles
adding resource limits, 11-75
assigning to a user, 15-176
changing resource limits, 11-75
creating, 14-95
examples, 14-95
deassigning from users, 17-15
dropping resource limits, 11-75
granting system privileges for, 18-30
modifying, examples, 11-76
removing from the database, 17-14
proxy clause
of ALTER USER, 12-195, 12-197
pseudocolumns, 3-1
COLUMN_VALUE, 3-7
CONNECT_BY_ISCYCLE, 3-1
CONNECT_BY_ISLEAF, 3-2
CURRVAL, 3-3
flashback queries, 3-6
in hierarchical queries, 3-1
LEVEL, 3-2
NEXTVAL, 3-3
OBJECT_ID, 3-8, 15-123, 15-125, 15-186, 15-187
OBJECT_VALUE, 3-8
ORA_ROWSCN, 3-9
ROWID, 3-10
ROWNUM, 3-11
version queries, 3-6
XMLDATA, 3-12
PUBLIC clause
of CREATE OUTLINE, 14-63
of CREATE SYNONYM, 15-13
of DROP DATABASE LINK, 16-3
public database links
dropping, 16-3
public synonyms, 15-13
dropping, 17-20
PURGE statement, 19-20
PUSH_PRED hint, 2-130
PUSH_SUBQ hint, 2-131
PX_JOIN_FILTER hint, 2-131

Q

QB_NAME hint, 2-131
queries, 9-1, 19-39
comments in, 9-2
compound, 9-11
correlated
left correlation, 19-39
default locking of, B-3
defined, 9-1
distributed, 9-18
grouping returned rows on a value, 19-39
hierarchical, ordering, 19-83
hierarchical. *See* hierarchical queries, 9-2
hints in, 9-2
join, 9-12, 19-39
locking rows during, 19-39
multiple versions of rows, 19-39
of past data, 19-39
ordering returned rows, 19-39
outer joins in, 19-67
referencing multiple tables, 9-12
select lists of, 9-2
selecting all columns, 19-59
selecting from a random sample of rows, 19-39
sorting results, 9-11
syntax, 9-1
top-level, 9-1
top-N, 3-11, 19-39
query rewrite
and dimensions, 13-75
defined, 19-39
QUIESCE RESTRICTED clause
of ALTER SYSTEM, 12-14
QUOTA clause
of ALTER USER. *See* CREATE USER, 12-197
of CREATE USER, 15-175

R

range conditions, 6-32
range partitioning
converting to interval partitioning, 12-118
range partitions
adding, 12-29
creating, 15-17
values of, 15-17
RANK function, 7-268
RATIO_TO_REPORT function, 7-270
RAW data type, 2-28
converting from CHAR data, 2-29
RAWTOHEX function, 7-271
RAWTONHEX function, 7-272
READ ANY TABLE system privilege, 18-49
READ object privilege
on a materialized view, 18-54
on a table, 18-57
on a view, 18-58
READ ONLY clause
of ALTER TABLESPACE, 12-184
of ALTER VIEW, 12-208
READ WRITE clause
of ALTER TABLESPACE, 12-184
of ALTER VIEW, 12-208
REBUILD clause
of ALTER INDEX, 10-135
of ALTER OUTLINE, 11-52
REBUILD PARTITION clause
of ALTER INDEX, 10-146
REBUILD SUBPARTITION clause
of ALTER INDEX, 10-146
REBUILD UNUSABLE LOCAL INDEXES clause
of ALTER TABLE, 12-144
RECOVER AUTOMATIC clause
of ALTER DATABASE, 10-57
RECOVER CANCEL clause
of ALTER DATABASE, 10-42, 10-60
RECOVER clause
of ALTER DATABASE, 10-56
RECOVER CONTINUE clause
of ALTER DATABASE, 10-42, 10-59
RECOVER DATABASE clause
of ALTER DATABASE, 10-42, 10-57
RECOVER DATAFILE clause
of ALTER DATABASE, 10-42, 10-59
RECOVER LOGFILE clause
of ALTER DATABASE, 10-42, 10-59
RECOVER MANAGED STANDBY DATABASE clause
of ALTER DATABASE, 10-43
RECOVER TABLESPACE clause
of ALTER DATABASE, 10-42, 10-58
RECOVERABLE, 10-145, 15-77
See also LOGGING clause
recovery
discarding data, 10-54
distributed, enabling, 12-10
instance, continue after interruption, 10-56
media, designing, 10-56
media, performing ongoing, 10-60
of database, 10-42
recovery clauses
of ALTER DATABASE, 10-42
recursive subquery factoring, 19-57
recycle bin
purging objects from, 19-20
redo log files
specifying, 8-32
specifying for a control file, 13-46
redo logs, 10-54
adding, 10-40, 10-70
applying to logical standby database, 10-81
archive location, 12-9
automatic archiving, 12-3
automatic name generation, 10-56, 10-57
clearing, 10-40
dropping, 10-40, 10-72
enabling and disabling thread, 10-40
manual archiving, 12-3
all, 12-9
by group number, 12-8
by SCN, 12-8
current, 12-8
next, 12-9
with sequence numbers, 12-8
members
adding to existing groups, 10-71
dropping, 10-72
renaming, 10-64
remove changes from, 10-54
reusing, 8-32
size of, 8-32
specifying, 8-32, 13-58
for media recovery, 10-59
specifying archive mode, 13-60
switching groups, 12-13
REF columns
rescoping, 11-14
specifying, 15-17
specifying from table or column, 15-17
REF constraints
defining scope, for materialized views, 11-23
of ALTER TABLE, 12-29
REF function, 7-273
reference partitioning, 15-109
reference-partitioned tables, 12-116
maintenance operations, 12-143

REFERENCES clause
of CREATE TABLE, 15-17
referential integrity constraints, 8-3
REFRESH clause
of ALTER MATERIALIZED VIEW, 11-14, 11-23
of CREATE MATERIALIZED VIEW, 14-9
REFRESH COMPLETE clause
of ALTER MATERIALIZED VIEW, 11-29
of CREATE MATERIALIZED VIEW, 14-4
REFRESH FAST clause
of ALTER MATERIALIZED VIEW, 11-28
of CREATE MATERIALIZED VIEW, 14-4
REFRESH FORCE clause
of ALTER MATERIALIZED VIEW, 11-29
of CREATE MATERIALIZED VIEW, 14-4
REFRESH ON COMMIT clause
of ALTER MATERIALIZED VIEW, 11-29
of CREATE MATERIALIZED VIEW, 14-4
REFRESH ON DEMAND clause
of ALTER MATERIALIZED VIEW, 11-29
of CREATE MATERIALIZED VIEW, 14-4
REFs, 2-37, 8-3
as containers for object identifiers, 2-37
dangling, 12-214
updating, 12-214
validating, 12-214
REFTOHEX function, 7-273
REGEXP_COUNT function, 7-274
REGEXP_INSTR function, 7-279
REGEXP_LIKE condition, 6-19
REGEXP_REPLACE function, 7-283
REGEXP_SUBSTR function, 7-288
REGISTER clause
of ALTER SYSTEM, 12-20
REGISTER LOGFILE clause
of ALTER DATABASE, 10-78
REGR_AVGX function, 7-292
REGR_AVGY function, 7-292
REGR_COUNT function, 7-292
REGR_INTERCEPT function, 7-292
REGR_R2 function, 7-292
REGR_SLOPE function, 7-292
REGR_SXX function, 7-292
REGR_SXY function, 7-292
REGR_SYY function, 7-292
regular expressions
multilingual syntax, D-1
operators, multilingual enhancements, D-2
Oracle support of, D-1
Perl-influenced operators, D-3
subexpressions, 7-280, 7-289
relational tables
creating, 15-17, 15-18
RELY clause
of constraints, 8-3
REMAINDER function, 7-297
RENAME clause
of ALTER INDEX, 10-151
of ALTER OUTLINE, 11-52
of ALTER TABLE, 12-88
of ALTER TABLESPACE, 12-178
of ALTER TRIGGER, 12-193
RENAME CONSTRAINT clause
of ALTER TABLE, 12-114
RENAME DATAFILE clause
of ALTER TABLESPACE, 12-171
RENAME FILE clause
of ALTER DATABASE, 10-40, 10-64
RENAME GLOBAL_NAME clause
of ALTER DATABASE, 10-84
of ALTER PLUGGABLE DATABASE, 11-61
RENAME PARTITION clause
of ALTER INDEX, 10-135
of ALTER TABLE, 12-29
RENAME statement, 19-22
RENAME SUBPARTITION clause
of ALTER INDEX, 10-135
of ALTER TABLE, 12-29
REPLACE function, 7-298
replication
row-level dependency tracking, 13-40, 15-117
reserved words, 2-141, E-1
reset sequence of, 10-54
RESETLOGS parameter
of CREATE CONTROLFILE, 13-48
RESOLVE clause
of ALTER JAVA CLASS, 10-165
of CREATE JAVA, 13-149
RESOLVER clause
of CREATE JAVA, 13-151
Resource Manager, 12-14
resource parameters
of CREATE PROFILE, 14-96
RESOURCE_VIEW, 6-22, 6-23
response time
optimizing, 2-100
restore points
guaranteed, 14-105
preserved, 14-105
using
to flash back a table, 18-26
to flashback the database, 18-23
result cache, 15-115
RESULT_CACHE hint, 2-132
resumable space allocation, 11-91
RESUME clause
of ALTER SYSTEM, 12-14

RETENTION parameter
of LOB storage, 15-87
RETRY_ON_ROW_CHANGE hint, 2-132
RETURNING clause
of DELETE, 15-195
of INSERT, 18-64
of UPDATE, 19-140, 19-146
REUSE clause
of CREATE CONTROLFILE, 13-47
of file specifications, 8-32
REVERSE clause
of CREATE INDEX, 13-125
reverse indexes, 13-125
REVERSE parameter
of ALTER INDEX ... REBUILD, 10-147, 10-148
REVOKE CONNECT THROUGH clause
of ALTER USER, 12-195, 12-197
REVOKE statement, 19-24
locks, B-5
REWRITE hint, 2-133
right outer joins, 19-39
roles, 18-30
authorization
by a password, 14-108
by an external service, 14-108
by the database, 14-108
by the enterprise directory service, 14-108
changing, 11-80
creating, 14-106
disabling
for the current session, 19-126, 19-128
enabling
for the current session, 19-126, 19-128
granting, 18-30
system privileges for, 18-30
to a user, 18-34
to another role, 18-34
to PUBLIC, 18-34
identifying by password, 14-108
identifying externally, 14-108
identifying through enterprise directory service, 14-108
identifying using a package, 14-108
removing from the database, 17-16
revoking, 19-24
from another role, 17-16, 19-28
from PUBLIC, 19-28
from users, 17-16, 19-28
ROLES clause
of CREATE PLUGGABLE DATABASE, 14-78
rollback segments
removing from the database, 17-17
specifying optimal size of, 8-58
rollback segments granting
system privileges for, 18-30
ROLLBACK statement, 19-35
rollback undo, 11-83, 13-52
ROLLUP clause
of SELECT statements, 19-76
ROUND (date) function, 7-299
format models, 7-429
ROUND (number) function, 7-299
routines
calling, 12-243
executing, 12-243
row constructor, 5-33
ROW EXCLUSIVE lock mode, 18-83
row limiting, 19-39
row locking, B-1
ROW SHARE lock mode, 18-83
row value constructor, 5-33
row values
pivoting into columns, 19-68
ROW_NUMBER function, 7-301
row-level dependency tracking, 13-40, 15-117
row-level locking, B-1
ROWDEPENDENCIES clause
of CREATE CLUSTER, 13-40
of CREATE TABLE, 15-117
ROWID data type, 2-33
ROWID pseudocolumn, 2-33, 2-34, 3-10
rowids, 2-33
description of, 2-33
extended
base 64, 2-34
not directly available, 2-34
nonphysical, 2-34
of foreign tables, 2-34
of index-organized tables, 2-34
uses for, 3-10
ROWIDTOCHAR function, 7-302
ROWIDTONCHAR function, 7-303
ROWNUM pseudocolumn, 3-11
rows
adding to a table, 18-61
allowing movement of between partitions, 15-33
inserting
into partitions, 18-68
into remote databases, 18-61
into subpartitions, 18-68
locking, B-1
locks on, B-1
movement between partitions, 15-17
removing
from a cluster, 19-132, 19-134
from a table, 19-132, 19-134
from partitions and subpartitions, 15-199

rows *(continued)*
removing *(continued)*
from tables and views, 15-195
selecting in hierarchical order, 9-2
specifying constraints on, 8-3
storing if in violation of constraints, 12-142
RPAD function, 7-304
RR datetime format element, 2-78
RTRIM function, 7-305
run-time compilation
avoiding, 11-73, 12-206

S

SAMPLE clause
of SELECT, 19-39
of SELECT and subqueries, 19-44
SAVEPOINT statement, 19-38
savepoints
erasing, 13-1
rolling back to, 19-36
specifying, 19-38
scalar subqueries, 5-31
scale
greater than precision, 2-14
of NUMBER data type, 2-14
SCHEMA clause
of CREATE JAVA, 13-150
schema objects, 2-138
defining default buffer pool for, 8-58
dropping, 18-13
in other schemas, 2-147
list of, 2-138
name resolution, 2-145
namespaces, 2-142
naming
examples, 2-144
guidelines, 2-144
rules, 2-140
object types, 2-36
on remote databases, 2-147
partitioned indexes, 2-149
partitioned tables, 2-149
parts of, 2-139
protecting location, 15-12
protecting owner, 15-12
providing alternate names for, 15-12
reauthorizing, 10-2
recompiling, 10-2
referring to, 2-145, 11-96
remote, accessing, 13-69
validating structure, 12-214
schemas
changing for a session, 11-96
creating, 14-113
schemas *(continued)*
definition of, 2-138
scientific notation, 2-69
SCN_TO_TIMESTAMP function, 7-306
SCOPE FOR clause
of ALTER MATERIALIZED VIEW, 11-23
of CREATE MATERIALIZED VIEW, 14-4
SCORE operator, 4-1
SDO_GEOMETRY data type, 2-41
SDO_GEORASTER data type, 2-41
SDO_TOPO_GEOMETRY data type, 2-42
security
enforcing, 15-164
security clauses
of ALTER SYSTEM, 12-17
segment attributes clause
of CREATE TABLE, 15-33
SEGMENT MANAGEMENT clause
of CREATE TABLESPACE, 15-141
segments
space management
automatic, 15-141
manual, 15-141
using bitmaps, 15-141
using free lists, 15-141
table
compacting, 10-143, 11-27, 11-41, 12-88
select lists, 9-2
ordering, 9-11
SELECT object privilege
granting on a view, 18-37
SELECT statement, 9-1, 19-39
self joins, 9-13
semijoins, 9-16
sequences, 3-3, 15-1
accessing values of, 15-2
changing
the increment value, 11-85
creating, 15-1
creating without limit, 15-4
global, 15-7
granting system privileges for, 18-30
guarantee consecutive values, 15-6
how to use, 3-4
increment value, setting, 15-4
incrementing, 15-1
initial value, setting, 15-5
keeping values during transaction replay, 15-6
maximum value
eliminating, 11-86
setting, 15-5
setting or changing, 11-85
minimum value
eliminating, 11-86

sequences *(continued)*
minimum value *(continued)*
setting, *15-5*
setting or changing, *11-85*
number of cached values, changing, *11-85*
ordering values, *11-85*
preallocating values, *15-5*
recycling values, *11-85*
removing from the database, *17-18*
renaming, *19-22*
restarting, *17-18*
at a different number, *11-86*
at a predefined limit, *15-4*
values, *15-5*
reusing, *15-1*
session, *15-7*
stopping at a predefined limit, *15-4*
synonyms for, *15-12*
where to use, *3-4*
server parameter files
creating, *15-8*
from memory, *15-11*
server wallet
keys, *12-17*
service name
of remote database, *13-69*
session control statements, *10-4*
PL/SQL support of, *10-4*
session locks
releasing, *12-12*
SESSION parameter
of CREATE SEQUENCE, *15-7*
session parameters
changing settings, *11-96*
INSTANCE, *11-96*
session sequences, *15-7*
SESSION_ROLES view, *19-127*
sessions
calculating resource cost limits, *11-78*
changing resource cost limits, *11-78*
disconnecting, *12-12*
granting system privileges for, *18-30*
limiting CPU time, *11-79*
limiting data block reads, *11-79*
limiting inactive periods, *11-75*
limiting private SGA space, *11-79*
limiting resource costs, *11-78*
limiting total elapsed time, *11-79*
limiting total resources, *11-75*
modifying characteristics of, *11-87*
restricting, *12-14*
restricting to privileged users, *12-17*
switching to a different instance, *11-96*
terminating, *12-12*
terminating across instances, *12-12*
sessions *(continued)*
time zone setting, *11-96*
SESSIONS_PER_USER parameter
of ALTER PROFILE, *11-75*
SESSIONTIMEZONE function, *7-307*
SET clause
of ALTER SESSION, *11-87*
of ALTER SYSTEM, *12-21*
SET conditions, *6-12*
SET CONSTRAINT(S) statement, *19-125*
SET CONTAINER system privilege, *18-47*
SET DANGLING TO NULL clause
of ANALYZE, *12-214*
SET DATABASE clause
of CREATE CONTROLFILE, *13-47*
SET ENCRYPTION KEY clause
of ALTER SYSTEM, *12-18*
SET ENCRYPTION WALLET clause
of ALTER SYSTEM, *12-17*
SET function, *7-308*
set operators, *4-6*
INTERSECT, *4-6*
MINUS, *4-6*
UNION, *4-6*
UNION ALL, *4-6*
SET ROLE statement, *19-126*
SET STANDBY DATABASE clause
of ALTER DATABASE, *10-77*
SET STATEMENT_ID clause
of EXPLAIN PLAN, *18-18*
SET TIME_ZONE clause
of ALTER DATABASE, *10-52*, *10-87*
of ALTER PLUGGABLE DATABASE, *11-62*
of ALTER SESSION, *11-96*
of CREATE DATABASE, *13-55*
SET TRANSACTION statement, *19-129*
SET UNUSED clause
of ALTER TABLE, *12-105*
SGA
See system global area (SGA)
SHARE ROW EXCLUSIVE lock mode, *18-83*
SHARE UPDATE lock mode, *18-83*
SHARED clause
of CREATE DATABASE LINK, *13-71*
shared pool
flushing, *12-10*
shared server,
processes
creating additional, *12-25*
terminating, *12-25*
system parameters, *12-25*
short-circuit evaluation
DECODE function, *7-102*
SHRINK SPACE clause
of ALTER INDEX, *10-143*

SHRINK SPACE clause *(continued)*
of ALTER MATERIALIZED VIEW, *11-27*
of ALTER MATERIALIZED VIEW LOG, *11-41*
of ALTER TABLE, *12-88*
SHUTDOWN clause
of ALTER SYSTEM, *12-20*
SI_AverageColor data type, *2-43*
SI_Color data type, *2-43*
SI_ColorHistogram data type, *2-43*
SI_FeatureList data type, *2-43*
SI_PositionalColorHistogram data type, *2-43*
SI_StillImage data type, *2-43*
SI_Texture data type, *2-43*
siblings
ordering in a hierarchical query, *19-83*
SIGN function, *7-308*
simple comparison conditions, *6-5*
simple expressions, *5-3*
SIN function, *7-309*
single row function expression, *5-16*
SINGLE TABLE clause
of CREATE CLUSTER, *13-38*
single-row functions, *7-4*
single-table insert, *18-61*
SINH function, *7-310*
SIZE clause
of ALTER CLUSTER, *10-38*
of CREATE CLUSTER, *13-37*
of file specifications, *8-32*
SOME operator, *6-4*
SOUNDEX function, *7-310*
SOURCE_FILE_NAME_CONVERT clause
of CREATE PLUGGABLE DATABASE, *14-90*
SP datetime format element suffix, *2-79*
special characters
in passwords, *14-101*
spelled numbers
specifying, *2-79*
SPLIT PARTITION clause
of ALTER INDEX, *10-135*
of ALTER TABLE, *12-134*
SPTH datetime format element suffix, *2-79*
SQL
See Structured Query Language (SQL)
SQL Developer, *1-3*
SQL Function
Data Mining, *7-119*
FEATURE_COMPARE, *7-119*
SQL functions, *7-2*, *7-431*
ABS, *7-22*
ACOS, *7-22*
ADD_MONTHS, *7-23*
aggregate functions, *7-12*
analytic functions, *7-14*
APPENDCHILDXML, *7-24*
SQL functions *(continued)*
applied to LOB columns, *7-2*
APPROX_COUNT_DISTINCT, *7-25*
APPROX_COUNT_DISTINCT_AGG, *7-26*
APPROX_COUNT_DISTINCT_DETAIL, *7-27*
APPROX_MEDIAN, *7-30*
APPROX_PERCENTILE, *7-33*
APPROX_PERCENTILE_AGG, *7-36*
APPROX_PERCENTILE_DETAIL, *7-37*
ASCII, *7-41*
ASCIISTR, *7-42*
ASIN, *7-42*
ATAN, *7-43*
ATAN2, *7-44*
AVG, *7-45*
BFILENAME, *7-46*
BIN_TO_NUM, *7-47*
BITAND, *7-48*
CARDINALITY, *7-50*
CAST, *7-51*
CEIL, *7-56*
character functions
returning character values, *7-5*
returning number values, *7-5*
character set functions, *7-6*
CHARTOROWID, *7-56*
CHR, *7-57*
CLUSTER_DETAILS, *7-59*
CLUSTER_DISTANCE, *7-62*
CLUSTER_ID, *7-65*
CLUSTER_PROBABILITY, *7-67*
CLUSTER_SET, *7-70*
COALESCE, *7-72*
COLLATION, *7-74*
collation functions, *7-6*
COLLECT, *7-75*
collection functions, *7-8*
COMPOSE, *7-76*
CON_DBID_TO_ID, *7-77*
CON_GUID_TO_ID, *7-78*
CON_NAME_TO_ID, *7-79*
CON_UID_TO_ID, *7-79*
CONCAT, *7-80*
conversion functions, *7-7*
CONVERT, *7-81*
CORR, *7-83*
CORR_K, *7-86*
CORR_S, *7-86*
COS, *7-87*
COSH, *7-87*
COUNT, *7-88*
COVAR_POP, *7-90*
COVAR_SAMP, *7-92*
CUBE_TABLE, *7-93*
CUME_DIST, *7-95*

SQL functions *(continued)*
CURRRENT_DATE, *7-97*
CURRRENT_TIMESTAMP, *7-97*
CV, *7-98*
data cartridge functions, *7-21*
data mining functions, *7-9*
DATAOBJ_TO_MAT_PARTITION, *7-100*
DATAOBJ_TO_PARTITION, *7-100*
datetime functions, *7-6*
DBTIMEZONE, *7-101*
DECODE, *7-102*
DECOMPOSE, *7-103*
DELETEXML, *7-105*
DENSE_RANK, *7-106*
DEPTH, *7-108*
DEREF, *7-108*
DUMP, *7-109*
EMPTY_BLOB, *7-111*
EMPTY_CLOB, *7-111*
encoding and decoding functions, *7-11*
environment and identifier functions, *7-11*
EXISTSNODE, *7-112*
EXP, *7-113*
EXTRACT (datetime), *7-114*
EXTRACT (XML), *7-116*
EXTRACTVALUE, *7-117*
FEATURE_DETAILS, *7-121*
FEATURE_ID, *7-124*
FEATURE_SET, *7-126*
FEATURE_VALUE, *7-129*
FIRST, *7-132*
FIRST_VALUE, *7-134*
FLOOR, *7-136*
FROM_TZ, *7-137*
general comparison functions, *7-7*
GREATEST, *7-138*
GROUP_ID, *7-139*
GROUPING, *7-140*
GROUPING_ID, *7-141*
HEXTORAW, *7-142*
hierarchical functions, *7-8*
INITCAP, *7-142*
INSERTCHILDXML, *7-143*
INSERTCHILDXMLAFTER, *7-145*
INSERTCHILDXMLBEFORE, *7-146*
INSERTXMLAFTER, *7-147*
INSERTXMLBEFORE, *7-148*
INSTR, *7-149*
INSTR2, *7-149*
INSTR4, *7-149*
INSTRB, *7-149*
INSTRC, *7-149*
ITERATION_NUMBER, *7-151*
JSON_ARRAY, *7-153*
JSON_ARRAYAGG, *7-154*

SQL functions *(continued)*
JSON_DATAGUIDE, *7-156*
JSON_OBJECT, *7-157*
JSON_OBJECTAGG, *7-159*
JSON_QUERY, *7-161*
JSON_TABLE, *7-167*
JSON_VALUE, *7-176*
LAG, *7-181*
large object functions, *7-8*
LAST, *7-182*
LAST_DAY, *7-183*
LAST_VALUE, *7-184*
LEAD, *7-187*
LEAST, *7-188*
LENGTH, *7-189*
LENGTH2, *7-189*
LENGTH4, *7-189*
LENGTHB, *7-189*
LENGTHC, *7-189*
linear regression, *7-292*
LISTAGG, *7-190*
LN, *7-194*
LNNVL, *7-195*
LOCALTIMESTAMP, *7-196*
LOG, *7-197*
LOWER, *7-197*
LPAD, *7-198*
LTRIM, *7-199*
MAKE_REF, *7-200*
MAX, *7-200*
MEDIAN, *7-202*
MIN, *7-204*
MOD, *7-206*
model functions, *7-21*
MONTHS_BETWEEN, *7-207*
NANVL, *7-207*
NCHR, *7-208*
NEW_TIME, *7-209*
NEXT_DAY, *7-210*
NLS_CHARSET_DECL_LEN, *7-211*
NLS_CHARSET_ID, *7-211*
NLS_CHARSET_NAME, *7-212*
NLS_COLLATION_ID, *7-212*
NLS_COLLATION_NAME, *7-213*
NLS_INITCAP, *7-214*
NLS_LOWER, *7-216*
NLS_UPPER, *7-216*
NLSSORT, *7-217*
NTH_VALUE, *7-220*
NTILE, *7-222*
NULL-related functions, *7-11*
NULLIF, *7-223*
numeric functions, *7-4*
NUMTODSINTERVAL, *7-224*
NUMTOYMINTERVAL, *7-225*

SQL functions *(continued)*
NVL, *7-226*
NVL2, *7-227*
object reference functiions, *7-21*
OLAP functions, *7-21*
ORA_DM_PARTITION_NAME, *7-228*
ORA_DST_AFFECTED, *7-230*
ORA_DST_CONVERT, *7-230*
ORA_DST_ERROR, *7-231*
ORA_HASH, *7-232*
ORA_INVOKING_USER, *7-233*
ORA_INVOKING_USERID, *7-233*
PATH, *7-234*
PERCENT_RANK, *7-235*
PERCENTILE_CONT, *7-237*
PERCENTILE_DISC, *7-239*
POWER, *7-241*
POWERMULTISET, *7-242*
POWERMULTISET_BY_CARDINALITY, *7-243*
PREDICTION, *7-244*
PREDICTION_BOUNDS, *7-248*
PREDICTION_COST, *7-250*
PREDICTION_DETAILS, *7-254*
PREDICTION_PROBABILITY, *7-258*
PREDICTION_SET, *7-261*
PRESENTNNV, *7-265*
PRESENTV, *7-266*
PREVIOUS, *7-267*
RANK, *7-268*
RATIO_TO_REPORT, *7-270*
RAWTOHEX, *7-271*
RAWTONHEX, *7-272*
REF, *7-273*
REFTOHEX, *7-273*
REGEXP_COUNT, *7-274*
REGEXP_INSTR, *7-279*
REGEXP_REPLACE, *7-283*
REGEXP_SUBSTR, *7-288*
REGR_AVGX, *7-292*
REGR_AVGY, *7-292*
REGR_COUNT, *7-292*
REGR_INTERCEPT, *7-292*
REGR_R2, *7-292*
REGR_SLOPE, *7-292*
REGR_SXX, *7-292*
REGR_SXY, *7-292*
REGR_SYY, *7-292*
REMAINDER, *7-297*
REPLACE, *7-298*
ROUND (date), *7-299*
ROUND (number), *7-299*
ROW_NUMBER, *7-301*
ROWIDTOCHAR, *7-302*
ROWIDTONCHAR, *7-303*

SQL functions *(continued)*
RPAD, *7-304*
RTRIM, *7-305*
SCN_TO_TIMESTAMP, *7-306*
SESSIONTIMEZONE, *7-307*
SET, *7-308*
SIGN, *7-308*
SIN, *7-309*
single-row functions, *7-4*
SINH, *7-310*
SOUNDEX, *7-310*
SQRT, *7-311*
STANDARD_HASH, *7-312*
STATS_BINOMIAL_TEST, *7-313*
STATS_CROSSTAB, *7-314*
STATS_F_TEST, *7-316*
STATS_KS_TEST, *7-317*
STATS_MODE, *7-318*
STATS_MW_TEST, *7-319*
STATS_ONE_WAY_ANOVA, *7-321*
STATS_T_TEST_INDEP, *7-323*, *7-324*
STATS_T_TEST_INDEPU, *7-323*, *7-324*
STATS_T_TEST_ONE, *7-322*, *7-324*
STATS_T_TEST_PAIRED, *7-322*, *7-324*
STATS_WSR_TEST, *7-326*
STDDEV, *7-327*
STDDEV_POP, *7-328*
STDDEV_SAMP, *7-329*
SUBSTR, *7-331*
SUBSTR2, *7-331*
SUBSTR4, *7-331*
SUBSTRB, *7-331*
SUBSTRC, *7-331*
SUM, *7-332*
SYS_CONNECT_BY_PATH, *7-334*
SYS_CONTEXT, *7-335*
SYS_DBURIGEN, *7-342*
SYS_EXTRACT_UTC, *7-343*
SYS_GUID, *7-344*
SYS_OP_ZONE_ID, *7-345*
SYS_TYPEID, *7-346*
SYS_XMLAGG, *7-347*
SYS_XMLGEN, *7-348*
SYSDATE, *7-349*
SYSTIMESTAMP, *7-350*
t-test, *7-322*
TAN, *7-351*
TANH, *7-351*
TIMESTAMP_TO_SCN, *7-352*
TO_APPROX_COUNT_DISTINCT, *7-353*
TO_APPROX_PERCENTILE, *7-354*
TO_BINARY_DOUBLE, *7-355*
TO_BINARY_FLOAT, *7-357*
TO_BLOB (bfile), *7-358*
TO_BLOB (raw), *7-359*

SQL functions *(continued)*
TO_CHAR (bfile|blob), *7-359*
TO_CHAR (character), *7-360*
TO_CHAR (datetime), *7-361*
TO_CHAR (number), *7-363*
TO_CLOB (bfile|blob), *7-365*
TO_CLOB (character), *7-366*
TO_DATE, *7-367*
TO_DSINTERVAL, *7-369*
TO_LOB, *7-371*
TO_MULTI_BYTE, *7-372*
TO_NCHAR (character), *7-372*
TO_NCHAR (datetime), *7-373*
TO_NCHAR (number), *7-374*
TO_NCLOB, *7-375*
TO_NUMBER, *7-376*
TO_SINGLE_BYTE, *7-377*
TO_TIMESTAMP, *7-378*
TO_TIMESTAMP_TZ, *7-379*
TO_YMINTERVAL, *7-381*
TRANSLATE, *7-382*
TRANSLATE ... USING, *7-383*
TREAT, *7-385*
TRIM, *7-386*
TRUNC (date), *7-387*
TRUNC (number), *7-388*
TZ_OFFSET, *7-389*
UID, *7-390*
UNISTR, *7-390*
UPDATEXML, *7-391*
UPPER, *7-392*
USER, *7-393*
USERENV, *7-394*
VALIDATE_CONVERSION, *7-395*
VALUE, *7-398*
VAR_POP, *7-399*
VAR_SAMP, *7-400*
VARIANCE, *7-401*
VSIZE, *7-403*
WIDTH_BUCKET, *7-403*
XML functions, *7-9*
XMLAGG, *7-405*
XMLCAST, *7-406*
XMLCDATA, *7-407*
XMLCOLATTVAL, *7-408*
XMLCOMMENT, *7-409*
XMLCONCAT, *7-409*
XMLDIFF, *7-410*
XMLELEMENT, *7-412*
XMLEXISTS, *7-415*
XMLFOREST, *7-415*
XMLISVALID, *7-416*
XMLPARSE, *7-417*
XMLPATCH, *7-418*
XMLPI, *7-419*

SQL functions *(continued)*
XMLQUERY, *7-420*
XMLROOT, *7-421*
XMLSEQUENCE, *7-422*
XMLSERIALIZE, *7-424*
XMLTABLE, *7-425*
XMLTRANSFORM, *7-428*
SQL statements
ALTER FLASHBACK ARCHIVE, *10-130*
auditing
by access, *12-229*
by session, *12-229*
stopping, *19-11*
successful, *12-230*
CREATE FLASHBACK ARCHIVE, *13-96*
DDL, *10-2*
determining the execution plan for, *18-16*
DML, *10-3*
DROP FLASHBACK ARCHIVE, *16-9*
organization of, *10-4*
rolling back, *19-35*
session control, *10-4*
space allocation, resumable, *11-91*
storage in the result cache, *15-115*
suspending and completing, *11-91*
system control, *10-4*
tracking the occurrence in a session, *12-222*
transaction control, *10-3*
type of, *10-1*
undoing, *19-35*
SQL translation profiles
auditing, *12-228*
granting object privileges for, *18-56*, *18-57*
granting system privileges for, *18-48*
SQL*Loader inserts, logging, *10-135*
SQL/DS data types, *2-34*
restrictions on, *2-35*
SQL/JSON
conditions, *6-23*
SQRT function, *7-311*
staging log, *14-38*
standalone procedures
dropping, *17-13*
standard SQL, *C-1*
Oracle extensions to, *C-33*
STANDARD_HASH function, *7-312*
standby database
synchronizing with primary database, *11-92*
standby databases
activating, *10-76*
and Data Guard, *10-81*
committing to primary status, *10-79*
controlling use, *10-87*
converting to physical standby, *10-81*
designing media recovery, *10-56*

standby databases *(continued)*
mounting, *10-54*
recovering, *10-40*
STAR_TRANSFORMATION hint, *2-133*
START LOGICAL STANDBY APPLY clause
of ALTER DATABASE, *10-81*
START WITH clause
of ALTER MATERIALIZED VIEW ...
REFRESH, *11-30*
of queries and subqueries, *19-39*
of SELECT and subqueries, *19-47*
START WITH parameter
of CREATE SEQUENCE, *15-5*
startup_clauses
of ALTER DATABASE, *10-41*
STATEMENT_QUEUING hint, *2-134*
statistics
collection during index rebuild, *10-135*
deleting from the data dictionary, *12-216*
forcing disassociation, *15-205*
gathering for bulk loads, *2-102*, *2-112*
on index usage, *10-152*
on scalar object attributes
collecting, *12-209*
on schema objects
collecting, *12-209*
deleting, *12-209*
user-defined
dropping, *16-12*, *16-14*, *17-10*, *18-1*,
18-10
statistics types
associating
with columns, *12-218*
with domain indexes, *12-218*
with functions, *12-218*
with indextypes, *12-218*
with object types, *12-218*
with packages, *12-218*
disassociating
from columns, *15-204*
from domain indexes, *15-204*
from functions, *15-204*
from indextypes, *15-204*
from object types, *15-204*
from packages, *15-204*
STATS_BINOMIAL_TEST function, *7-313*
STATS_CROSSTAB function, *7-314*
STATS_F_TEST function, *7-316*
STATS_KS_TEST function, *7-317*
STATS_MODE function, *7-318*
STATS_MW_TEST function, *7-319*
STATS_ONE_WAY_ANOVA function, *7-321*
STATS_T_TEST_INDEP function, *7-323*, *7-324*
STATS_T_TEST_INDEPU function, *7-323*, *7-324*
STATS_T_TEST_ONE function, *7-322*, *7-324*
STATS_T_TEST_PAIRED function, *7-322*, *7-324*
STATS_WSR_TEST function, *7-326*
STDDEV function, *7-327*
STDDEV_POP function, *7-328*
STDDEV_SAMP function, *7-329*
STOP LOGICAL STANDBY clause
of ALTER DATABASE, *10-81*
STORAGE clause
of ALTER CLUSTER, *10-36*
of ALTER INDEX, *10-135*
of ALTER MATERIALIZED VIEW LOG, *11-37*
of CREATE MATERIALIZED VIEW LOG. *See*
CREATE TABLE, *14-38*
of CREATE TABLE, *8-47*
storage parameters
resetting, *19-132*, *19-134*
Storage Snapshot Optimization, *10-58*
STORE IN clause
of ALTER TABLE, *12-92*, *15-104*
stored functions, *13-99*
string literals
See text literals.
strings, *2-56*
converting to ASCII values, *7-42*
converting to unicode, *7-76*
See also text literals.
Structured Query Language (SQL),
description, *1-1*
functions, *7-2*
keywords, *A-2*
Oracle Tools support of, *1-3*
parameters, *A-2*
standards, *1-1*, *C-1*
statements
determining the cost of, *18-16*
syntax, *10-4*, *A-1*
structures
locking, *B-4*
subexpressions
of regular expressions, *7-280*, *7-289*
SUBMULTISET condition, *6-14*
SUBPARTITION BY HASH clause
of CREATE TABLE, *15-17*, *15-43*
SUBPARTITION BY LIST clause
of CREATE TABLE, *15-108*
SUBPARTITION clause
of ANALYZE, *12-212*
of DELETE, *15-199*
of INSERT, *18-68*
of LOCK TABLE, *18-80*
of UPDATE, *19-142*
subpartition template
creating, *12-29*
replacing, *12-29*

subpartition-extended table names
in DML statements, 2-150
restrictions on, 2-149
syntax, 2-149
subpartitions
adding, 12-29
adding rows to, 18-61
allocating extents for, 12-85
coalescing, 12-122
converting into nonpartitioned tables, 12-29
creating, 15-43
creating a template for, 12-29, 15-17
deallocating unused space from, 12-86
exchanging with tables, 12-69
hash, 15-17
inserting rows into, 18-68
list, 15-108
list, adding, 12-29
locking, 18-80
logging insert operations, 12-80
moving to a different segment, 12-29
physical attributes
changing, 12-79
removing rows from, 12-29, 15-199
renaming, 12-29
revising values in, 19-142
specifying, 15-105
template, creating, 15-17
template, dropping, 12-29
template, replacing, 12-29
subqueries, 9-1, 9-16, 19-39, 19-40
containing subqueries, 9-16
correlated, 9-16
defined, 9-1
extended subquery unnesting, 9-18
inline views, 9-16
nested, 9-16
of past data, 19-39
scalar, 5-31
to insert table data, 15-121
unnesting, 9-17
using in place of expressions, 5-31
SUBSTR function, 7-331
SUBSTR2 function, 7-331
SUBSTR4 function, 7-331
SUBSTRB function, 7-331
SUBSTRC function, 7-331
subtotal values
deriving, 19-76
subtypes
dropping safely, 18-12
SUM function, 7-332
supplemental logging
identification key (full), 10-73
minimal, 10-73
SUSPEND clause
of ALTER SYSTEM, 12-14
sustained standby recovery mode, 10-60
SWITCH LOGFILE clause
of ALTER SYSTEM, 12-13
SYNC WITH PRIMARY
clause of ALTER SESSION, 11-92
synchronous refresh, 14-38
synonyms
changing the definition of, 17-19
creating, 15-12
granting system privileges for, 18-30
local, 15-12
private, dropping, 17-20
public, 15-13
dropping, 17-20
remote, 15-12
removing from the database, 17-19
renaming, 19-22, 19-23
synonyms for, 15-12
syntax diagrams, A-1
loops, A-4
multipart diagrams, A-4
SYS user
assigning password for, 13-56
SYS_CONNECT_BY_PATH function, 7-334
SYS_CONTEXT function, 7-335
SYS_DBURIGEN function, 7-342
SYS_EXTRACT_UTC function, 7-343
SYS_GUID function, 7-344
SYS_NC_ROWINFO$ column, 15-17, 15-179
SYS_OP_ZONE_ID function, 7-345
SYS_SESSION_ROLES namespace, 7-335
SYS_TYPEID function, 7-346
SYS_XMLAGG function, 7-347
SYS_XMLGEN function, 7-348
SYSAUX clause
of CREATE DATABASE, 13-62
SYSAUX tablespace
creating, 13-62
SYSDATE function, 7-349
system change numbers
obtaining, 3-9
system control statements, 10-4
PL/SQL support of, 10-4
system global area
flushing, 12-10, 12-11
updating, 12-10
system parameters
GLOBAL_TOPIC_ENABLED, 12-24
system partitioning, 15-110
system privileges
ADMINISTER ANY SQL TUNING SET, 18-41
ADMINISTER KEY MANAGEMENT, 18-44

system privileges *(continued)*
ADMINISTER SQL MANAGEMENT OBJECT, *18-41*
ADMINISTER SQL TUNING SET, *18-41*
ALTER ANY SQL PROFILE, *18-41*
ALTER DATABASE LINK, *18-42*
ALTER PUBLIC DATABASE LINK, *18-42*
BECOME USER, *18-51*
CHANGE NOTIFICATION, *18-51*
CREATE ANY SQL PROFILE, *18-41*
CREATE PLUGGABLE DATABASE, *18-47*
DROP ANY SQL PROFILE, *18-41*
for job scheduler tasks, *18-30*
for the Advisor framework, *18-41*
granting, *14-106*, *18-30*
to a role, *18-34*
to a user, *18-34*
to PUBLIC, *18-34*
list of, *12-223*, *18-41*
MERGE ANY VIEW, *18-50*
READ ANY TABLE, *18-49*
revoking, *19-24*
from a role, *19-27*
from a user, *19-27*
from PUBLIC, *19-27*
SET CONTAINER, *18-47*
SYSTEM tablespace
locally managed, *13-61*
SYSTEM user
assigning password for, *13-56*
SYSTIMESTAMP function, *7-350*

T

TABLE clause
of ANALYZE, *12-211*
of INSERT, *18-61*
of SELECT, *19-39*
of TRUNCATE, *19-135*
of UPDATE, *19-138*
TABLE collection expression, *19-39*
table compression, *11-25*, *12-80*, *14-20*, *15-69*
Advanced Row Compression, *15-70*
basic, *15-70*
during bulk load operations, *15-70*
for archiving data, *15-70*
Hybrid Columnar, *15-70*
table locks,
and queries, *18-80*
disabling, *12-153*
duration of, *18-80*
enabling, *12-153*
EXCLUSIVE, *18-80*, *18-83*
modes of, *18-80*
on partitions, *18-80*
table locks *(continued)*
on remote database, *18-80*
on subpartitions, *18-80*
ROW EXCLUSIVE, *18-80*, *18-83*
ROW SHARE, *18-80*, *18-83*
SHARE, *18-80*
SHARE ROW EXCLUSIVE, *18-83*
SHARE UPDATE, *18-83*
table partitions
compression of, *12-80*, *15-69*
table REF constraints, *8-3*
of CREATE TABLE, *15-17*
tables
adding a constraint to, *12-113*
adding rows to, *18-61*
aliases, *2-151*
in DELETE, *15-195*
allocating extents for, *12-85*
assigning to a cluster, *15-83*
changing degree of parallelism on, *12-29*
changing existing values in, *19-138*
collecting statistics on, *12-211*
comments on, *12-249*
compression of, *12-80*, *15-69*
creating, *15-17*
multiple, *14-113*
creating comments about, *12-247*
data stored outside database, *15-17*
deallocating unused space from, *12-86*
default physical attributes
changing, *12-79*
degree of parallelism
specifying, *15-17*
disassociating statistics types from, *18-1*
dropping
along with cluster, *15-210*
along with owner, *18-14*
indexes of, *18-1*
partitions of, *18-1*
enabling tracking, *15-119*
external, *15-77*
creating, *15-17*
restrictions on, *15-81*
externally organized, *15-77*
flashing back to an earlier version, *18-23*
granting system privileges for, *18-30*
heap organized, *15-77*
index-organized, *15-77*
overflow segment for, *15-79*
space in index block, *15-78*
inserting rows with a subquery, *15-121*
inserting using the direct-path method, *18-61*
joining in a query, *19-39*
LOB storage of, *8-47*
locking, *18-80*

tables *(continued)*
logging
insert operations, *12-80*
table creation, *15-17*
migrated and chained rows in, *12-216*
moving, *12-75*
moving to a new segment, *12-29*
moving, index-organized, *12-149*
nested
storage characteristics, *15-17*
object
creating, *15-19*
querying, *15-17*
of XMLType, creating, *15-17*
organization, defining, *15-77*
parallel creation of, *15-17*
parallelism
setting default degree, *15-17*
partition attributes of, *12-29*
partitioning, *2-149*, *15-17*
allowing rows to move between partitions, *12-29*
default attributes of, *12-29*
physical attributes
changing, *12-79*
purging from the recycle bin, *19-20*
read-only mode, *12-89*
read/write mode, *12-89*
reference-partitioned, *12-116*, *12-143*, *15-109*
relational
creating, *15-18*
remote, accessing, *13-69*
removing from the database, *18-1*
removing rows from, *15-195*
renaming, *12-88*, *19-22*
restricting
records in a block, *12-87*
retrieving data from, *19-39*
saving blocks in a cache, *15-114*
SQL examples, *15-17*
storage attributes
defining, *15-17*
storage characteristics
defining, *8-47*
storage properties of, *15-17*, *15-83*
subpartition attributes of, *12-29*
synonyms for, *15-12*
tablespace for
defining, *15-17*, *15-69*
temporary
duration of data, *15-66*
session-specific, *15-49*
transaction specific, *15-49*
unclustering, *15-209*

tables *(continued)*
updating through views, *15-189*
validating structure, *12-214*
XMLType, querying, *15-17*
TABLESPACE clause
of ALTER INDEX ... REBUILD, *10-147*
of CREATE CLUSTER, *13-37*
of CREATE INDEX, *13-123*
of CREATE MATERIALIZED VIEW, *14-19*
of CREATE MATERIALIZED VIEW LOG, *14-42*
of CREATE TABLE, *15-69*
tablespaces
allocating space for users, *15-175*
allowing write operations on, *12-184*
automatic segment-space management, *15-155*
backing up data files, *12-179*
bigfile, *15-147*
database default, *13-58*
default temporary, *13-63*
resizing, *12-177*
undo, *13-52*
bringing online, *12-183*, *15-153*
coalescing free extents, *12-178*
converting
from permanent to temporary, *12-184*
from temporary to permanent, *12-184*
creating, *15-141*
data files
adding, *12-171*
renaming, *12-171*
default, *10-83*
specifying for a user, *12-199*
default permanent, *13-63*
default temporary, *10-83*
learning name of, *10-83*
designing media recovery, *10-56*
dropping contents, *18-7*
encrypting, *8-51*
ending online backup, *12-180*
extent size, *15-150*
granting system privileges for, *18-30*
in FLASHBACK mode, *12-171*, *15-141*
in FORCE LOGGING mode, *12-182*, *15-151*
locally managed, *8-54*
altering, *12-176*
logging attribute, *12-171*, *15-141*
managing extents of, *15-141*
read only, *12-184*
reconstructing lost or damaged, *10-56*, *10-64*
recovering, *10-56*, *10-58*
removing from the database, *18-5*
renaming, *12-178*
size of free extents in, *12-177*

tablespaces *(continued)*
smallfile, 15-147
database default, 13-58
default temporary, 13-63
undo, 13-52
specifying
data files for, 15-148
for a table, 15-17
for a user, 15-174
for index rebuild, 12-149
taking offline, 12-183, 15-153
temp files
adding, 12-171
temporary
creating, 15-158
defining for the database, 13-54
shrinking, 12-178
specifying for a user, 12-199, 15-175
undo
altering, 12-176
creating, 13-52, 15-157
dropping, 18-6
TAN function, 7-351
TANH function, 7-351
TDE
See Transparent Data Encryption
temp files
bringing online, 10-67
defining for a tablespace, 15-142, 15-146, 15-147
defining for the database, 13-55
disabling autoextend, 10-67
dropping, 10-67, 12-181
enabling autoextend, 8-32, 10-67
extending automatically, 8-32
renaming, 10-64
resizing, 10-67
reusing, 8-32
shrinking, 12-181
size of, 8-32
specifying, 8-32
taking offline, 10-67
TEMPFILE clause
of ALTER DATABASE, 10-44, 10-67
TEMPORARY clause
of ALTER TABLESPACE, 12-184
of CREATE TABLESPACE, 15-158
temporary tables
creating, 15-17, 15-49
session-specific, 15-49
transaction-specific, 15-49
TEMPORARY TABLESPACE clause
of ALTER USER, 12-199
of ALTER USER. *See* CREATE USER, 12-197
TEMPORARY TABLESPACE clause *(continued)*
of CREATE USER, 15-175
temporary tablespace groups
reassigning for a user, 12-199
specifying for a user, 15-175
temporary tablespaces
creating, 15-158
default, 10-83
specifying extent management during database creation, 13-54
specifying for a user, 12-199, 15-175
TEST clause
of ALTER DATABASE ... RECOVER, 10-59
testing for a set, 6-12
text, 2-56
date and number formats, 2-67
literals
in SQL syntax, 2-56
properties of CHAR and VARCHAR2 data types, 2-57
syntax of, 2-56
text literals
conversion to database character set, 2-56
TH datetime format element suffix, 2-79
throughput
optimizing, 2-94
THSP datetime format element suffix, 2-79
TIME data type
DB2, 2-35
SQL/DS, 2-35
time format models
short, 2-70, 2-75
time zone
changing time zone data file, 7-230
converting data to particular, 5-23
determining for session, 7-307
formatting, 2-75
setting for the database, 13-52
TIME_ZONE session parameter, 11-96
timestamp
converting to local time zone, 5-23
TIMESTAMP data type, 2-22
DB2, 2-35
SQL/DS, 2-35
TIMESTAMP WITH LOCAL TIME ZONE data type, 2-23
TIMESTAMP WITH TIME ZONE data type, 2-22
TIMESTAMP_TO_SCN function, 7-352
TO SAVEPOINT clause
of ROLLBACK, 19-36
TO_APPROX_COUNT_DISTINCT function, 7-353
TO_APPROX_PERCENTILE function, 7-354
TO_BINARY_DOUBLE function, 7-355
TO_BINARY_FLOAT function, 7-357

TO_BLOB (bfile) function, 7-358
TO_BLOB (raw) function, 7-359
TO_CHAR (bfile|blob) function, 7-359
TO_CHAR (character) function, 7-360
TO_CHAR (datetime) function, 7-361
  format models, 2-71, 2-79
TO_CHAR (number) function, 7-363
  format models, 2-68, 2-79
TO_CLOB (bfile|blob) function, 7-365
TO_CLOB (character) function, 7-366
TO_DATE function, 7-367
  format models, 2-71, 2-78, 2-79
TO_DSINTERVAL function, 7-369
TO_LOB function, 7-371
TO_MULTI_BYTE function, 7-372
TO_NCHAR (character) function, 7-372
TO_NCHAR (datetime) function, 7-373
TO_NCHAR (number) function, 7-374
TO_NCLOB function, 7-375
TO_NUMBER function, 7-376
  format models, 2-68
TO_SINGLE_BYTE function, 7-377
TO_TIMESTAMP function, 7-378
TO_TIMESTAMP_TZ function, 7-379
TO_YMINTERVAL function, 7-381
top-level SQL statements, 12-226
top-N reporting, 3-11, 7-107, 7-269, 7-301, 19-39
tracking
  enabling for a table, 12-88, 15-119
transaction control statements, 10-3
  PL/SQL support of, 10-3
transactions
  allowing to complete, 12-12
  assigning
    rollback segment to, 19-129
  automatically committing, 13-2
  changes, making permanent, 13-1
  commenting on, 13-3
  distributed, forcing, 11-89
  ending, 13-1
  implicit commit of, 10-2–10-4
  in-doubt
    committing, 13-2
    forcing, 13-5
    resolving, 19-131
  isolation level, 19-129
  locks, releasing, 13-1
  naming, 19-131
  read-only, 19-129
  read/write, 19-129
  rolling back, 12-12, 19-35
    to a savepoint, 19-36
  savepoints for, 19-38
TRANSLATE ... USING function, 7-383
TRANSLATE function, 7-382
TRANSLATE SQL object privilege
  on a user, 18-57
Transparent Data Encryption, 15-57
  key management, 10-5
  master key, 12-18
TREAT function, 7-385
triggers,
  compiling, 12-191
  creating, 15-164
  database
    altering, 12-191
    dropping, 18-9, 18-14
  disabling, 12-154, 12-191
  enabling, 12-153, 12-191, 12-193, 15-164
  granting system privileges for, 18-30
  INSTEAD OF
    dropping, 15-182
  re-creating, 15-165
  removing from the database, 18-9
  renaming, 12-193
TRIM function, 7-386
TRUNC (date) function, 7-387
  format models, 7-429
TRUNC (number) function, 7-388
TRUNCATE PARTITION clause
  of ALTER TABLE, 12-29
TRUNCATE SUBPARTITION clause
  of ALTER TABLE, 12-29
TRUNCATE_CLUSTER statement, 19-132
TRUNCATE_TABLE statement, 19-134
type constructor expressions, 5-31
types
  See object types or data types
TYPES clause
  of ASSOCIATE STATISTICS, 12-218, 12-220
  of DISASSOCIATE STATISTICS, 15-204
TZ_OFFSET function, 7-389

U

UID function, 7-390
unary operators, 4-1
UNDER object privilege
  on a view, 18-58
UNDER_PATH condition, 6-23
undo
  rollback, 11-83, 13-52
  system managed, 11-83, 13-52
UNDO tablespace clause
  of CREATE DATABASE, 13-52
  of CREATE TABLESPACE, 15-157
undo tablespaces
  creating, 13-52, 15-157
  dropping, 18-6
  modifying, 12-176

undo tablespaces *(continued)*
preserving unexpired data, *12-171*, *15-141*
UNDO_RETENTION initialization parameter
setting with ALTER SYSTEM, *18-23*
unified audit policies
comments on, *12-248*
creating, *13-23*
dropping, *15-207*
modifying, *10-31*
unified auditing
ALTER AUDIT POLICY statement, *10-31*
AUDIT statement, *12-238*
CREATE AUDIT POLICY statement, *13-23*
DROP AUDIT POLICY statement, *15-207*
NOAUDIT statement, *19-15*
UNIFORM clause
of CREATE TABLESPACE, *15-141*
UNION ALL set operator, *4-6*
UNION set operator, *4-6*
UNIQUE clause
of CREATE INDEX, *13-115*
of CREATE TABLE, *15-17*
of SELECT, *19-59*
unique constraints
conditional, *13-138*
enabling, *15-118*
index on, *15-17*
unique elements of, *7-308*
unique indexes, *13-115*
unique queries, *19-59*
UNISTR function, *7-390*
universal rowids
See urowids
UNNEST hint, *2-134*
unnesting collections, *19-39*
examples, *19-120*
unnesting subqueries, *9-17*
unpivot operations, *19-69*
examples, *19-112*
syntax, *19-44*
UNQUIESCE clause
of ALTER SYSTEM, *12-14*
UNRECOVERABLE, *10-145*, *15-77*
See also NOLOGGING clause
unsorted indexes, *13-125*
UNUSABLE clause
of ALTER INDEX, *10-151*
UNUSABLE LOCAL INDEXES clause
of ALTER MATERIALIZED VIEW, *11-26*
of ALTER TABLE, *12-144*
UPDATE BLOCK REFERENCES clause
of ALTER INDEX, *10-152*, *10-154*
UPDATE GLOBAL INDEXES clause
of ALTER TABLE, *12-29*
update operations
collecting supplemental log data for, *10-73*
UPDATE SET clause
of MERGE, *19-2*
UPDATE statement, *19-138*
updates
and simultaneous insert, *19-1*
using MERGE, *19-1*, *19-2*
UPDATEXML function, *7-391*
UPGRADE clause
of ALTER DATABASE, *10-55*
of ALTER TABLE, *12-29*
UPPER function, *7-392*
URLs
generating, *7-342*
UROWID data type, *2-34*
urowids, *2-34*
and foreign tables, *2-34*
and index-organized tables, *2-34*
description of, *2-34*
USABLE clause
of ALTER INDEX, *10-151*
USE object privilege
on a SQL translation profile, *18-56*
USE_BAND hint, *2-135*
USE_CONCAT hint, *2-135*
USE_CUBE hint, *2-136*
USE_HASH hint, *2-136*
USE_MERGE hint, *2-136*
USE_NL hint, *2-137*
USE_NL_WITH_INDEX hint, *2-137*
USE_PRIVATE_OUTLINES session parameter, *11-96*
USE_STORED_OUTLINES session parameter, *11-96*, *12-24*
USER function, *7-393*
user groups
adding or dropping a member, *10-121*
adding to a disk group, *10-121*
dropping from a disk group, *10-121*
USER SYS clause
of CREATE DATABASE, *13-56*
USER SYSTEM clause
of CREATE DATABASE, *13-56*
USER_COL_COMMENTS data dictionary view, *12-248*
USER_INDEXTYPE_COMMENTS data dictionary view, *12-249*
USER_MVIEW_COMMENTS data dictionary view, *12-249*
USER_OPERATOR_COMMENTS data dictionary view, *12-249*
USER_TAB_COMMENTS data dictionary view, *12-248*

user-defined functions, 7-431
name precedence of, 7-432
naming conventions, 7-433
user-defined operators, 4-10
user-defined statistics
dropping, 16-12, 16-14, 17-10, 18-1, 18-10
user-defined types, 2-36
USERENV function, 7-394
USERENV namespace, 7-335
users
allocating space for, 15-175
and database links, 13-72
assigning
default roles, 12-200
profiles, 15-176
authenticating, 12-204
authenticating to a remote server, 13-69
changing authentication, 12-204
creating, 15-169
default tablespaces for, 12-199, 15-174
denying access to tables and views, 18-80
external, 14-108, 15-173
global, 14-108, 15-173
granting system privileges for, 18-30
local, 14-108, 15-172
locking accounts, 15-176
operating system
adding to a disk group, 10-121
dropping from a disk group, 10-122
password expiration of, 15-176
removing from the database, 18-13
SQL examples, 15-169
temporary tablespaces for, 12-199, 15-175
USING BFILE clause
of CREATE JAVA, 13-152
USING BLOB clause
of CREATE JAVA, 13-152
USING clause
of ALTER INDEXTYPE, 10-160
of ASSOCIATE STATISTICS, 12-218, 12-219
of CREATE DATABASE LINK, 13-69
of CREATE INDEXTYPE, 13-144
of CREATE PLUGGABLE DATABASE, 14-89
USING CLOB clause
of CREATE JAVA, 13-152
USING INDEX clause
of ALTER MATERIALIZED VIEW, 11-27
of ALTER TABLE, 12-76
of constraints, 8-3
of CREATE MATERIALIZED VIEW, 14-23
of CREATE TABLE, 15-17
USING NO INDEX clause
of CREATE MATERIALIZED VIEW, 14-23
USING ROLLBACK SEGMENT clause
of ALTER MATERIALIZED VIEW ...
REFRESH, 11-30
of CREATE MATERIALIZED VIEW, 14-28
UTC
extracting from a datetime value, 7-343
UTC offset
replacing with time zone region name, 2-63
UTLCHN.SQL script, 12-216
UTLEXPT1.SQL script, 12-142
UTLXPLAN.SQL script, 18-16

V

VALIDATE clause
of DROP TYPE, 18-12
VALIDATE REF UPDATE clause
of ANALYZE, 12-214
VALIDATE STRUCTURE clause
of ANALYZE, 12-214
VALIDATE_CONVERSION function, 7-395
validation
of clusters, 12-214
of database objects
offline, 12-215
of database objects, online, 12-215
of indexes, 12-214
of tables, 12-214
VALUE function, 7-398
VALUES clause
of CREATE INDEX, 13-128
of INSERT, 18-61
VALUES LESS THAN clause
of CREATE TABLE, 15-17
VAR_POP function, 7-399
VAR_SAMP function, 7-400
VARCHAR data type, 2-12
VARCHAR2 data type, 2-11
converting to NUMBER, 2-68
VARGRAPHIC data type
DB2, 2-35
SQL/DS, 2-35
VARIANCE function, 7-401
VARRAY clause
of ALTER TABLE, 12-50, 12-51
VARRAY column properties
of ALTER TABLE, 12-29, 12-50, 12-51
of CREATE MATERIALIZED VIEW, 14-13, 14-14
of CREATE TABLE, 15-17, 15-29
varrays, 2-37
changing returned value, 12-29
compared with nested tables, 2-48
comparison rules, 2-48
creating, 15-166

varrays *(continued)*
dropping the body of, *18-12*
dropping the specification of, *18-10*
modifying column properties, *12-54*
storage characteristics, *12-29*, *15-17*
storing out of line, *2-37*
varying arrays
See varrays
version queries
pseudocolumns for, *3-6*
view constraints, *8-3*, *15-185*
and materialized views, *8-3*
dropping, *18-16*
modifying, *12-208*
views
base tables
adding rows, *18-61*
changing
definition, *18-15*
values in base tables, *19-138*
creating
before base tables, *15-182*
comments about, *12-247*
multiple, *14-113*
definer's rights, *15-188*
defining, *15-179*
dropping constraints on, *12-208*
editioning, *15-182*
granting system privileges for, *18-30*
invoker's rights, *15-188*
modifying constraints on, *12-208*
object, creating, *15-186*
re-creating, *15-182*
recompiling, *12-206*
remote, accessing, *13-69*
removing
from the database, *18-15*
rows from the base table of, *15-195*
renaming, *19-22*
retrieving data from, *19-39*
subquery of, *15-189*
restricting, *15-179*
synonyms for, *15-12*
updatable, *15-189*
with joins
and key-preserved tables, *15-190*
making updatable, *15-190*
XMLType, *15-179*
XMLType, creating, *15-194*
XMLType, querying, *15-179*
virtual columns
adding to a table, *12-96*
creating, *15-17*
modifying, *12-96*
VSIZE function, *7-403*

W

WHENEVER SUCCESSFUL clause
of AUDIT sql_statements, *12-230*
WHERE clause
of DELETE, *15-201*
of queries and subqueries, *19-39*
of SELECT, *9-4*
of UPDATE, *19-138*
WIDTH_BUCKET function, *7-403*
WITH ... AS clause
of SELECT, *19-39*
WITH ADMIN OPTION clause
of GRANT, *18-35*
WITH CHECK OPTION clause
of CREATE VIEW, *15-179*, *15-181*
of DELETE, *15-195*
of INSERT, *18-61*
of SELECT, *19-45*
of UPDATE, *19-138*
WITH clause
of SELECT, *19-56*
WITH GRANT OPTION clause
of GRANT, *18-39*
WITH HIERARCHY OPTION
of GRANT, *18-39*
WITH INDEX CONTEXT clause
of CREATE OPERATOR, *14-60*
WITH OBJECT ID clause
of CREATE MATERIALIZED VIEW LOG, *14-43*
WITH PRIMARY KEY clause
of ALTER MATERIALIZED VIEW, *11-30*
of CREATE MATERIALIZED VIEW ... REFRESH, *14-27*
of CREATE MATERIALIZED VIEW LOG, *14-43*
WITH READ ONLY clause
of CREATE VIEW, *15-179*, *15-181*
of DELETE, *15-195*
of INSERT, *18-61*
of SELECT, *19-45*
of UPDATE, *19-138*
WITH ROWID clause
of column ref constraints, *8-3*
of CREATE MATERIALIZED VIEW ... REFRESH, *14-4*
of CREATE MATERIALIZED VIEW LOG, *14-43*
WITH SEQUENCE clause
of CREATE MATERIALIZED VIEW LOG, *14-43*
WRITE clause
of COMMIT, *13-4*

X

XML
- conditions, 6-22
- data
  - storage of, 15-17
- database repository
  - SQL access to, 6-22, 6-23
- documents
  - producing from XML fragments, 7-347
  - retrieving from the database, 7-342
- examples, F-8
- format models, 2-83
- fragments, 7-116
- functions, 7-9

XMLAGG function, 7-405
XMLCAST function, 7-406
XMLCDATA function, 7-407
XMLCOLATTVAL function, 7-408
XMLCOMMENT function, 7-409
XMLCONCAT function, 7-409
XMLDATA pseudocolumn, 3-12
XMLDIFF function, 7-410
XMLELEMENT function, 7-412
XMLEXISTS function, 7-415
XMLFOREST function, 7-415
XMLGenFormatType object, 2-83
XMLIndex
- creating, 13-133
- modifying, 10-148

XMLISVALID function, 7-416
XMLPARSE function, 7-417
XMLPATCH function, 7-418
XMLPI function, 7-419
XMLQUERY function, 7-420
XMLROOT function, 7-421
XMLSchemas
- adding to a table, 15-17
- single and multiple, 15-17

XMLSEQUENCE function, 7-422
XMLSERIALIZE function, 7-424
XMLTABLE function, 7-425
XMLTRANSFORM function, 7-428
XMLType columns
- properties of, 12-29, 15-17
- storage of, 15-17
- storing in binary XML format, 15-17

XMLType storage clause
- of CREATE TABLE, 15-17

XMLType tables
- creating, 15-17, 15-132
- creating index on, 13-136

XMLType views, 15-179
- querying, 15-179

Z

zone maps
- creating, 14-49
- modifying, 11-44
- removing from the database, 17-7