Oracle8™

SQL Reference

Release 8.0

December 1997

Part No. A58225-01

ORACLE®

Enabling the Information Age™

Oracle8™ Server SQL Reference

Part No. A58225-01

Release 8.0

Copyright © 1997 Oracle Corporation. All Rights Reserved.

Primary Author: Diana Lorentz

Contributors: Steve Bobrowski, Robert Jenkins, Susan Kotsovolos, Andre Kruglikov, Vishu Krishna, Muralidhar Krishnaprasa, Michael Kung, Paul Lane, Nina Lewis, Lefty Leverenz, Phil Locke, William Maimone, Mohammad Monajjemi, Rita Moran, Thomas Portfolio, Valarie Moore, Denis Raphaely, Richard Sarwal, Rick Wong

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

This Program contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright patent and other intellectual property law. Reverse engineering of the software is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

If this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are 'commercial computer software' and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are 'restricted computer software' and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52..227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle Call Interface, Oracle7, Oracle7 Server, Oracle8, Oracle Forms, Oracle Parallel Server, Net8, PL/SQL, SQL*Net, Pro*C/C++, and SQL*Module are trademarks of Oracle Corporation, Redwood Shores, California.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

Send Us Your Comments xv

Preface xvii

1 Introduction

History of SQL 1- 1
SQL Standards 1- 2
Embedded SQL 1- 3
Lexical Conventions 1- 4
Tools Support 1- 5

2 Elements of Oracle8 SQL

Literals 2-2
Text 2-2
Integer 2-3
Number 2-4
Datatypes 2-5
Nulls 2-31
Pseudocolumns 2-32
Comments 2-38
Database Objects 2-44
Schema Object Names and Qualifiers 2-47
Referring to Schema Objects and Parts 2-51

3 Operators, Functions, Expressions, Conditions

Operators 3-1
SQL Functions 3-16
User Functions 3-60
Format Models 3-63
Expressions 3-78
Conditions 3-90

4 Commands

Summary of SQL Commands ..... 4 - 2
Data Definition Language (DDL) Commands ..... 4 - 2
Data Manipulation Language (DML) Commands ..... 4 - 7
Transaction Control Commands ..... 4 - 8
Session Control Commands ..... 4 - 8
System Control Command ..... 4 - 9
Embedded SQL Commands ..... 4 - 10
ALTER CLUSTER ..... 4 - 11
Altering Clusters ..... 4 - 13
ALTER DATABASE ..... 4 - 15
Examples ..... 4 - 23
ALTER FUNCTION ..... 4 - 26
Recompiling Standalone Functions ..... 4 - 26
ALTER INDEX ..... 4 - 28
Examples ..... 4 - 35
ALTER PACKAGE ..... 4 - 38
Recompiling Stored Packages ..... 4 - 38
ALTER PROCEDURE ..... 4 - 41
Recompiling Stored Procedures ..... 4 - 41
ALTER PROFILE ..... 4 - 43
Using Password History ..... 4 - 45
Examples ..... 4 - 45
ALTER RESOURCE COST ..... 4 - 48
Altering Resource Costs ..... 4 - 48
ALTER ROLE ..... 4 - 51
Changing Authorizations ..... 4 - 51

ALTER ROLLBACK SEGMENT ..... 4 - 53
Altering Rollback Segments ..... 4 - 54
ALTER SEQUENCE ..... 4 - 56
Examples ..... 4 - 57
ALTER SESSION ..... 4 - 58
Enabling and Disabling the SQL Trace Facility ..... 4 - 67
Using NLS Parameters ..... 4 - 67
Changing the Optimization Approach and Mode ..... 4 - 71
FIPS Flagging ..... 4 - 71
Caching Session Cursors ..... 4 - 71
Accessing the Database as if Connected to Another Instance in a Parallel Server .. 4 - 72
Closing Database Links ..... 4 - 72
Forcing In-Doubt Distributed Transactions ..... 4 - 73
Transaction Control in Procedures and Stored Functions ..... 4 - 74
Parallel DML ..... 4 - 74
ALTER SNAPSHOT ..... 4 - 76
Examples ..... 4 - 81
Specifying Rollback Segments ..... 4 - 82
Primary Key Snapshots ..... 4 - 82
Partitioned Snapshots ..... 4 - 83
ALTER SNAPSHOT LOG ..... 4 - 84
Modifying Physical Attributes ..... 4 - 86
Adding Primary Key, ROWID, and Filter Columns ..... 4 - 87
Partitioned Snapshot Logs ..... 4 - 87
ALTER SYSTEM ..... 4 - 88
Restricting Logons ..... 4 - 97
Clearing the Shared Pool ..... 4 - 97
Performing a Checkpoint ..... 4 - 97
Checking Datafiles ..... 4 - 98
Using Resource Limits ..... 4 - 98
Global Name Resolution ..... 4 - 99
Managing Processes for the Multithreaded Server ..... 4 - 99
Using Licensing Limits ..... 4 - 101
Switching Redo Log File Groups ..... 4 - 102
Enabling and Disabling Distributed Recovery ..... 4 - 103

Terminating a Session ..... 4 - 103
Disconnecting a Session ..... 4 - 104
ALTER TABLE ..... 4 - 106
Adding Columns ..... 4 - 122
Modifying Column Definitions ..... 4 - 123
Index-Organized Tables ..... 4 - 125
LOB Columns ..... 4 - 125
Nested Table Columns ..... 4 - 126
REFs ..... 4 - 127
Modifying Table Partitions ..... 4 - 128
ALTER TABLESPACE ..... 4 - 133
Using ALTER TABLESPACE ..... 4 - 138
ALTER TRIGGER ..... 4 - 141
Invalid Triggers ..... 4 - 141
Enabling and Disabling Triggers ..... 4 - 142
ALTER TYPE ..... 4 - 144
Restriction ..... 4 - 147
ALTER USER ..... 4 - 150
Establishing Default Roles ..... 4 - 152
Changing Authentication Methods ..... 4 - 152
ALTER VIEW ..... 4 - 154
Recompiling Views ..... 4 - 154
ANALYZE ..... 4 - 156
Restrictions ..... 4 - 160
Collecting Statistics ..... 4 - 160
Clusters ..... 4 - 163
Deleting Statistics ..... 4 - 163
Validating Structures ..... 4 - 164
Listing Chained Rows ..... 4 - 166
ARCHIVE LOG clause ..... 4 - 167
Restrictions ..... 4 - 169
AUDIT (SQL Statements) ..... 4 - 170
Auditing ..... 4 - 171
Statement Options for Database Objects ..... 4 - 172
Statement Options for Commands ..... 4 - 174

Shortcuts for System Privileges and Statement Options ..... 4 - 176
AUDIT (Schema Objects) ..... 4 - 178
Object Options ..... 4 - 179
Default Auditing ..... 4 - 180
COMMENT ..... 4 - 183
Using Comments ..... 4 - 183
COMMIT ..... 4 - 185
About Transactions ..... 4 - 186
Ending Transactions ..... 4 - 187
CONSTRAINT clause ..... 4 - 188
Defining Integrity Constraints ..... 4 - 192
NOT NULL Constraints ..... 4 - 193
UNIQUE Constraints ..... 4 - 193
PRIMARY KEY Constraints ..... 4 - 195
Referential Integrity Constraints ..... 4 - 196
CHECK Constraints ..... 4 - 201
DEFERRABLE Constraints ..... 4 - 204
Enabling and Disabling Constraints ..... 4 - 205
CREATE CLUSTER ..... 4 - 207
About Clusters ..... 4 - 210
Cluster Keys ..... 4 - 210
Types of Clusters ..... 4 - 211
Cluster Size ..... 4 - 212
Adding Tables to a Cluster ..... 4 - 213
CREATE CONTROLFILE ..... 4 - 215
Re-creating Control Files ..... 4 - 218
CREATE DATABASE ..... 4 - 219
Examples ..... 4 - 223
CREATE DATABASE LINK ..... 4 - 225
Creating Database Links ..... 4 - 226
Current-User Database Links ..... 4 - 227
Examples ..... 4 - 228
CREATE DIRECTORY ..... 4 - 230
Directory Objects ..... 4 - 231
CREATE FUNCTION ..... 4 - 232

Examples ... 4 - 235
CREATE INDEX ... 4 - 237
Creating Indexes ... 4 - 243
Index Columns ... 4 - 243
Multiple Indexes Per Table ... 4 - 244
The NOSORT Option ... 4 - 244
NOLOGGING ... 4 - 245
Nulls ... 4 - 245
Creating Cluster Indexes ... 4 - 245
Creating Partitioned Indexes ... 4 - 246
Creating Bitmap Indexes ... 4 - 246
Creating Indexes on Nested Table Columns ... 4 - 247
CREATE LIBRARY ... 4 - 248
Examples ... 4 - 249
CREATE PACKAGE ... 4 - 250
Packages ... 4 - 251
CREATE PACKAGE BODY ... 4 - 254
Examples ... 4 - 255
CREATE PROCEDURE ... 4 - 259
Using Procedures ... 4 - 262
CREATE PROFILE ... 4 - 265
Using Profiles ... 4 - 268
Fractions in Dates ... 4 - 269
The DEFAULT Profile ... 4 - 269
CREATE ROLE ... 4 - 272
Using Roles ... 4 - 273
Roles Predefined by Oracle ... 4 - 273
CREATE ROLLBACK SEGMENT ... 4 - 275
Rollback Segments and Tablespaces ... 4 - 276
CREATE SCHEMA ... 4 - 278
Creating Schemas ... 4 - 278
CREATE SEQUENCE ... 4 - 281
Using Sequences ... 4 - 283
Sequence Defaults ... 4 - 284
Incrementing Sequence Values ... 4 - 284

Caching Sequence Numbers ..... 4 - 284
Accessing Sequence Values ..... 4 - 285
CREATE SNAPSHOT ..... 4 - 286
About Snapshots ..... 4 - 291
Types of Snapshots ..... 4 - 291
Refreshing Snapshots ..... 4 - 292
Specifying Rollback Segments ..... 4 - 294
Specifying Primary-Key or ROWID Snapshots ..... 4 - 295
Partitioned Snapshots ..... 4 - 296
CREATE SNAPSHOT LOG ..... 4 - 297
Using Snapshot Logs ..... 4 - 299
Recording Primary Keys, ROWIDs, and Filter Columns ..... 4 - 300
CREATE SYNONYM ..... 4 - 302
Using Synonyms ..... 4 - 303
Scope of Synonyms ..... 4 - 304
CREATE TABLE ..... 4 - 306
Examples ..... 4 - 321
LOB Column Example ..... 4 - 322
Index-Organized Tables ..... 4 - 323
Partitioned Tables ..... 4 - 323
Object Tables ..... 4 - 324
Nested Table Storage ..... 4 - 325
REFs ..... 4 - 325
CREATE TABLESPACE ..... 4 - 328
About Tablespaces ..... 4 - 330
CREATE TRIGGER ..... 4 - 333
Using Triggers ..... 4 - 336
Conditional Predicates ..... 4 - 337
Parts of a Trigger ..... 4 - 338
Types of Triggers ..... 4 - 339
Snapshot Log Triggers ..... 4 - 340
INSTEAD OF Triggers ..... 4 - 342
User-Defined Types, LOB, and REF Columns ..... 4 - 343
CREATE TYPE ..... 4 - 345
Incomplete Object Types ..... 4 - 351

Constructors ..... 4 - 352
CREATE TYPE BODY ..... 4 - 353
TYPE and TYPE BODY ..... 4 - 355
CREATE USER ..... 4 - 357
Verifying Users Through Your Operating System ..... 4 - 359
Verifying Users Through the Network ..... 4 - 360
Establishing Tablespace Quotas for Users ..... 4 - 360
Granting Privileges to a User ..... 4 - 360
CREATE VIEW ..... 4 - 363
Using Views ..... 4 - 366
The View Query ..... 4 - 366
Join Views ..... 4 - 368
Partition Views ..... 4 - 369
Examples ..... 4 - 370
Object Views ..... 4 - 370
DEALLOCATE UNUSED clause ..... 4 - 372
Deallocating Unused Space ..... 4 - 372
DELETE ..... 4 - 374
Using DELETE ..... 4 - 377
Deleting from a Single Partition ..... 4 - 378
The RETURNING Clause ..... 4 - 378
DISABLE clause ..... 4 - 380
Using the DISABLE Clause ..... 4 - 381
DROP clause ..... 4 - 384
Removing Integrity Constraints ..... 4 - 384
DROP CLUSTER ..... 4 - 386
Restrictions ..... 4 - 386
DROP DATABASE LINK ..... 4 - 388
Restrictions ..... 4 - 388
Example ..... 4 - 388
DROP DIRECTORY ..... 4 - 389
Dropping a Directory ..... 4 - 389
DROP FUNCTION ..... 4 - 390
Dropping Functions ..... 4 - 390
DROP INDEX ..... 4 - 392

Dropping an Index ..... 4 - 392
DROP LIBRARY ..... 4 - 393
Example ..... 4 - 393
DROP PACKAGE ..... 4 - 394
Dropping a Package ..... 4 - 394
DROP PROCEDURE ..... 4 - 396
Dropping a Procedure ..... 4 - 396
DROP PROFILE ..... 4 - 398
Dropping a Profile ..... 4 - 398
DROP ROLE ..... 4 - 399
Dropping a Role ..... 4 - 399
DROP ROLLBACK SEGMENT ..... 4 - 400
Dropping Rollback Segments ..... 4 - 400
DROP SEQUENCE ..... 4 - 401
Dropping Sequences ..... 4 - 401
DROP SNAPSHOT ..... 4 - 402
Dropping Snapshots ..... 4 - 402
DROP SNAPSHOT LOG ..... 4 - 403
Dropping Snapshot Logs ..... 4 - 403
DROP SYNONYM ..... 4 - 404
Dropping Synonyms ..... 4 - 404
DROP TABLE ..... 4 - 405
Dropping Tables ..... 4 - 405
DROP TABLESPACE ..... 4 - 407
Dropping Tablespaces ..... 4 - 407
DROP TRIGGER ..... 4 - 409
Dropping Triggers ..... 4 - 409
DROP TYPE ..... 4 - 410
Dropping Types ..... 4 - 410
DROP TYPE BODY ..... 4 - 412
Dropping Type Bodies ..... 4 - 412
DROP USER ..... 4 - 414
Dropping Users and Their Objects ..... 4 - 414
DROP VIEW ..... 4 - 416
Dropping Views ..... 4 - 416

ENABLE clause ..... 4 - 417
Enabling and Disabling Constraints ..... 4 - 419
How Oracle Validates Integrity Constraints ..... 4 - 421
How to Identify Exceptions ..... 4 - 421
Enabling Triggers ..... 4 - 424
EXPLAIN PLAN ..... 4 - 425
Using EXPLAIN PLAN ..... 4 - 426
EXPLAIN PLAN and Partitioned Tables ..... 4 - 427
EXPLAIN PLAN and Parallel DML ..... 4 - 430
Filespec ..... 4 - 431
Examples ..... 4 - 432
GRANT (System Privileges and Roles) ..... 4 - 434
Granting System Privileges and Roles ..... 4 - 435
Granting the ADMIN OPTION ..... 4 - 441
Other Authorization Methods ..... 4 - 442
Examples ..... 4 - 442
GRANT (Object Privileges) ..... 4 - 444
Database Object Privileges ..... 4 - 446
Synonym Privileges ..... 4 - 448
Directory Privileges ..... 4 - 448
Examples ..... 4 - 449
INSERT ..... 4 - 451
The VALUES Clause and Subqueries ..... 4 - 454
Parallel DML ..... 4 - 454
Inserting Into Views ..... 4 - 454
The RETURNING Clause ..... 4 - 455
Examples ..... 4 - 455
LOCK TABLE ..... 4 - 458
Locking Tables ..... 4 - 459
NOAUDIT (SQL Statements) ..... 4 - 461
Stopping Auditing ..... 4 - 462
NOAUDIT (Schema Objects) ..... 4 - 463
Examples ..... 4 - 464
PARALLEL clause ..... 4 - 465
Using the PARALLEL Clause ..... 4 - 466

Nonpartitioned Tables and Indexes ..... 4 - 466
Partitioned Tables and Indexes ..... 4 - 467
Examples ..... 4 - 467
RECOVER clause ..... 4 - 469
Examples ..... 4 - 471
RENAME ..... 4 - 473
Renaming Objects ..... 4 - 473
Restrictions ..... 4 - 473
REVOKE (System Privileges and Roles) ..... 4 - 475
Revoking Privileges ..... 4 - 475
Revoking Roles ..... 4 - 476
Limitations ..... 4 - 476
Examples ..... 4 - 477
REVOKE (Schema Object Privileges) ..... 4 - 478
Revoking Object Privileges ..... 4 - 480
Using FORCE ..... 4 - 480
Revoking Multiple Identical Grants ..... 4 - 480
Cascading Revokes ..... 4 - 481
Examples ..... 4 - 481
ROLLBACK ..... 4 - 484
Rolling Back Transactions ..... 4 - 484
Distributed Transactions ..... 4 - 486
SAVEPOINT ..... 4 - 487
Creating Savepoints ..... 4 - 487
SELECT ..... 4 - 489
Creating Simple Queries ..... 4 - 493
Hierarchical Queries ..... 4 - 495
GROUP BY Clause ..... 4 - 499
HAVING Clause ..... 4 - 500
UNION, UNION ALL, INTERSECT, and MINUS ..... 4 - 501
ORDER BY Clause ..... 4 - 501
FOR UPDATE Clause ..... 4 - 503
Joins ..... 4 - 505
SET CONSTRAINT(S) ..... 4 - 514
Examples ..... 4 - 514

SET ROLE ..... 4 - 516
Privilege Domains ..... 4 - 517
Examples ..... 4 - 518
SET TRANSACTION ..... 4 - 519
Establishing Read-Only Transactions ..... 4 - 520
Assigning Transactions to Rollback Segments ..... 4 - 521
STORAGE clause ..... 4 - 523
Specifying Storage Parameters ..... 4 - 526
Rollback Segments and MAXEXTENTS UNLIMITED ..... 4 - 527
Examples ..... 4 - 527
Subqueries ..... 4 - 530
Using Subqueries ..... 4 - 532
Using Flattened Subqueries ..... 4 - 533
Correlated Subqueries ..... 4 - 534
Selecting from the DUAL Table ..... 4 - 535
Using Sequences ..... 4 - 536
Distributed Queries ..... 4 - 536
TRUNCATE ..... 4 - 538
Truncating Tables and Clusters ..... 4 - 539
Restrictions ..... 4 - 540
Examples ..... 4 - 540
UPDATE ..... 4 - 542
Updating Views ..... 4 - 545
Updating Partitioned Tables ..... 4 - 546
Correlated Update ..... 4 - 547
The RETURNING Clause ..... 4 - 549

A Syntax Diagrams

B Oracle and Standard SQL

C Oracle Reserved Words and Keywords

Send Us Your Comments

Oracle8 SQL Reference, Release 8.0

Part No. A58225-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). Please send your comments to

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

or e-mail comments to: infodev@us.oracle.com.

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 Standards Organization (ISO) SQL92 standard at entry level conformance.

For information on PL/SQL, Oracle's procedural language extension to SQL, see *PL/SQL User's Guide and Reference.*

Detailed descriptions of Oracle embedded SQL can be found in the *Pro*C/C++ Precompiler Programmer's Guide*, *SQL*Module for Ada Programmer's Guide*, and the *Pro*COBOL Precompiler Programmer's Guide*.

Features and Functionality

Oracle8 SQL Reference contains information that describes the features and functionality of the Oracle8 and the Oracle8 Enterprise Edition products. Oracle8 and Oracle8 Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use the CREATE TYPE command, you must have the Enterprise Edition and the Objects Option.

For information about the differences between Oracle8 and the Oracle8 Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8 and the Oracle8 Enterprise Edition*.

Audience

This reference is intended for all users of Oracle SQL.

How this Reference Is Organized

This reference is divided into the following parts:

Chapter 1: Introduction

This chapter defines SQL and describes its history as well as the advantages of using it to access relational databases.

Chapter 2: Elements of Oracle8 SQL

This chapter describes the basic building blocks of an Oracle database and the elements of Oracle SQL.

Chapter 3: Operators, Functions, Expressions, Conditions

This chapter describes how to use SQL operators and functions to combine data into expressions and conditions.

Chapter 4: Commands

This chapter lists and describes all of the SQL commands in alphabetical order.

Appendix A: Syntax Diagrams

This appendix describes how to read the syntax diagrams that appear in this reference.

Appendix B: Oracle and Standard SQL

This appendix describes Oracle compliance with ANSI and ISO standards and lists Oracle extensions beyond the standards.

Appendix C: Oracle Reserved Words and Keywords

This appendix lists Oracle reserved words and keywords.

Conventions Used in this Reference

This section explains the conventions used in this book including:

- Text
- Syntax Diagrams and Notation
- Code Examples
- Example Data

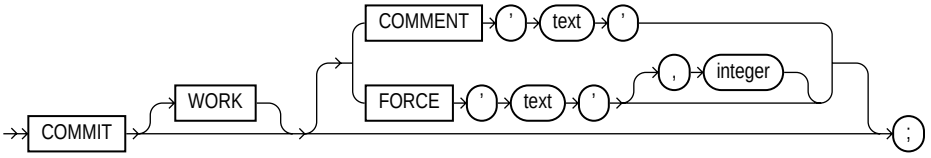
Text

The text in this reference adheres to the following conventions:

UPPERCASE	Uppercase text calls attention to SQL commands, keywords, filenames, and initialization parameters.
italics	Italicized text calls attention to definitions of terms and parameters of SQL commands.

Syntax Diagrams and Notation

Syntax Diagrams. This reference uses syntax diagrams to show SQL commands in Chapter 4, "Commands", and to show other elements of the SQL language in Chapter 2, "Elements of Oracle8 SQL", and Chapter 3, "Operators, Functions, Expressions, Conditions". These syntax diagrams use lines and arrows to show syntactic structure, as shown here:



If you are not familiar with this type of syntax diagram, refer to Appendix A, "Syntax Diagrams", for a description of how to read them. This section describes the components of syntax diagrams and gives examples of how to write SQL statements. Syntax diagrams are made up of these items:

Keywords. Keywords have special meanings in the SQL language. In the syntax diagrams, keywords appear in UPPERCASE. You must use keywords in your SQL statements exactly as they appear in the syntax diagram, except that they can be either uppercase or lowercase. For example, you must use the CREATE keyword to begin your CREATE TABLE statements just as it appears in the CREATE TABLE syntax diagram.

Parameters. Parameters act as placeholders in syntax diagrams. They appear in lowercase. Parameters are usually names of database objects, Oracle datatype names, or expressions. When you see a parameter in a syntax diagram, substitute an object or expression of the appropriate type in your SQL statement. For example, to write a CREATE TABLE statement, use the name of the table you want

to create, such as EMP, in place of the *table* parameter in the syntax diagram. (Note that parameter names appear in italics in the text.)

This lists shows parameters that appear in the syntax diagrams and provides examples of the values you might substitute for them in your statements:

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" on page 2-44.	`emp`
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 quotes. See the syntax description of '*text*' in "Text" on page 2-2.	`'Employee records'`
char	The substitution value must be an expression of datatype CHAR or VARCHAR2 or a character literal in single quotes.	`ename` `'Smith'`
condition	The substitution value must be a condition that evaluates to TRUE or FALSE. See the syntax description of *condition* in "Conditions" on page 3-90.	`ename >'A'`

date
d
	The substitution value must be a date constant or an expression of DATE datatype.	`TO_DATE(`
		`'01-Jan-1994',`
		`'DD-MON-YYYY')`

expr	The substitution value can be an expression of any datatype as defined in the syntax description of *expr* in "Expressions" on page 3-78.	`sal + 1000`
integer	The substitution value must be an integer as defined by the syntax description of integer in "Integer" on page 2-3.	`72`
label	The substitution value must be an expression of datatype MLSLABEL. For information on such expressions, see your Trusted Oracle documentation.	`TO_LABEL(` `'SENSITIVE:ALPHA')`

Parameter	Description	Examples
number *m* *n*	The substitution value must be an expression of NUMBER datatype or a number constant as defined in the syntax description of *number* in "Number" on page 2-4.	`AVG(sal)` `15 * 7`
raw	The substitution value must be an expression of datatype RAW.	`HEXTORAW('7D')`
rowid	The substitution value must be an expression of datatype ROWID.	`AAAAZzAABAAABrXAAA`
subquery	The substitution value must be a SELECT statement that will be used in another SQL statement. See "Subqueries" on page 4-530.	`SELECT ename` `FROM emp`
:host_variable	The substitution value must be the name of a variable declared in an embedded SQL program. This reference also uses *:host_integer* and *:d* to indicate specific datatypes.	`:employee_number`
cursor	The substitution value must be the name of a cursor in an embedded SQL program.	`curs1`
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 a Net8 database connection. For details, see the user's guide for your specific Net8 protocol.	
statement_name *block_name*	The substitution value must be an identifier for a SQL statement or PL/SQL block.	`s1` `lab1`

Code Examples

This reference contains many examples of SQL statements. These examples show you how to use elements of SQL. The following example shows a CREATE TABLE statement:

```
CREATE TABLE accounts
    ( accno     NUMBER,
      owner     VARCHAR2(10),
      balance   NUMBER(7,2) );
```

Note that examples appear in a different font than the text.

Examples follow these case conventions:

- Keywords, such as CREATE and NUMBER, appear in uppercase.
- Names of database objects and their parts, such as ACCOUNTS and ACCNO, appear in lowercase, although they appear in uppercase in the text.

SQL is not case sensitive (except for quoted identifiers), so you need not follow these conventions when writing your own SQL statements, although your statements may be easier for you to read if you do.

Some Oracle tools require you to terminate SQL statements with a special character. For example, the code examples in this reference were issued through SQL*Plus, and therefore are terminated with a semicolon (;). If you issue these example statements to Oracle, you must terminate them with the special character expected by the Oracle tool you are using.

Example Data

Many of the examples in this reference operate on sample tables. The definitions of some of these tables appear in a SQL script available on your distribution medium. On most operating systems the name of this script is UTLSAMPL.SQL, although its exact name and location depend on your operating system. This script creates sample users and creates these sample tables in the schema of the user SCOTT:

```
CREATE TABLE dept
    (deptno    NUMBER(2)      CONSTRAINT pk_dept PRIMARY KEY,
     dname     VARCHAR2(14),
     loc       VARCHAR2(13) );
CREATE TABLE emp
    (empno     NUMBER(4)      CONSTRAINT pk_emp PRIMARY KEY,
     ename     VARCHAR2(10),
     job       VARCHAR2(9),
     mgr       NUMBER(4),
     hiredate  DATE,
     sal       NUMBER(7,2),
     comm      NUMBER(7,2),
     deptno    NUMBER(2)    CONSTRAINT fk_deptno REFERENCES dept );
CREATE TABLE bonus
    (ename     VARCHAR2(10),
     job       VARCHAR2(9),
     sal       NUMBER,
     comm      NUMBER );
CREATE TABLE salgrade
    (grade     NUMBER,
     losal     NUMBER,
```

```
     hisal     NUMBER );
```

The script also fills the sample tables with this data:

```
SELECT * FROM dept

DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON

SELECT * FROM emp

EMPNO  ENAME   JOB          MGR  HIREDATE    SAL   COMM  DEPTNO
-----  ------- --------- ------ --------- ------ ------ -------
 7369  SMITH   CLERK       7902 17-DEC-80    800             20
 7499  ALLEN   SALESMAN    7698 20-FEB-81   1600    300      30
 7521  WARD    SALESMAN    7698 22-FEB-81   1250    500      30
 7566  JONES   MANAGER     7839 02-APR-81   2975             20
 7654  MARTIN  SALESMAN    7698 28-SEP-81   1250   1400      30
 7698  BLAKE   MANAGER     7839 01-MAY-81   2850             30
 7782  CLARK   MANAGER     7839 09-JUN-81   2450             10
 7788  SCOTT   ANALYST     7566 19-APR-87   3000             20
 7839  KING    PRESIDENT         17-NOV-81   5000             10
 7844  TURNER  SALESMAN    7698 08-SEP-81   1500             30
 7876  ADAMS   CLERK       7788 23-MAY-87   1100             20
 7900  JAMES   CLERK       7698 03-DEC-81    950             30
 7902  FORD    ANALYST     7566 03-DEC-81   3000             20
 7934  MILLER  CLERK       7782 23-JAN-82   1300             10

SELECT * FROM salgrade

GRADE LOSAL HISAL
----- ----- -----
1     700   1200
2     1201  1400
3     1401  2000
4     2001  3000
5     3001  9999
```

To perform all the operations of the script, run it when you are logged into Oracle as the user SYSTEM.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of our references. As we write, revise, and evaluate, your opinions are the most important input we receive. At the front of this reference is a Reader's Comment Form that we encourage you to use to tell us both what you like and what you dislike about this (or other) Oracle manuals. If the form is missing, or you would like to contact us, please use the following address or fax number:

Oracle8 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
FAX: 650-506-7200

You can also e-mail your comments to: infodev@us.oracle.com

1

Introduction

Structured Query Language (SQL), is the set of commands that all programs and users must use to 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 relational database systems. Topics include:

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

History of SQL

The paper, "A Relational Model of Data for Large Shared Data Banks," by Dr. E. F. Codd, was published 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 Corporation) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

SQL Standards

Oracle SQL complies with industry-accepted standards. Oracle Corporation ensures future compliance with evolving SQL standards by actively involving key personnel in SQL standards committees. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Standards Organization (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.

The latest SQL standard published by ANSI and ISO is often called SQL92 (and sometimes SQL2). The formal names of the new standard are:

- ANSI X3.135-1992, "Database Language SQL"
- ISO / IEC 9075:1992, "Database Language SQL"

SQL92 defines four levels of compliance: Entry, Transitional, Intermediate, and Full. A conforming SQL implementation must support at least Entry SQL. Oracle8, Release 8.0, fully supports Entry SQL and has many features that conform to Transitional, Intermediate, or Full SQL.

Oracle8 conformance to Entry-level SQL92 was tested by the National Institute for Standards and Technology (NIST) using the Federal Information Processing Standard (FIPS), FIPS PUB 127-2.

> **Additional Information:** For more information about Oracle and standard SQL, see Appendix B, "Oracle and Standard SQL".

How SQL Works

The strengths of SQL benefit all types of users, including application programmers, database administrators, management, 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, and all SQL statements are instructions to the database. In this it 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 were not part of SQL originally,

but they are found in the recently accepted optional part of SQL, ISO/IEC 9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL/SQL extension to SQL is close to PSM.

Essentially, 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 that determines a fast means of accessing the specified data. Oracle also provides techniques you can use to make the optimizer perform its job better.

SQL provides commands 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 above 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.

Embedded SQL

Embedded SQL refers to the use of standard SQL commands embedded within a procedural programming language. The embedded SQL commands are documented in the Oracle precompiler books, *SQL*Module for Ada Programmer's Guide, Pro*C/C++ Precompiler Programmer's Guide,* and *Pro*COBOL Precompiler Programmer's Guide*.

Embedded SQL is a collection of these commands:

- all SQL commands, such as SELECT and INSERT, available with SQL with interactive tools
- dynamic SQL execution commands, such as PREPARE and OPEN, which integrate the standard SQL commands with a procedural programming language

Embedded SQL also includes extensions to some standard SQL commands. Embedded SQL is supported by the Oracle precompilers. The Oracle precompilers interpret embedded SQL statements and translate them into statements that can be understood by procedural language compilers.

Each of these Oracle precompilers translates embedded SQL programs into a different procedural language:

- the Pro*C/C++ precompiler
- the Pro*COBOL precompiler
- the Pro*FORTRAN precompiler
- the SQL*Module for ADA

> **Additional Information:** For a definition of the Oracle precompilers and the embedded SQL commands, see *SQL*Module for Ada Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *Pro*COBOL Precompiler Programmer's Guide*.

Lexical Conventions

The following lexical conventions for issuing SQL statements apply specifically to Oracle's implementation of SQL, but are generally acceptable in all 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 command. Thus, Oracle evaluates the following two statements in the same manner:

```
SELECT ENAME,SAL*12,MONTHS_BETWEEN(HIREDATE,SYSDATE) FROM EMP;

SELECT ENAME,
     SAL * 12,
          MONTHS_BETWEEN( HIREDATE, SYSDATE )
FROM EMP;
```

Case is insignificant in reserved words, keywords, identifiers and parameters. However, case is significant in text literals and quoted names. See the syntax description in "Text" on page 2-2.

Tools Support

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

2

Elements of Oracle8 SQL

This chapter contains reference information on the basic elements of Oracle SQL.

> **Note:** Commands and descriptions preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Before using any of the commands described in Chapter 4, “Commands”, you should familiarize yourself with the concepts covered in this chapter:

- Literals
- Text
- Integer
- Number
- Datatypes
- Nulls
- Pseudocolumns
- Comments
- Database Objects
- Schema Object Names and Qualifiers
- Referring to Schema Objects and Parts

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. Note that character literals are enclosed in single quotation marks, which enable Oracle to distinguish them from schema object names.

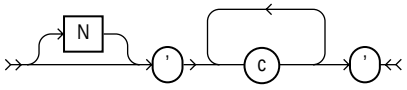
Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions. You can specify character literals with the '*text*' notation, national character literals with the *N'text'* notation, and numeric literals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

Text

Text specifies a text or character literal. You must use this notation to specify values whenever '*text*' or *char* appear in expressions, conditions, SQL functions, and SQL commands in other parts of this reference.

The syntax of text is as follows:

text::=



where

N	specifies representation of the literal using the national character set. Text entered using this notation is translated into the national character set by Oracle when used.
c	is any member of the user's character set, except a single quotation mark (').
' '	are two single quotation marks that begin and end text literals. To represent one single quotation mark within a literal, enter two single quotation marks.

A text literal must be enclosed in single quotation marks. This reference uses the terms *text literal* and *character literal* interchangeably.

Text literals have properties of both the CHAR and VARCHAR2 datatypes:

- Within expressions and conditions, Oracle treats text literals as though they have the datatype CHAR by comparing them using blank-padded comparison semantics.
- A text literal can have a maximum length of 4000 bytes.

Here are some valid text literals:

```
'Hello'
'ORACLE.dbs'
'Jackie''s raincoat'
'09-MAR-92'
N'nchar literal'
```

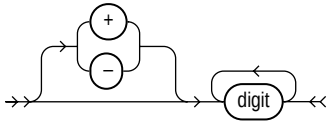
For more information, see the syntax description of *expr* in "Expressions" on page 3-78.

Integer

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

The syntax of *integer* is as 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
```

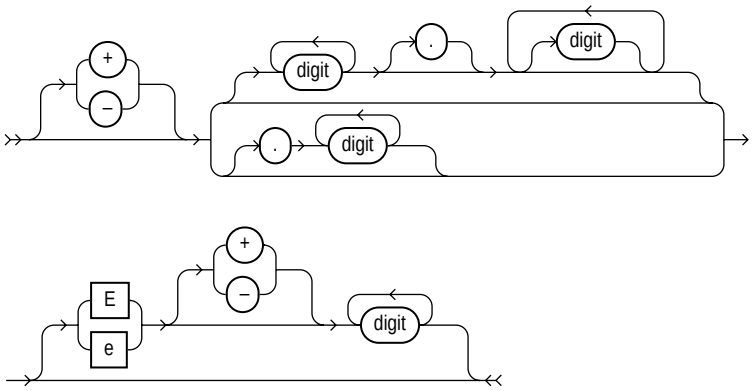
For more information, see the syntax description of *expr* in "Expressions" on page 3-78.

Number

You must use the number notation to specify values whenever *number* appears in expressions, conditions, SQL functions, and SQL commands in other parts of this reference.

The syntax of *number* is as follows:

number::=



where

+, -	indicates a positive or negative value. If you omit the sign, a positive value is the default.
digit	is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
e, 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.

A *number* can store a maximum of 38 digits of precision.

If you have established a decimal character other than a period (.) with the initialization parameter NLS_NUMERIC_CHARACTERS, you must specify numeric literals with '*text*' notation. In such cases, Oracle automatically converts the text literal to a numeric value.

For example, if the NLS_NUMERIC_CHARACTERS parameter specifies a decimal character of comma, specify the number 5.123 as follows:

```
'5,123'
```

For more information on this parameter, see *Oracle8 Reference*.

Here are some valid representations of *number*:

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

For more information, see the syntax description of *expr* in "Expressions" on page 3-78.

Datatypes

Each literal or column value manipulated by Oracle has a *datatype*. A value's datatype associates a fixed set of properties with the value. These properties cause Oracle to treat values of one datatype differently from values of another. For example, you can add values of NUMBER datatype, but not values of RAW datatype.

When you create a table or cluster, you must specify an internal datatype for each of its columns. When you create a procedure or stored function, you must specify an internal datatype for each of its arguments. These datatypes 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 column's datatype. For example, if you insert '01-JAN-92' into a DATE column, Oracle treats the '01-JAN-92' character string as a DATE value after verifying that it translates to a valid date.

Table 2–1 summarizes Oracle internal datatypes. The rest of this section describes these datatypes in detail.

Note: The Oracle precompilers recognize other datatypes in embedded SQL programs. These datatypes are called *external datatypes* and are associated with host variables. Do not confuse the internal datatypes with external datatypes. For information on external datatypes, including how Oracle converts between internal and external datatypes, see *Pro*COBOL Precompiler Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *SQL*Module for Ada Programmer's Guide*.

Table 2–1 Internal Datatype Summary

Code	Internal Datatype	Description
1	VARCHAR2(*size*)	Variable-length character string having maximum length *size* bytes. Maximum *size* is 4000, and minimum is 1. You must specify *size* for a VARCHAR2.
	NVARCHAR2(*size*)	Variable-length character string having maximum length *size* characters or bytes, depending on the choice of national character set. Maximum *size* is determined by the number of bytes required to store each character, with an upper limit of 4000 bytes. You must specify *size* for NVARCHAR2.
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.
8	LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes.
12	DATE	Valid date range from January 1, 4712 BC to December 31, 4712 AD.
23	RAW(*size*)	Raw binary data of length *size* bytes. Maximum *size* is 2000 bytes. You must specify *size* for a RAW value.
24	LONG RAW	Raw binary data of variable length up to 2 gigabytes.
69	ROWID	Hexadecimal string representing the unique address of a row in its table. This datatype is primarily for values returned by the ROWID pseudocolumn.
96	CHAR(*size*)	Fixed length character data of length *size* bytes. Maximum *size* is 2000 bytes. Default and minimum *size* is 1 byte.
	NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the choice of national character set. Maximum *size* is determined by the number of bytes required to store each character, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character or 1 byte, depending on the character set.
106	MLSLABEL	Binary format of an operating system label. This datatype is used for backward compatibility with Trusted Oracle.

Table 2–1 (Cont.) Internal Datatype Summary

Code	Internal Datatype	Description
112	CLOB	A character large object containing single-byte characters. Variable-width character sets are not supported. Maximum size is 4 gigabytes.
	NCLOB	A character large object containing fixed-width multibyte characters. Variable-width character sets are not supported. Maximum size is 4 gigabytes. Stores national character set data.
113	BLOB	A binary large object. Maximum size is 4 gigabytes.
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 codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned when you use the DUMP function.

Character Datatypes

Character datatypes store character (alphanumeric) data—words and free-form text—in the database or national character set. They are less restrictive than other datatypes 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 Code Page 500, specified when the database was created. Oracle supports both single-byte and multibyte character sets.

These datatypes are used for character data:

- CHAR Datatype
- NCHAR Datatype
- NVARCHAR2 Datatype
- VARCHAR2 Datatype

CHAR Datatype

The CHAR datatype specifies a fixed-length character string. When you create a table with a CHAR column, you supply the column length in bytes. Oracle subsequently ensures that all values stored in that column have this length. If you insert a value that is shorter than the column length, Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, Oracle returns an error.

The default length for a CHAR column is 1 character and the maximum allowed is 2000 characters. A zero-length string can be inserted into a CHAR column, but the column is blank-padded to 1 character when used in comparisons. For information on comparison semantics, see "Datatype Comparison Rules" on page 2-24.

NCHAR Datatype

The NCHAR datatype specifies a fixed-length national character set character string. When you create a table with an NCHAR column, you define the column length either in characters or in bytes. You define the national character set when you create your database.

If the national character set specified is fixed width, such as JA16EUCFIXED, then you declare the NCHAR column size as the number of characters desired for the string length. If the national character set is variable width, such as JA16SJIS, you declare the column size in bytes. The following statement creates a table with one NCHAR column that can store strings up to 30 characters in length using JA16EUCFIXED as the national character set:

```
CREATE TABLE tab1 (col1 NCHAR(30));
```

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NCHAR refer to the number of characters if the national character set is fixed width and refer to the number of bytes if the national character set is variable width. The maximum column size allowed is 2000 bytes. For fixed-width, multibyte character sets, the maximum length of a column allowed is the number of characters that fit into no more than 2000 bytes.

If you insert a value that is shorter than the column length, Oracle blank-pads the value to column length. You cannot insert a CHAR value into an NCHAR column, nor can you insert an NCHAR value into a CHAR column.

The following example compares the `col1` column of `tab1` with national character set string *NCHAR literal*:

```
SELECT * FROM tab1 WHERE col1 = N'NCHAR literal';
```

OBJ You cannot create an object with NCHAR attributes, but you can specify NCHAR parameters in methods.

NVARCHAR2 Datatype

The NVARCHAR2 datatype specifies a variable-length national character set character string. When you create a table with an NVARCHAR2 column, you supply the maximum number of characters or bytes that it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided it does not exceed the column's maximum length.

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NVARCHAR2 refer to the number of characters if the national character set is fixed width and refer to the number of bytes if the national character set is variable width. The maximum column size allowed is 4000 bytes. For fixed-width, multibyte character sets, the maximum length of a column allowed is the number of characters that fit into no more than 4000 bytes.

The following statement creates a table with one NVARCHAR2 column of 2000 characters in length (stored as 4000 bytes, because each character takes two bytes) using JA16EUCFIXED as the national character set:

```
CREATE TABLE tab1 (col1 NVARCHAR2(2000));
```

OBJ You cannot create an object with NVARCHAR2 attributes, but you can specify NVARCHAR2 parameters in methods.

VARCHAR2 Datatype

The VARCHAR2 datatype specifies a variable-length character string. When you create a VARCHAR2 column, you can supply the maximum number of bytes of data that it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided it does not exceed the column's maximum length. This maximum must be at least 1 byte, although the actual length of the string stored is permitted to be zero. If you try to insert a value that exceeds the specified length, Oracle returns an error.

You must specify a maximum length for a VARCHAR2 column. The maximum length of VARCHAR2 data is 4000 bytes. Oracle compares VARCHAR2 values using nonpadded comparison semantics. For information on comparison semantics, see "Datatype Comparison Rules" on page 2-24.

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. Oracle recommends that you use VARCHAR2 rather than VARCHAR. In the future, VARCHAR might be defined as a separate datatype used for variable-length character strings compared with different comparison semantics.

NUMBER Datatype

The NUMBER datatype stores zero, positive and negative fixed and floating-point numbers with magnitudes between 1.0×10^{-130} and $9.9...9 \times 10^{125}$ (38 nines followed by 88 zeroes) with 38 digits of precision. If you specify an arithmetic expression whose value has a magnitude greater than or equal to 1.0×10^{126}, Oracle returns an error.

Specify a fixed-point number using the following form:

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

where:

p is the *precision*, or the total number of digits. Oracle guarantees the portability of numbers with precision ranging from 1 to 38.

s is the *scale*, or the number of digits to the right of the decimal point. The scale can range from -84 to 127.

Specify an integer using the following form:

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

Specify a floating-point number using the following form:

`NUMBER` is a floating-point number with decimal precision 38. Note that a scale value is not applicable for floating-point numbers. (See "Floating-Point Numbers" on page 2-12 for more information.)

Scale and Precision

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, Oracle returns an error. If a value exceeds the scale, Oracle rounds it.

The following examples show how Oracle stores data using different precisions and scales.

Actual Data	Specified As	Stored As
7456123.89	NUMBER	7456123.89
7456123.89	NUMBER(9)	7456124
7456123.89	NUMBER(9,2)	7456123.89
7456123.89	NUMBER(9,1)	7456123.9
7456123.89	NUMBER(6)	exceeds precision
7456123.89	NUMBER(7,-2)	7456100
7456123.89	NUMBER(-7,2)	exceeds precision

Negative Scale

If the scale is negative, 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 Greater than Precision

You can specify a scale that is greater than precision, although it is uncommon. In this case, the precision specifies the maximum number of digits to the right of the decimal point. As with all number datatypes, if the value exceeds the precision, Oracle returns an error message. If the value exceeds the scale, Oracle rounds the value. 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. The following examples show the effects of a scale greater than precision:

Actual Data	Specified As	Stored As
.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

Floating-Point Numbers

Oracle allows you to specify floating-point numbers, which can have a decimal point anywhere from the first to the last digit or can have no decimal point at all. 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.

You can specify floating-point numbers with the form discussed in "NUMBER Datatype" on page 2-10. Oracle also supports the ANSI datatype FLOAT. You can specify this datatype using one of these syntactic forms:

`FLOAT` specifies a floating-point number with decimal precision 38, or binary precision 126.

`FLOAT(`*b*`)` specifies a floating-point number with binary precision b. The precision b can range from 1 to 126. To convert from binary to decimal precision, multiply *b* 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.

LONG Datatype

LONG columns store variable length character strings containing up to 2 gigabytes, 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. Oracle uses LONG columns in the data dictionary to store the text of view definitions. The length of LONG values may be limited by the memory available on your computer.

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 are subject to some restrictions:

- A table cannot contain more than one LONG column.
- LONG columns cannot appear in integrity constraints (except for NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- A stored function cannot return a LONG value.

- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.

LONG columns cannot appear in certain parts of SQL statements:

- WHERE, GROUP BY, ORDER BY, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- the UNIQUE clause of a SELECT statement
- the column list of a CREATE CLUSTER statement
- the CLUSTER clause of a CREATE SNAPSHOT statement
- SQL functions (such as SUBSTR or INSTR)
- expressions or conditions
- SELECT lists of queries containing GROUP BY clauses
- SELECT lists of subqueries or queries combined by set operators
- SELECT lists of CREATE TABLE ... AS SELECT statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG datatype 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 datatype (such as CHAR and VARCHAR2), a LONG column can be referenced in a SQL statement within a trigger. Note that the maximum length for these datatypes is 32K.
- Variables in triggers cannot be declared using the LONG datatype.
- :NEW and :OLD cannot be used with LONG columns.

You can use the Oracle Call Interface functions to retrieve a portion of a LONG value from the database. See *Oracle Call Interface Programmer's Guide*.

DATE Datatype

The DATE datatype stores date and time information. Although date and time information can be represented in both CHAR and NUMBER datatypes, the DATE datatype has special associated properties. For each DATE value, Oracle stores the following information: century, year, month, day, hour, minute, and second.

To specify a date value, you must convert a character or numeric value to a date value with the TO_DATE function. Oracle automatically converts character values that are in the default date format into date values when they are used in date expressions. The default date format is specified by the initialization parameter NLS_DATE_FORMAT and is a string such as 'DD-MON-YY'. This example date format includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year.

If you specify a date value without a time component, the default time is 12:00:00 AM (midnight). If you specify a date value without a date, the default date is the first day of the current month.

The date function SYSDATE returns the current date and time. For information on the SYSDATE and TO_DATE functions and the default date format, see Chapter 3, "Operators, Functions, Expressions, Conditions".

Date Arithmetic

You can add and subtract number constants as well as other dates from dates. Oracle interprets number constants in arithmetic date 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 HIREDATE column of the EMP table from SYSDATE returns the number of days since each employee was hired. You cannot multiply or divide DATE values.

Oracle provides functions for many of the common date operations. For example, the ADD_MONTHS function allows you to add or subtract months from a date. 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. For more information on date functions, see "Date Functions" on page 3-36.

Because each date contains a time component, most results of date operations include a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours.

Using Julian Dates

A Julian date is the number of days since January 1, 4712 BC. Julian dates 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.

Example This statement returns the Julian equivalent of January 1, 1997:

```
SELECT TO_CHAR(TO_DATE('01-01-1997', 'MM-DD-YYYY'),'J')
    FROM DUAL;

TO_CHAR
--------
2450450
```

For a description of the DUAL table, see "Selecting from the DUAL Table" on page 4-535.

RAW and LONG RAW Datatypes

The RAW and LONG RAW datatypes store data that is not to be interpreted (not explicitly converted when moving data between different systems) by Oracle. These datatypes 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; the interpretation is dependent on the use.

RAW is a variable-length datatype like the VARCHAR2 character datatype, except that Net8 (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Net8 and Import/Export automatically convert CHAR, VARCHAR2, and LONG data from the database character set to the user session character set (set by the NLS_LANGUAGE parameter of the ALTER SESSION command), if the two character sets are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

You can index RAW data, but not LONG RAW data.

Large Object (LOB) Datatypes

Internal LOB datatypes—BLOB, CLOB, NCLOB—and external datatype BFILE, can store large and unstructured data such as text, image, video, and spatial data up to 4 gigabytes in size.

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

Internal LOB columns contain LOB locators that can refer to out-of-line or in-line LOB values. Selecting a LOB from a table actually returns the LOB's locator and not

the entire LOB value. The DBMS_LOB package and OCI operations on LOBs are performed through these locators. For more information about these interfaces and LOBs, see *Oracle8 Application Developer's Guide* and *Oracle Call Interface Programmer's Guide*.

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

- LOBs can be attributes of a user-defined datatype (object).
- 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 and 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 4 gigabytes in size. BFILE maximum size is operating system dependent, but cannot exceed 4 gigabytes.
- 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 and / or an object with one or more LOB attributes. (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 so that it points 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. Note that for BFILEs, the actual operating system file is not deleted.

For more information, please refer to the discussion of LOB restrictions in *Oracle8 Application Developer's Guide*.

To access and populate rows of an internal LOB column, use the INSERT statement first to initialize the internal LOB value to empty. Once the row is inserted, you can select the empty LOB and populate it using the DBMS_LOB package or the OCI.

The following example creates a table with LOB columns:

```
CREATE TABLE  person_table (name CHAR(40),
                            resume  CLOB,
                            picture BLOB)
  LOB (resume) STORE AS
               ( TABLESPACE resumes
                  STORAGE (INITIAL 5M NEXT 5M) );
```

Use LOBs to read and write large chunks of the LOB value.

BFILE Datatype

The BFILE datatype enables access to binary file LOBs that are stored in file systems outside the Oracle database. A BFILE column or attribute stores a BFILE locator, which serves as a pointer to a binary file on the server's file system. The locator maintains the directory alias and the filename. See CREATE DIRECTORY on page 4-230.

Binary file LOBs do not participate in transactions. Rather, the underlying operating system provides file integrity and durability. The maximum file size supported is 4 gigabytes.

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

The BFILE datatype allows read-only support of large binary files; you cannot modify 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 the OCI. For more information about LOBs, see *Oracle8 Application Developer's Guide* and *Oracle Call Interface Programmer's Guide*.

BLOB Datatype

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bitstreams with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOBs have full transactional support; changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. The BLOB value manipulations can be committed or rolled back. Note, however, that 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 Datatype

The CLOB datatype stores single-byte character large object data. Variable-width character sets are not supported. CLOBs can store up to 4 gigabytes of character data.

CLOBs have full transactional support; changes made through SQL, the OCI, or the DBMS_LOB package participate fully in the transaction. The CLOB value manipulations can be committed or rolled back. Note, however, that, 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 Datatype

The NCLOB datatype stores fixed-width, multibyte national character set character (NCHAR) data. Variable-width character sets are not supported. NCLOBs can store up to 4 gigabytes of character text data.

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

OBJ You cannot create an object with NCLOB attributes, but you can specify NCLOB parameters in methods.

ROWID Datatype

Each row in the database has an address. You can examine a row's address by querying the pseudocolumn ROWID. Values of this pseudocolumn are hexadecimal strings representing the address of each row. These strings have the datatype ROWID. For more information on the ROWID pseudocolumn, see "Pseudocolumns" on page 2-32. You can also create tables and clusters that contain actual columns having the ROWID datatype. Oracle does not guarantee that the values of such columns are valid ROWIDs.

Restricted ROWIDs

Oracle8 incorporates an extended format for ROWIDs to efficiently support partitioned tables and indexes and tablespace-relative data block addresses (DBAs) without ambiguity.

Character values representing ROWIDs in Oracle7 and earlier releases are as follows:

```
block.row.file
```

where:

block is a hexadecimal string identifying the data block of the datafile containing the row. The length of this string depends on your operating system.

row is a four-digit hexadecimal string identifying the row in the data block. The first row of the block has a digit of 0.

file is a hexadecimal string identifying the database file containing the row. The first datafile has the number 1. The length of this string depends on your operating system.

In Oracle8, this kind of ROWID is called a *restricted* ROWID.

Extended ROWIDs

The Oracle8 *extended* ROWID datatype stored in a user column includes the data in the restricted ROWID plus a *data object number*. The data object number is an identification number assigned to every database segment. You can retrieve the data object number from 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.

Extended ROWIDs are not available directly. You can use a supplied package, DBMS_ROWID, to interpret extended ROWID contents. The package functions extract and provide information that would be available directly from a restricted ROWID. You can use functions from the DBMS_ROWID package as you would any built-in SQL function. Table 2–2 lists the functions and procedures in the DBMS_ROWID package. For more information on DBMS_ROWID, see *Oracle8 Application Developer's Guide*.

Table 2–2 DBMS_ROWID Functions

Function Name	Description
ROWID_CREATE	Create a ROWID, for testing only.
ROWID_TYPE	Returns the ROWID type: 0 is restricted, 1 is extended.
ROWID_OBJECT	Returns the object number of the extended ROWID.
ROWID_RELATIVE_FNO	Returns the file number of a ROWID.

Table 2–2 DBMS_ROWID Functions

Function Name	Description
ROWID_BLOCK_NUMBER	Returns the block number of a ROWID.
ROWID_ROW_NUMBER	Returns the row number.
ROWID_TO_ABSOLUTE_ FNO	Returns the absolute file number associated with the ROWID for a row in a specific table.
ROWID_TO_EXTENDED	Converts a ROWID from restricted format to extended.
ROWID_TO_RESTRICTED	Converts an extended ROWID to restricted format.
ROWID_VERIFY	Checks if a ROWID can be correctly extended by the ROWID_TO_EXTENDED function.

Compatibility and Migration

The restricted form of ROWID is still supported in Oracle8 for backward compatibility, but all tables return ROWIDs in the extended format. For information regarding compatibility and migration issues, see *Oracle8 Migration*.

MLSLABEL Datatype

The MLSLABEL datatype stores the binary format of a label used on a secure operating system. This datatype is supported in Oracle8 for backward compatibility with earlier versions of Oracle servers using Trusted Oracle.

Labels are used by Trusted Oracle to mediate access to information. You can also define columns with this datatype if you are using the standard Oracle server. For more information on Trusted Oracle, including this datatype and labels, see your Trusted Oracle documentation.

ANSI, DB2, and SQL/DS Datatypes

SQL commands that create tables and clusters can also use ANSI datatypes and datatypes from IBM's products SQL/DS and DB2. Oracle recognizes the ANSI or IBM datatype name and records it as the name of the datatype of the column, and then stores the column's data in an Oracle datatype based on the conversions shown in Table 2–3 and Table 2–4.

Table 2–3 ANSI Datatypes Converted to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER(*n*)	CHAR(*n*)
CHAR(*n*)	
CHARACTER VARYING(*n*)	VARCHAR(*n*)
CHAR VARYING(*n*)	
NATIONAL CHARACTER(*n*)	NCHAR(*n*)
NATIONAL CHAR(*n*)	
NCHAR(*n*)	
NATIONAL CHARACTER VARYING(*n*)	NVARCHAR2(*n*)
NATIONAL CHAR VARYING(*n*)	
NCHAR VARYING(*n*)	
NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)
DECIMAL(*p*,*s*)[a]	
INTEGER	NUMBER(38)
INT	
SMALLINT	
FLOAT(*b*)[b]	NUMBER
DOUBLE PRECISION[c]	
REAL[d]	

[a]The NUMERIC and DECIMAL datatypes can specify only fixed-point numbers. For these datatypes, *s* defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision *b*. The default precision for this datatype is 126 binary, or 38 decimal.

[c]The DOUBLE PRECISION datatype is a floating-point number with binary precision 126.

[d]The REAL datatype is a floating-point number with a binary precision of 63, or 18 decimal.

Table 2–4 SQL/DS and DB2 Datatypes Converted to Oracle Datatypes

SQL/DS or DB2 Datatype	Oracle Datatype
CHARACTER(*n*)	CHAR(*n*)
VARCHAR(*n*)	VARCHAR(*n*)
LONG VARCHAR(*n*)	LONG
DECIMAL(*p*,*s*)[a]	NUMBER(*p*,*s*)
INTEGER	NUMBER(38)
SMALLINT	
FLOAT(*b*)[b]	NUMBER

[a]The DECIMAL datatype can specify only fixed-point numbers. For this datatype, *s* defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision *b*. This default precision for this datatype is 126 binary, or 38 decimal.

Do not define columns with these SQL/DS and DB2 datatypes, because they have no corresponding Oracle datatype:

- GRAPHIC
- LONG VARGRAPHIC
- VARGRAPHIC
- TIME
- TIMESTAMP

Note that data of type TIME and TIMESTAMP can also be expressed as Oracle DATE data.

OBJ User-Defined Types

User-defined datatypes use Oracle built-in datatypes and other user-defined datatypes as the building blocks of types that model the structure and behavior of data in applications. For information about Oracle built-in datatypes, see *Oracle8 Concepts*. For information about creating user-defined types, see CREATE TYPE on page 4-345 and the CREATE TYPE BODY on page 4-353. For information about using user-defined types, see *Oracle8 Application Developer's Guide*.

Object Types

Object types are abstractions of the real-world entities—for example, 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 and stored externally. Methods implement operations the application can perform on the real-world entity.

REFs

An object identifier (OID) uniquely identifies an object and enables you to reference the object from other objects or from relational tables. A built-in datatype called REF represents such references. A REF is a container for an object identifier. REFs are pointers to objects.

When a REF value points to a nonexistent object, the REF is said to be DANGLING. DANGLING is different from being NULL. To check to see if a REF is dangling or not, use the predicate IS [NOT] DANGLING. For example, given table DEPT with column MGR whose type is a REF to type EMP_T:

```
SELECT t.mgr.name
FROM dept t
WHERE t.mgr IS NOT DANGLING;
```

VARRAYs

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

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

- the datatype of a column of a relational table
- an object type attribute

- a PL/SQL variable, parameter, or function return type

> **Note:** You cannot specify VARRAY as a column datatype in an index-organized table.

An array object may be stored in line (that is, in the same tablespace as the other data in its row) or out of line, depending on its size.

Nested Tables

A nested table type is used to model 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

- columns of a relational table
- object type attributes
- PL/SQL variables, parameters, and function return values

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.

Datatype Comparison Rules

This section describes how Oracle compares values of each datatype.

Number 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.

Date Values

A later date is considered greater than an earlier one. For example, the date equivalent of '29-MAR-1991' is less than that of '05-JAN-1992' and '05-JAN-1992 1:35pm' is greater than '05-JAN-1992 10:09am'.

Character String Values

Character values are compared using one of these comparison rules:

- blank-padded comparison semantics
- nonpadded comparison semantics

The following sections explain these comparison semantics. The results of comparing two character values using different comparison semantics may be different. Table 2–5 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.

Table 2–5 Comparisons with Blank-Padded and Nonpadded Comparison Semantics

Blank-Padded	Nonpadded
'ab' > 'aa'	'ab' > 'aa'
'ab' > 'a '	'ab' > 'a '
'ab' > 'a'	'ab' > 'a'
'ab' = 'ab'	'ab' = 'ab'
'a ' = 'a'	**'a ' > 'a'**

Blank-Padded Comparison Semantics If the two values have different lengths, 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 datatype CHAR, NCHAR, text literals, or values returned by the USER function.

Nonpadded Comparison 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, 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 datatype VARCHAR2 or NVARCHAR2.

Single Characters

Oracle compares single characters according to their numeric values in the database character set. One character is greater than another if it has a greater numeric value than the other in the character set. Oracle considers blanks to be less than any character, which is true in most character sets.

These are some common character sets:

- 7-bit ASCII (American Standard Code for Information Interchange)
- EBCDIC Code (Extended Binary Coded Decimal Interchange Code) Page 500
- ISO 8859 / 1 (International Standards Organization)
- JEUC Japan Extended UNIX

Portions of the ASCII and EBCDIC character sets appear in Table 2–6 and Table 2–7. Note that uppercase and lowercase letters are not equivalent. Also, note that the numeric values for the characters of a character set may not match the linguistic sequence for a particular language.

Table 2–6 ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	32	;	59
!	33	<	60
″	34	=	61
#	35	>	62
$	36	?	63
%	37	@	64
&	38	A-Z	65-90
′	39	[	91
(	40	\	92
)	41	]	93
*	42	^^	94
+	43	_	95
,	44	‘	96
-	45	a-z	97-122

Table 2–6 (Cont.) ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
.	46	{	123
/	47	|	124
0-9	48-57	}	125
:	58	~	126

Table 2–7 EBCDIC Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	64	%	108
¢	74	_	109
.	75	>	110
<	76	?	111
(	77	:	122
+	78	#	123
|	79	@	124
&	80	’	125
!	90	=	126
$	91	”	127
*	92	a-i	129-137
)	93	j-r	145-153
;	94	s-z	162-169
ÿ	95	A-I	193-201
-	96	J-R	209-217
/	97	S-Z	226-233

OBJ 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 the object type.

A MAP function takes a single object as an argument and returns a scalar value. To compare two objects, the MAP method is applied to each of the objects individually. Then the results are compared. MAP is like a hash function that maps object types into scalars.

An ORDER function takes two objects (object1 and object2, for example) and simply compares one object value to the other. ORDER returns the result +1 if object1 is greater than object2, 0 if equal to, or -1 if less than. Order methods cannot return a NULL value.

Use MAP if you are performing extensive sorting or hash join operations on object instances. MAP is applied once to map the objects to scalar values and then the scalars are used during sorting and merging. A MAP method is more efficient than an ORDER method, which must invoke the method for each object comparison. You must use a MAP method for hash joins and cannot use an ORDER method because the hash mechanism hashes on the object value.

An object specification can contain only one comparison method, which must be a function. You can define either MAP method or ORDER method in a type specification, but not both.

No comparison method needs to be specified to determine the equality of two object types.

See CREATE TYPE on page 4-345 and the *Oracle8 Application Developer's Guide* for more information.

VARRAYs and Nested Tables

You cannot compare VARRAYs and nested tables in Oracle8.

Data Conversion

Generally an expression cannot contain values of different datatypes. 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 datatype to another.

Implicit Data Conversion

Oracle automatically converts a value from one datatype to another when such a conversion makes sense. Oracle performs conversions in these cases:

- When an INSERT or UPDATE statement assigns a value of one datatype to a column of another, Oracle converts the value to the datatype of the column.

- When you use a SQL function or operator with an argument with a datatype other than the one it accepts, Oracle converts the argument to the accepted datatype.
- When you use a comparison operator on values of different datatypes, Oracle converts one of the expressions to the datatype of the other.

Example 1 The text literal '10' has datatype CHAR. Oracle implicitly converts it to the NUMBER datatype if it appears in a numeric expression as in the following statement:

```
SELECT sal + '10'
    FROM emp;
```

Example 2 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 '7936' to 7936:

```
SELECT ename
    FROM emp
    WHERE empno = '7936';
```

Example 3 In the following statement, Oracle implicitly converts '12-MAR-1993' to a DATE value using the default date format 'DD-MON-YYYY':

```
SELECT ename
    FROM emp
    WHERE hiredate = '12-MAR-1993';
```

Example 4 In the following statement, Oracle implicitly converts the text literal 'AAAAZ8AABAAABvlAAA' to a ROWID value:

```
SELECT ename
    FROM emp
    WHERE ROWID = 'AAAAZ8AABAAABvlAAA';
```

Explicit Data Conversion

You can also explicitly specify datatype conversions using SQL conversion functions. Table 2–8 shows SQL functions that explicitly convert a value from one datatype to another.

Table 2–8 SQL Functions for Datatype Conversion

TO: / FROM:	CHAR	NUMBER	DATE	RAW	ROWID
CHAR	–	TO_NUMBER	TO_DATE	HEXTORAW	CHARTOROWID
NUMBER	TO_CHAR	–	TO_DATE (number,'J')		
DATE	TO_CHAR	TO_CHAR (date,'J')	–		
RAW	RAWTOHEX			–	
ROWID	ROWIDTOCHAR				–

For information on these functions, see "Conversion Functions" on page 3-42.

Note: Table 2–8 does not show conversions from LONG and LONG RAW values because it is impossible to specify LONG and LONG RAW values in cases in which Oracle can perform implicit datatype conversion. For example, LONG and LONG RAW values cannot appear in expressions with functions or operators. For information on the limitations on LONG and LONG RAW datatypes, see "LONG Datatype" on page 2-12.

Implicit vs. 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 datatype conversions functions.
- Automatic datatype conversion can have a negative impact on performance, especially if the datatype 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.

- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.

Nulls

If a column in a row has no value, then column is said to be *null*, or to contain a null. Nulls can appear in columns of any datatype 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.

Do not use null to represent a value of zero, because they are not equivalent. (Oracle 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

All scalar functions (except NVL and TRANSLATE) return null when given a null argument. The NVL function can be used to return a value when a null occurs. For example, the expression NVL(COMM,0) returns 0 if COMM is null or the value of COMM if it is not null.

Most group functions ignore nulls. For example, consider a query that averages the five values 1000, null, null, null, and 2000. Such a query ignores the nulls and calculates the average to be (1000+2000) / 2 = 1500.

Nulls with Comparison Operators

To test for nulls, use only the comparison operators IS NULL and IS NOT NULL. If you use any other operator with nulls and the result depends on the value of the null, 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, note that Oracle considers two nulls to be equal when evaluating a DECODE expression. For information on the DECODE syntax, see “Expressions” on page 3-78.

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 2–9 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 2–9 Conditions Containing Nulls

If A is:	Condition	Evaluates to:
10	a IS NULL	FALSE
10	a IS NOT NULL	TRUE
NULL	a IS NULL	TRUE
NULL	a IS NOT NULL	FALSE
10	a = NULL	UNKNOWN
10	a != NULL	UNKNOWN
NULL	a = NULL	UNKNOWN
NULL	a != NULL	UNKNOWN
NULL	a = 10	UNKNOWN
NULL	a != 10	UNKNOWN

For the truth tables showing the results of logical expressions containing nulls, see Table 3–6 on page 3-12, as well as Table 3–7 and Table 3–8.

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. This section describes these pseudocolumns:

- CURRVAL and NEXTVAL
- LEVEL

- ROWID
- ROWNUM

CURRVAL and NEXTVAL

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
```

For more information on referring to database links, see “Referring to Objects in Remote Databases” on page 2-54.

If you are using Trusted Oracle in DBMS MAC mode, you can refer to a sequence only if your DBMS label dominates the sequence’s creation label or if one of these criteria is satisfied:

- If the sequence’s creation label is higher than your DBMS label, you must have READUP and WRITEUP system privileges.
- If the sequence’s creation label and your DBMS label are not comparable, you must have READUP, WRITEUP, and WRITEDOWN system privileges.

If you are using Trusted Oracle in OS MAC mode, you cannot refer to a sequence with a lower creation label than your DBMS label.

Where to Use Sequence Values

You can use CURRVAL and NEXTVAL in these places:

- the SELECT list of a SELECT statement that is not contained in a subquery, snapshot, 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

You *cannot* use CURRVAL and NEXTVAL in these places:

- a subquery in a DELETE, SELECT, or UPDATE statement
- a view's query or snapshot's query
- a SELECT statement with the DISTINCT operator
- a SELECT statement with a GROUP BY 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
- DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- the condition of a CHECK constraint

Also, within a single SQL statement that uses CURVAL 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 sequence's initial value. Subsequent references to NEXTVAL increment the sequence value by the defined increment and return the new value. Any reference to CURRVAL always returns the sequence's current value, which is the value returned by the last reference to NEXTVAL. Note that before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL.

Within a single SQL statement, Oracle will increment the sequence only once. If a statement contains more than one reference to NEXTVAL for a sequence, Oracle increments the sequence once and returns the same value for all occurrences of NEXTVAL. If a statement contains references to both CURRVAL and NEXTVAL, Oracle increments the sequence and returns the same value for both CURRVAL and NEXTVAL regardless of their order within the statement.

A sequence can be accessed by many users concurrently with no waiting or locking. For information on sequences, see CREATE SEQUENCE on page 4-281.

Example 1 This example selects the current value of the employee sequence:

```
SELECT empseq.currval
    FROM DUAL;
```

Example 2 This example increments the employee sequence and uses its value for a new employee inserted into the employee table:

```
INSERT INTO emp
    VALUES (empseq.nextval, 'LEWIS', 'CLERK',
              7902, SYSDATE, 1200, NULL, 20);
```

Example 3 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:

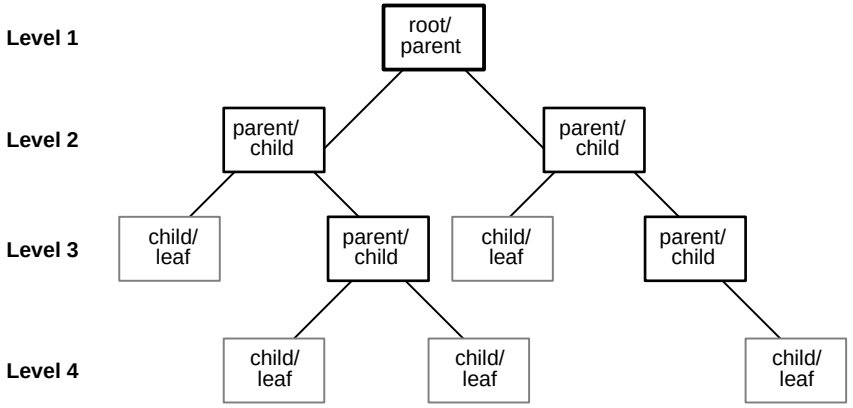
```
INSERT INTO master_order(orderno, customer, orderdate)
    VALUES (orderseq.nextval, 'Al''s Auto Shop', SYSDATE);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'SPARKPLUG', 4);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'FUEL PUMP', 1);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'TAILPIPE', 2);
```

LEVEL

For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root node, 2 for a child of a root, and so on. A *root node* is the highest node within an inverted tree. A *child node* is any nonroot node. A *parent node* is any node

that has children. A *leaf node* is any node without children. Figure 2–1 shows the nodes of an inverted tree with their LEVEL values.

Figure 2–1 Hierarchical Tree



To define a hierarchical relationship in a query, you must use the START WITH and CONNECT BY clauses. For more information on using the LEVEL pseudocolumn, see SELECT on page 4-489.

ROWID

For each row in the database, the ROWID pseudocolumn returns a row's address. Oracle8 ROWID values contain information necessary to locate a row:

- the data object number of the object
- which data block in the datafile
- which row in the data block (first row is 0)
- which datafile (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 datatype ROWID. For information on the ROWID datatype, see "ROWID Datatype" on page 2-18.

ROWID values have several important uses:

- They are the fastest way to access a single row.

- They can show you how a table's rows are stored.
- They are unique identifiers for rows in a table.

You should not use ROWID as a table's primary key. If you delete and reinsert a row with the Import and Export utilities, for example, its ROWID may change. If you delete a row, Oracle may reassign its ROWID to a new row inserted later.

Although you can use the ROWID pseudocolumn in the SELECT and WHERE clauses 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, ename
    FROM emp
    WHERE deptno = 20;

ROWID              ENAME
------------------ ----------
AAAAfSAABAAAClaAAA SMITH
AAAAfSAABAAAClaAAD JONES
AAAAfSAABAAAClaAAH SCOTT
AAAAfSAABAAAClaAAK ADAMS
AAAAfSAABAAAClaAAM FORD
```

ROWNUM

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 emp
    WHERE ROWNUM < 10;
```

Note that conditions testing for ROWNUM values *greater than* a positive integer are always false. For example, this query returns no rows:

```
SELECT * FROM emp
    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 tabx
    SET col1 = ROWNUM;
```

Oracle assigns a ROWNUM value to each row as it is retrieved, before rows are sorted for an ORDER BY clause, so an ORDER BY clause normally does not affect the ROWNUM of each row. However, if an ORDER BY clause causes Oracle to use an index to access the data, Oracle may retrieve the rows in a different order than without the index, so the ROWNUMs may be different than they would be without the ORDER BY clause.

Note: Using ROWNUM in a query can affect view optimization. For more information, see *Oracle8 Concepts*.

Comments

You can associate comments with SQL statements and schema objects.

Comments Within SQL Statements

Comments within SQL statements do not affect the statement execution, but they may make your application easier for you to read and maintain. You may want to include a comment in a statement that describes the statement's purpose within your application.

A comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement using either of these means:

- Begin the comment with /*. Proceed with the text of the comment. This text can span multiple lines. End the comment with */. 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.

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.

> **Note:** You cannot use these styles of comments between SQL statements in a SQL script. Use the Server Manager or SQL*Plus REMARK command for this purpose. For information on these commands, see *Oracle Server Manager User's Guide* or *SQL*Plus User's Guide and Reference.*

Example These statements contain many comments:

```
SELECT ename, sal + NVL(comm, 0), job, loc
/* Select all employees whose compensation is
greater than that of Jones.*/
  FROM emp, dept
       /*The DEPT table is used to get the department name.*/
  WHERE emp.deptno = dept.deptno
    AND sal + NVL(comm,0) >   /* Subquery:       */
   (SELECT sal + NLV(comm,0)
                             /* total compensation is sal + comm */
      FROM emp
      WHERE ename = 'JONES')

SELECT ename,                     -- select the name
    sal + NVL(comm, 0),           -- total compensation
    job,                          -- job
    loc                           -- and city containing the office
  FROM emp,                       -- of all employees
       dept
  WHERE emp.deptno = dept.deptno
    AND sal + NVL(comm, 0) >      -- whose compensation
                                  -- is greater than
      (SELECT sal + NVL(comm,0)   -- the compensation
    FROM emp
    WHERE ename = 'JONES')        -- of Jones.
```

Comments on Schema Objects

You can associate a comment with a table, view, snapshot, or column using the COMMENT command described in Chapter 4, "Commands". Comments associated with schema objects are stored in the data dictionary.

Hints

You can use comments in a SQL statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses these hints to choose an execution plan for the statement.

A statement block can have only one comment containing hints, and that comment must follow the SELECT, UPDATE, INSERT, or DELETE keyword. The syntax below shows the syntax for hints contained in both styles of comments that Oracle supports within a statement block.

```
{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */
```

or

```
{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...
```

where:

DELETE INSERT SELECT UPDATE	is a DELETE, INSERT, SELECT, or UPDATE keyword that begins a statement block. Comments containing hints can appear only after these keywords.
+	is a plus sign that 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 and in *Oracle8 Tuning*. The space between the plus sign and the hint is optional. If the comment contains multiple hints, each pair of hints must be separated by at least one space.
text	is other commenting text that can be interspersed with the hints.

Table 2–10 lists hint syntax and descriptions. For more information on hints, see *Oracle8 Tuning, Oracle8 Parallel Server Concepts and Administration,* and *Oracle8 Concepts.*

Table 2–10 Hint Syntax and Descriptions

Hint Syntax	Description
Optimization Approaches and Goals	
`/*+ ALL_ROWS */`	explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption)
`/*+ CHOOSE */`	causes the optimizer to choose between the rule-based approach and the cost-based approach for a SQL statement based on the presence of statistics for the tables accessed by the statement
`/*+ FIRST_ROWS */`	explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row)
`/*+ RULE */`	explicitly chooses rule-based optimization for a statement block
Access Methods	
`/*+ AND_EQUAL(table index) */`	explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes
`/*+ CLUSTER(table) */`	explicitly chooses a cluster scan to access the specified table
`/*+ FULL(table) */`	explicitly chooses a full table scan for the specified table
`/*+ HASH(table) */`	explicitly chooses a hash scan to access the specified table
`/*+ HASH_AJ(table) */`	transforms a NOT IN subquery into a hash antijoin to access the specified table
`/*+ HASH_SJ (table) */`	transforms a NOT IN subquery into a hash anti-join to access the specified table
`/*+ INDEX(table index) */`	explicitly chooses an index scan for the specified table
`/*+ INDEX_ASC(table index) */`	explicitly chooses an ascending-range index scan for the specified table

Table 2–10 (Cont.) Hint Syntax and Descriptions

Hint Syntax	Description
`/*+ INDEX_COMBINE(table index) */`	If no indexes are given as arguments for the INDEX_COMBINE hint, the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate. If particular indexes are given as arguments, the optimizer tries to use some Boolean combination of those particular bitmap indexes.
`/*+ INDEX_DESC(table index) */`	explicitly chooses a descending-range index scan for the specified table
`/*+ INDEX_FFS(table index) */`	causes a fast full index scan to be performed rather than a full table scan
`/*+ MERGE_AJ (table) */`	transforms a NOT IN subquery into a merge anti-join to access the specified table
`/*+ MERGE_SJ (table) */`	transforms a correlated EXISTS subquery into a merge semi-join to access the specified table
`/*+ ROWID(table) */`	explicitly chooses a table scan by ROWID for the specified table
`/*+ USE_CONCAT */`	forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator
Join Orders	
`/*+ ORDERED */`	causes Oracle to join tables in the order in which they appear in the FROM clause
`/*+ STAR */`	forces the large table to be joined last using a nested-loops join on the index
Join Operations	
`/*+ DRIVING_SITE (table) */`	forces query execution to be done at a different site from that selected by Oracle
`/*+ USE_HASH (table) */`	causes Oracle to join each specified table with another row source with a hash join
`/*+ USE_MERGE (table) */`	causes Oracle to join each specified table with another row source with a sort-merge join
`/*+ USE_NL (table) */`	causes Oracle to join each specified table to another row source with a nested-loops join using the specified table as the inner table

Table 2–10 (Cont.) Hint Syntax and Descriptions

Hint Syntax	Description
Parallel Execution	
`/*+ APPEND */`	specifies that data is simply appended (or not) to a table; existing free space is not used. Use these hints only following the INSERT keyword.
`/*+ NOAPPEND */`	
`/*+ NOPARALLEL(table) */`	disables parallel scanning of a table, even if the table was created with a PARALLEL clause
`/*+ PARALLEL(table, instances) */`	allows you to specify the desired number of concurrent slave processes that can be used for the operation.
	DELETE, INSERT, and UPDATE operations are considered for parallelization only if the session is in a PARALLEL DML enabled mode. (Use ALTER SESSION PARALLEL DML to enter this mode.)
`/*+ PARALLEL_INDEX`	allows you to parallelize fast full index scan for partitioned and nonpartitioned indexes that have the PARALLEL attribute
`/*+ NOPARALLEL_INDEX */`	overrides a PARALLEL attribute setting on an index
Other Hints	
`/*+ CACHE */`	specifies that the blocks retrieved for the table in the hint are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed
`/*+ NOCACHE */`	specifies 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
`/*+ MERGE (table) */`	causes Oracle to evaluate complex views or subqueries before the surrounding query
`/*+ NO_MERGE (table) */`	causes Oracle not to merge mergeable views
`/*+ PUSH_JOIN_PRED (table) */`	causes the optimizer to evaluate, on a cost basis, whether or not to push individual join predicates into the view
`/*+ NO_PUSH_JOIN_PRED (table) */`	Prevents pushing of a join predicate into the view
`/*+ PUSH_SUBQ */`	causes nonmerged subqueries to be evaluated at the earliest possible place in the execution plan
`/*+ STAR_TRANSFORMATION */`	makes the optimizer use the best plan in which the transformation has been used.

Database Objects

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:

- clusters
- database links
- database triggers
- external procedure libraries
- index-only tables
- indexes
- packages
- sequences
- stored functions
- stored procedures
- synonyms
- tables
- views

In addition, the following schema objects are available if you are using Oracle's distributed functionality:

- snapshots
- snapshot logs

In addition, the following schema objects are available if the objects option is installed on your database server:

- object tables
- object types
- object views

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:

- directories
- profiles
- roles
- rollback segments
- tablespaces
- users

In this reference, each type of object is briefly defined in Chapter 4, "Commands", in the section describing the command that creates the database object. These commands begin with the keyword CREATE. For example, for the definition of a cluster, see CREATE CLUSTER on page 4-207. For an overview of database objects, see *Oracle8 Concepts*.

You must provide names for most types of schema objects when you create them. These names must follow the rules listed in the following sections.

Parts of Schema Objects

Some schema objects are made up of parts that you must name, such as:

- columns in a table or view
- index and table partitions
- integrity constraints on a table
- packaged procedures, packaged stored functions, and other objects stored within a package

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 Table Names

Partitions can be used as tables. This capability provides you with a shortcut for performing some partition-level operations that would otherwise require using a WHERE clause.

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. The advantage of this method 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.

Partition-level bulk operations, such as deleting all of the rows from a partition, restrict the operation to one partition. These types of operations are easily expressed with the partition-extended table name syntax. Trying to phrase the same operation with a WHERE clause predicate can be cumbersome, especially when the range partitioning key uses more than one column.

Table specification syntax has been extended for the following DML statements to allow an optional partition specification for nonremote partitioned tables:

- DELETE
- INSERT
- LOCK TABLE
- SELECT
- UPDATE

Note: For application portability and ANSI syntax compliance, Oracle strongly recommends that you use views to insulate applications from this Oracle proprietary extension.

Currently, the use of partition-extended table names has the following restrictions:

- No remote tables: A partition-extended table name cannot contain a database link (dblink) or a synonym that translates to a table with a dblink. To use remote partitions, create a view at the remote site that uses the partition-extended table name syntax and then refer to the remote view.
- No direct PL/SQL support: A SQL statement using the partition-extended table name syntax cannot be used in a PL/SQL block, although it can be used through dynamic SQL by using the DBMS_SQL package. To refer to a partition within a PL/SQL block, use views that in turn use the partition-extended table name syntax.

- No synonyms: A partition extension must be specified with a base table. You cannot use synonyms, views, or any other objects.

Syntax The basic syntax for using partition-extended table names is:

```
[schema.]{table | view} [@dblink | PARTITION (partition_name)]
```

Example In the following statement, SALES is a partitioned table with partition JAN97. You can create a view of the single partition JAN97, and then use it as if it were a table. This example deletes rows from the partition.

```
CREATE VIEW sales_jan97 AS
     SELECT * FROM sales PARTITION (jan97);
DELETE FROM sales_jan97 WHERE amount < 0;
```

Schema Object Names and Qualifiers

This section provides:

- rules for naming schema objects and schema object location qualifiers
- guidelines for naming schema objects and qualifiers

Schema Object Naming Rules

The following rules apply when naming schema objects:

1. Names must be from 1 to 30 characters long with these exceptions:
   - Names of databases are limited to 8 characters.
   - Names of database links can be as long as 128 characters.
2. Names cannot contain quotation marks.
3. Names are not case sensitive.
4. A name must begin with an alphabetic character from your database character set unless surrounded by double quotation marks.
5. Names can contain only alphanumeric characters from your database character set and the characters _, $, and #. Names of database links can also contain periods (.) and at signs (@). Oracle strongly discourages you from using $ and #.

   If your database character set contains multibyte characters, Oracle recommends that each name for a user or a role contain at least one single-byte character.

> **Note:** You cannot use special characters from European or Asian character sets in a database name, global database name, or database link names. For example, the umlaut is not allowed.

6. A name cannot be an Oracle reserved word. Appendix C, "Oracle Reserved Words and Keywords", lists all Oracle reserved words.

   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. For a list of a product's reserved words, see the manual for the specific product, such as *PL/SQL User's Guide and Reference*.

7. Do not use the word DUAL as a name for an object or part. DUAL is the name of a dummy table.

8. The Oracle SQL language contains other keywords that have special meanings. Because these keywords are not reserved, you can also use them as names for objects and object parts. However, using them as names may make your SQL statements more difficult to read.

   Appendix C, "Oracle Reserved Words and Keywords" lists Oracle keywords.

9. Within a namespace, no two objects can have the same name.

   Figure 2–2 shows the namespaces for schema objects; each box is a namespace. Tables and views are in the same namespace; therefore, a table and a view 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.

Figure 2–2 Namespaces for Schema Objects

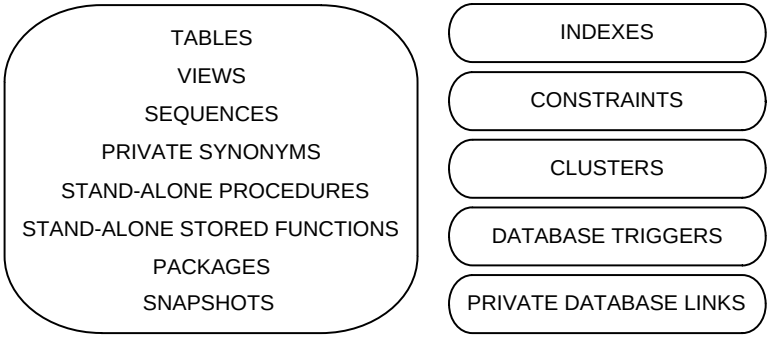
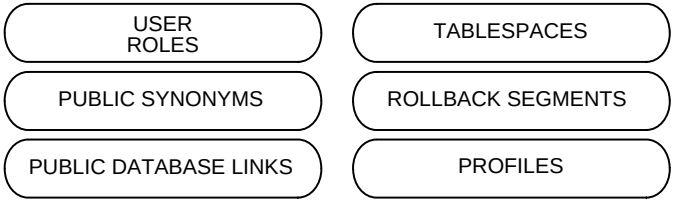


Figure 2–3 shows the namespaces for nonschema objects. Because the objects in these namespaces are not contained in schemas, these namespaces span the entire database.

Figure 2–3 Namespaces for Nonschema Objects



10. Columns in the same table or view cannot have the same name. However, columns in different tables or views can have the same name.

11. Procedures or functions contained in the same package can have the same name, provided that their arguments are not of the same number and datatypes. Creating multiple procedures or functions with the same name in the same package with different arguments is called *overloading* the procedure or function.

12. A name can be enclosed in double quotation marks. Such names can contain any combination of characters, including spaces, ignoring rules 3 through 7 in this list. This exception is allowed for portability, but Oracle recommends that you do not break rules 3 through 7.

If you give a schema object a name enclosed in double quotation marks, you must use double quotation marks whenever you refer to the object.

Enclosing a name in double quotes allows it to

- contain spaces
- be case sensitive
- begin with a character other than an alphabetic character, such as a numeric character
- contain characters other than alphanumeric characters and _, $, and #
- be a reserved word

By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

```
emp
"emp"
"Emp"
"EMP "
```

Note that Oracle interprets the following names the same, so they cannot be used for different objects in the same namespace:

```
emp
EMP
"EMP"
```

If you give a user or password a quoted name, the name cannot contain lowercase letters.

Database link names cannot be quoted.

Schema Object Naming Examples

The following examples are valid schema object names:

```
ename
horse
scott.hiredate
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

Although column aliases, table aliases, usernames, and passwords are not objects or parts of objects, they must also follow these naming rules with these exceptions:

- Column aliases and table aliases exist only for the execution of a single SQL statement and are not stored in the database, so rule 12 does not apply to them.
- Passwords do not have namespaces, so rule 9 does not apply to apply to them.
- Do not use quotation marks to make usernames and passwords case sensitive. For additional rules for naming users and passwords, see CREATE USER on page 4-357.

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 database 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 EMP and DEPT tables are both named DEPTNO.

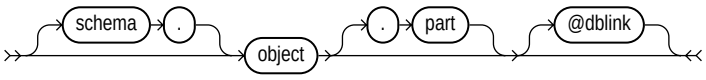
Referring to Schema Objects and Parts

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

The following diagram shows the general syntax for referring to an object or a part:

schema_object ::=



where:

object is the name of the object.

schema is the schema containing the object. The schema qualifier allows you to refer to an object in a schema other than your own. Note that you must be granted privileges to refer to objects in other schemas. If you omit *schema*, 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 in Figure 2–2 on page 2-49. Nonschema objects, shown in Figure 2–3 on page 2-49, 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 allows you to refer to a part of a schema object, such as a column or a partition of a table. Note that not all types of objects have parts.

dblink applies only when you are using Oracle's distributed functionality. This is the name of the database containing the object. The *dblink* qualifier allows you to refer to an object in a database other than your local database. If you omit *dblink*, Oracle assumes that you are referring to an object in your local database. Note that 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 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 statement's operation on the object. If the named object cannot be found in the appropriate namespace, Oracle returns an error message.

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 DEPT:

```
INSERT INTO dept
   VALUES (50, 'SUPPORT', 'PARIS');
```

Based on the context of the statement, Oracle determines that DEPT 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 DEPT 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, 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, Oracle attempts to perform the statement on the object. In this example, Oracle attempts to add the row of data to DEPT. If the object is not of the correct type for the statement, Oracle returns an error message. In this example, DEPT must be a table, view, or a private synonym resolving to a table or view. If DEPT is a sequence, Oracle returns an error message.
3. If the object is not in any namespace searched in thus far, Oracle searches the namespace containing public synonyms (see Figure 2–3 on page 2-49). If the object is in that namespace, Oracle attempts to perform the statement on it. If the object is not of the correct type for the statement, Oracle returns an error message. In this example, if DEPT is a public synonym for a sequence, Oracle returns an error message.

Referring 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 EMP table in the schema SCOTT:

```
DROP TABLE scott.emp
```

Referring 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 CREATE DATABASE LINK command described in Chapter 4, "Commands". The command allows you to specify this information about the database link:

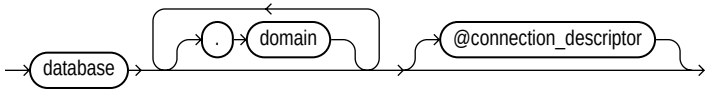
- the name of the database link
- the database connect string to access the remote database
- the username and password to connect to the remote database

Oracle stores this information in the data dictionary.

Database Link Names When you create a database link, you must specify its name. The name of a database link can be as long as 128 bytes and can contain periods (.) and the "at" sign (@). In these ways, database link names are different from names of other types of objects.

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	specifies the name of the remote database to which the database link connects. The name of the remote database is specified by its initialization parameter DB_NAME.
domain	specifies the domain of the remote database to which the database link connects. If you omit the domains from the name of a database link, Oracle expands the name by qualifying the database with the domain of your local database as it currently exists in the data dictionary, and then stores the link name in the data dictionary.
connect_descriptor	allows you to further qualify a database link. Using connect descriptors, you can create multiple database links to the same database. For example, you can use connect descriptors to create multiple database links to different instances of the Oracle Parallel Server that access the same database.

The combination *database.domain* is sometimes called the “service name”. For more information, see *Net8 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 Net8 to access the remote database. For information on writing database connect strings, see the Net8 documentation for your specific network protocol. The database string for a database link is optional.

Referring to Database Links

Database links are available only if you are using Oracle's 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:

complete is the complete database link name as stored in the data dictionary, including the *database*, *domain*, and optional *connect_descriptor* components.

partial is the *database* and optional *connect_descriptor* 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, 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, Oracle uses it. If it does not have an associated username and password, Oracle uses your current username and password.
   - If the first matching database link has an associated database string, Oracle uses it. If not, 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, Oracle returns an error message.
3. Oracle uses the database string to access the remote database. After accessing the remote database, Oracle verifies that both of these conditions are true:
   - The name of the remote database (specified by its initialization parameter DB_NAME) matches the *database* component of the database link name.
   - The domain (specified by the initialization parameter DB_DOMAIN) of the remote database matches the *domain* component of the database link name.

   If both of these conditions are true, Oracle proceeds with the connection, using the username and password chosen in Step 2. If not, Oracle returns an error message.

4. If the connection using the database string, username, and password is successful, 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 enable and disable Oracle resolution of names for remote objects using the initialization parameter GLOBAL_NAMES or the GLOBAL_NAMES parameter of the ALTER SYSTEM or ALTER SESSION command.

For more information on remote name resolution, see *Oracle8 Distributed Database Systems*.

Referencing Object Type Attributes and Methods

To reference object type attributes or methods in a SQL statement, you must fully qualify the reference with a table alias. For example, consider the following example:

```
CREATE TYPE person AS OBJECT
   (ssno VARCHAR(20),
    name VARCHAR (10));

CREATE TABLE emptab (pinfo person);
```

In a SQL statement, reference to the SSNO attribute must be fully qualified using a table alias, as illustrated below:

```
SELECT e.pinfo.ssno FROM emptab e;

UPDATE emptab e SET e.pinfo.ssno = '510129980'
   WHERE e.pinfo.name = 'Mike';
```

To reference an object type's member method that does not accept any arguments, you must provide "empty" parentheses. For example, assume that AGE is a method in the person type that does not take any arguments. In order to call this method in a SQL statement, you must provide empty parentheses as shows in this example:

```
SELECT e.pinfo.age() FROM emptab e
   WHERE e.pinfo.name = 'Mike';
```

For more information, see the sections on user-defined datatypes in *Oracle8 Concepts*.

3

Operators, Functions, Expressions, Conditions

This chapter describes methods of manipulating individual data items. Standard arithmetic operators such as addition and subtraction are discussed, as well as less common functions such as absolute value and string length. Topics include:

- Operators
- SQL Functions
- User Functions
- Format Models
- Expressions
- Conditions

> **Note:** Functions, expressions, and descriptions preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Operators

An operator manipulates individual data items and returns a result. The data items are called *operands* or *arguments*. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL. Tables in this section list SQL operators.

Unary and Binary Operators

There are two general classes of operators:

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, the result is always null. The only operator that does not follow this rule is concatenation (||).

Precedence

Precedence is the order in which Oracle 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 3–1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 3–1 SQL Operator Precedence

Operator	Operation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	exponentiation, logical negation
AND	conjunction
OR	disjunction

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.

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. Table 3–2 lists arithmetic operators.

Table 3–2 Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM orders WHERE qtysold = -1;` `SELECT * FROM emp WHERE -sal < 0;`
* /	Multiplies, divides. These are binary operators.	`UPDATE emp SET sal = sal * 1.1;`
+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp WHERE SYSDATE - hiredate > 365;`

Do not use two consecutive minus signs with no separation (--) 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 a parenthesis. For more information on comments within SQL statements, see “Comments” on page 2-38.

Concatenation Operator

The concatenation operator manipulates character strings. Table 3–3 describes the concatenation operator.

Table 3–3 Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is ' || ename FROM emp;`

The result of concatenating two character strings is another character string. If both character strings are of datatype CHAR, the result has datatype CHAR and is limited to 2000 characters. If either string is of datatype VARCHAR2, the result has datatype VARCHAR2 and is limited to 4000 characters. Trailing blanks in character strings are preserved by concatenation, regardless of the strings' datatypes. For more information on the differences between the CHAR and VARCHAR2 datatypes, see "Character Datatypes" on page 2-7.

On most platforms, the concatenation operator is two solid vertical bars, as shown in Table 3–3. 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 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. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

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, concatenating 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) );
```

```
Table created.

INSERT INTO tab1 (col1,  col2,     col3,     col4)
       VALUES   ('abc', 'def   ', 'ghi   ', 'jkl');

1 row created.

SELECT col1||col2||col3||col4 "Concatenation"
       FROM tab1;

Concatenation
------------------------
abcdef   ghi   jkl
```

Comparison Operators

Comparison operators compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN. For information on conditions, see "Conditions" on page 3-90. Table 3–4 lists comparison operators.

Table 3–4 Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500;`
!= ^= < > ¬=	Inequality test. Some forms of the inequality operator may be unavailable on some platforms.	`SELECT * FROM emp WHERE sal != 1500;`
>	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500;`
<		`SELECT * FROM emp WHERE sal < 1500;`
>=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500;`
<=		`SELECT * FROM emp WHERE sal <= 1500;`

Table 3–4 (Cont.) Comparison Operators

Operator	Purpose	Example
`IN`	"Equal to any member of" test. Equivalent to "`= ANY`".	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST');` `SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30);`
`NOT IN`	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	`SELECT * FROM emp WHERE sal NOT IN (SELECT sal FROM emp WHERE deptno = 30);` `SELECT * FROM emp WHERE job NOT IN ('CLERK', ANALYST');`
`ANY` `SOME`	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.	`SELECT * FROM emp WHERE sal = ANY (SELECT sal FROM emp WHERE deptno = 30);`
`ALL`	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to TRUE if the query returns no rows.	`SELECT * FROM emp WHERE sal >= ALL ( 1400, 3000);`
`[NOT] BETWEEN x AND y`	[Not] greater than or equal to *x* and less than or equal to *y*.	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000;`
`EXISTS`	TRUE if a subquery returns at least one row.	`SELECT ename, deptno FROM dept WHERE EXISTS (SELECT * FROM emp WHERE dept.deptno = emp.deptno);`

Table 3–4 (Cont.) Comparison Operators

Operator	Purpose	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, excepting percent (%) and underbar (_) may follow ESCAPE; a wildcard character is treated as a literal if preceded by the character designated as the escape character.	See "LIKE Operator" on page 3-8. `SELECT * FROM tab1 WHERE col1 LIKE 'A_C/%E%' ESCAPE '/';`
`IS [NOT] NULL`	Tests for nulls. This is the only operator that you should use to test for nulls. See "Nulls" on page 2-31.	`SELECT ename, deptno FROM emp WHERE comm IS NULL;`

Additional information on the NOT IN and LIKE operators appears in the sections that follow.

NOT IN Operator

If any item in the list following a NOT IN operation is null, all rows evaluate to UNKNOWN (and no rows are returned). For example, the following statement returns the string 'TRUE' for each row:

```
SELECT 'TRUE'
    FROM emp
    WHERE deptno NOT IN (5,15);
```

However, the following statement returns no rows:

```
SELECT 'TRUE'
    FROM emp
    WHERE deptno NOT IN (5,15,null);
```

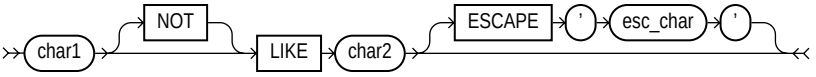
The above example returns no rows because the WHERE clause condition evaluates to:

```
deptno != 5 AND deptno != 15 AND deptno != null
```

Because all conditions that compare a null result in a null, the entire expression results in a null. This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

LIKE Operator

The LIKE operator is used in character string comparisons with pattern matching. The syntax for a condition using the LIKE operator is shown in this diagram:



where:

char1	is a value to be compared with a pattern. This value can have datatype CHAR or VARCHAR2.
NOT	logically inverts the result of the condition, returning FALSE if the condition evaluates to TRUE and TRUE if it evaluates to FALSE.
char2	is the pattern to which *char1* is compared. The pattern is a value of datatype CHAR or VARCHAR2 and can contain the special pattern matching characters % and _.
ESCAPE	identifies a single character as the escape character. The escape character can be used to cause Oracle to interpret % or _ literally, rather than as a special character, in the pattern. If you wish to search for strings containing an escape character, you must specify this character twice. For example, if the escape character is '/', to search for the string 'client/server', you must specify, 'client//server'.

While the equal (=) operator exactly matches one character value to another, the LIKE operator matches a portion of one character value to another by searching the first value for the pattern specified by the second. Note that blank padding is *not* used for LIKE comparisons.

With the LIKE operator, you can compare a value to a pattern rather than to a constant. The pattern can only appear after the LIKE keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with 'SM':

```
SELECT sal
    FROM emp
    WHERE ename LIKE 'SM%';
```

The following query uses the = operator, rather than the LIKE operator, to find the salaries of all employees with the name 'SM%':

```
SELECT sal
    FROM emp
    WHERE ename = 'SM%';
```

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 operator:

```
SELECT sal
    FROM emp
    WHERE 'SM%' LIKE ename;
```

Patterns usually use special characters that Oracle matches with different characters in the value:

- 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. Note that the pattern '%' cannot match a null.

Case Sensitivity and Pattern Matching Case is significant in all conditions comparing character expressions including the LIKE and equality (=) operators. You can use the UPPER() function to perform a case-insensitive match, as in this condition:

```
UPPER(ename) LIKE 'SM%'
```

Pattern Matching on Indexed Columns When LIKE is used to search an indexed column for a pattern, Oracle can use the index to improve the statement's performance 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 "_", the index cannot improve the query's performance because Oracle cannot scan the index.

Example 1 This condition is true for all ENAME values beginning with "MA":

```
ename LIKE 'MA%'
```

All of these ENAME values make the condition TRUE:

```
MARTIN, MA, MARK, MARY
```

Case is significant, so ENAME values beginning with "Ma," "ma," and "mA" make the condition FALSE.

Example 2 Consider this condition:

```
ename LIKE 'SMITH_'
```

This condition is true for these ENAME values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for 'SMITH', since the special character "_" must match exactly one character of the ENAME value.

ESCAPE Option You can include the actual characters "%" or "_" in the pattern by using the ESCAPE option. The ESCAPE option identifies the escape character. If the escape character appears in the pattern before the character "%" or "_" then Oracle interprets this character literally in the pattern, rather than as a special pattern matching character.

Example: To search for any employees with the pattern 'A_B' in their name:

```
SELECT ename
    FROM emp
    WHERE ename LIKE '%A\_B%' ESCAPE '\';
```

The ESCAPE option 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 % If a pattern does not contain the "%" character, the condition can be TRUE only if both operands have the same length.

Example: Consider the definition of this table and the values inserted into it:

```
CREATE TABLE freds (f CHAR(6), v VARCHAR2(6));
INSERT INTO freds VALUES ('FRED', 'FRED');
```

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.

Logical Operators

A logical operator 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 3–5 lists logical operators.

Table 3–5 Logical Operators

Operator	Function	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT * FROM emp WHERE NOT (job IS NULL);` `SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000);`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10;`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10;`

For example, in the WHERE clause of the following SELECT statement, the AND logical operator is used to ensure that only those hired before 1984 and earning more than $1000 a month are returned:

```
SELECT *
    FROM emp
    WHERE hiredate < TO_DATE('01-JAN-1984', 'DD-MON-YYYY')
        AND sal > 1000;
```

NOT Operator

Table 3–6 shows the result of applying the NOT operator to a condition.

Table 3–6 NOT Truth Table

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

AND Operator

Table 3–7 shows the results of combining two expressions with AND.

Table 3–7 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR Operator

Table 3–8 shows the results of combining two expressions with OR.

Table 3–8 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Set Operators

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table 3–9 lists SQL set operators.

Table 3–9 Set Operators

Operator	Returns
UNION	All 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.

All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order. 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.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Examples Consider these two queries and their results:

```
SELECT part
    FROM orders_list1;

PART
----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE

SELECT part
    FROM orders_list2;
```

```
PART
----------
CRANKSHAFT
TAILPIPE
TAILPIPE
```

The following examples combine the two query results with each of the set operators.

UNION Example The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows how datatype must match when columns do not exist in one or the other table:

```
SELECT part, partnum, to_date(null) date_in
    FROM orders_list1
UNION
SELECT part, to_null(null), date_in
    FROM orders_list2;

PART       PARTNUM DATE_IN
---------- ------- --------
SPARKPLUG  3323165
SPARKPLUG          10/24/98
FUEL PUMP  3323162
FUEL PUMP          12/24/99
TAILPIPE   1332999
TAILPIPE           01/01/01
CRANKSHAFT 9394991
CRANKSHAFT         09/12/02

SELECT part
    FROM orders_list1
UNION
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
TAILPIPE
CRANKSHAFT
```

UNION ALL Example The following statement combines the results with the UNION ALL operator, which does not eliminate duplicate selected rows:

```
SELECT part
    FROM orders_list1
UNION ALL
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE
CRANKSHAFT
TAILPIPE
TAILPIPE
```

Note that the UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. A PART value that appears multiple times in either or both queries (such as 'FUEL PUMP') 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 rows returned by both queries:

```
SELECT part
    FROM orders_list1
INTERSECT
SELECT part
    FROM orders_list2;

PART
----------
TAILPIPE
```

MINUS Example The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT part
    FROM orders_list1
MINUS
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
```

Other Operators

Table 3–10 lists other SQL operators.

Table 3–10 Other SQL Operators

Operator	Purpose	Example
(+)	Indicates that the preceding column is the outer join column in a join. See "Outer Joins" on page 4-508.	`SELECT ename, dname FROM emp, dept WHERE dept.deptno = emp.deptno(+);`
PRIOR	Evaluates the following expression for the parent row of the current row in a hierarchical, or tree-structured, query. In such a query, you must use this operator in the CONNECT BY clause to define the relationship between parent and child rows. You can also use this operator in other parts of a SELECT statement that performs a hierarchical query. The PRIOR operator is a unary operator and has the same precedence as the unary + and - arithmetic operators. See "Hierarchical Queries" on page 4-495.	`SELECT empno, ename, mgr FROM emp CONNECT BY PRIOR empno = mgr;`

SQL Functions

A SQL function is similar to an operator in that it manipulates data items and returns a result. SQL functions differ from operators in the format in which they appear with their arguments. This format allows them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, Oracle implicitly converts the argument to the expected datatype before performing the SQL function. See "Data Conversion" on page 2-28.

If you call a SQL function with a null argument, the SQL function automatically returns null. The only SQL functions that do not follow this rule are CONCAT, DECODE, DUMP, NVL, and REPLACE.

SQL functions should not be confused with user functions written in PL/SQL. User functions are described in "User Functions" on page 3-60.

In the syntax diagrams for SQL functions, arguments are indicated with their datatypes following the conventions described in "Syntax Diagrams and Notation" in the Preface of this reference.

SQL functions are of these general types:

- single-row (or scalar) functions
- group (or aggregate) functions

The two types of SQL functions differ in the number of rows upon which they act. A single-row function returns a single result row for every row of a queried table or view; a group function returns a single result row for a group of queried rows.

Single-row functions can appear in select lists (if the SELECT statement does not contain a GROUP BY clause), WHERE clauses, START WITH clauses, and CONNECT BY clauses.

Group functions can appear in select lists and HAVING clauses. If you use the GROUP BY clause in a SELECT statement, Oracle divides the rows of a queried table or view into groups. In a query containing a GROUP BY clause, all elements of the select list must be expressions from the GROUP BY clause, expressions containing group functions, or constants. Oracle applies the group functions in the select list to each group of rows and returns a single result row for each group.

If you omit the GROUP BY clause, Oracle applies group functions in the select list to all the rows in the queried table or view. You use group functions in the HAVING clause to eliminate groups from the output based on the results of the group functions, rather than on the values of the individual rows of the queried table or view. For more information on the GROUP BY and HAVING clauses, see the GROUP BY Clause on page 4-499 and the HAVING Clause on page 4-500.

In the sections that follow, functions are grouped by the datatypes of their arguments and return values.

Number Functions

Number functions accept numeric input and return numeric values. This section lists the SQL number functions. Most of these functions return 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.

ABS

Syntax `ABS(n)`

Purpose Returns the absolute value of *n*.

Example

```
SELECT ABS(-15) "Absolute" FROM DUAL;

  Absolute
----------
        15
```

ACOS

Syntax `ACOS(n)`

Purpose Returns the arc cosine of n. Inputs are in the range of -1 to 1, and outputs are in the range of 0 to π and are expressed in radians.

Example

```
SELECT ACOS(.3)"Arc_Cosine" FROM DUAL;

Arc_Cosine
----------
1.26610367
```

ASIN

Syntax `ASIN(n)`

Purpose Returns the arc sine of *n*. Inputs are in the range of -1 to 1, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Example

```
SELECT ASIN(.3) "Arc_Sine" FROM DUAL;

 Arc_Sine
----------
.304692654
```

ATAN

Syntax `ATAN(n)`

Purpose Returns the arc tangent of *n*. Inputs are in an unbounded range, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Syntax `ATAN(n)`

Example

```
SELECT ATAN(.3) "Arc_Tangent" FROM DUAL;

Arc_Tangent
-----------
.291456794
```

ATAN2

Syntax `ATAN2(n, m)`

Purpose Returns the arc tangent of *n* and *m*. Inputs are in an unbounded range, and outputs are in the range of -π to π, depending on the signs of n and m, and are expressed in radians. Atan2(n,m) is the same as atan2(n / m)

Example

```
SELECT ATAN2(.3, .2) "Arc_Tangent2" FROM DUAL;

Arc_Tangent2
------------
  .982793723
```

CEIL

Syntax `CEIL(n)`

Purpose Returns smallest integer greater than or equal to *n*.

Example

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;

   Ceiling
----------
        16
```

COS

Syntax `COS(n)`

Purpose Returns the cosine of *n* (an angle expressed in radians).

Example

```
SELECT COS(180 * 3.14159265359/180)
"Cosine of 180 degrees" FROM DUAL;

Cosine of 180 degrees
---------------------
                   -1
```

COSH

Syntax `COSH(n)`

Purpose Returns the hyperbolic cosine of *n*.

Example

```
SELECT COSH(0) "Hyperbolic cosine of 0" FROM DUAL;

Hyperbolic cosine of 0
----------------------
                     1
```

EXP

Syntax `EXP(n)`

Purpose Returns e raised to the *n*th power; e = 2.71828183 ...

Example

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
------------------
          54.59815
```

FLOOR

Syntax `FLOOR(n)`

Purpose Returns largest integer equal to or less than *n*.

Syntax `FLOOR(n)`

Example

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

     Floor
----------
        15
```

LN

Syntax `LN(n)`

Purpose Returns the natural logarithm of *n*, where *n* is greater than 0.

Example

```
SELECT LN(95) "Natural log of 95" FROM DUAL;

Natural log of 95
-----------------
       4.55387689
```

LOG

Syntax `LOG(m,n)`

Purpose Returns the logarithm, base *m*, of *n*. The base *m* can be any positive number other than 0 or 1 and *n* can be any positive number.

Example

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;

Log base 10 of 100
------------------
                 2
```

MOD

Syntax `MOD(m,n)`

Purpose Returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Example

```
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:

```
m - n * FLOOR(m/n)
```

The following statement illustrates the difference between the MOD function and the classical modulus:

```
SELECT m, n, MOD(m, n),
m - n * FLOOR(m/n) "Classical Modulus"
  FROM test_mod_table;

         M          N   MOD(M,N) Classical Modulus
---------- ---------- ---------- -----------------
        11          4          3                 3
        11         -4          3                -1
       -11          4         -3                 1
       -11         -4         -3                -3
```

POWER

Syntax `POWER(m, n)`

Purpose Returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, *n* must be an integer.

Example

```
SELECT POWER(3,2) "Raised" FROM DUAL;

    Raised
----------
         9
```

ROUND

Syntax `ROUND(n[,m])`

Purpose Returns *n* rounded to *m* places right of the decimal point; if *m* is omitted, to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

Example 1

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2
```

Example 2

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;

     Round
----------
        20
```

SIGN

Syntax

```
SIGN(n)
```

Purpose

If *n*<0, the function returns -1; if *n*=0, the function returns 0; if *n*>0, the function returns 1.

Example

```
SELECT SIGN(-15) "Sign" FROM DUAL;

      Sign
----------
        -1
```

SIN

Syntax

```
SIN(n)
```

Purpose

Returns the sine of *n* (an angle expressed in radians).

Example

```
SELECT SIN(30 * 3.14159265359/180)
 "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax

```
SINH(n)
```

Purpose

Returns the hyperbolic sine of *n*.

Syntax `SINH(n)`

Example

```
SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;

Hyperbolic sine of 1
--------------------
          1.17520119
```

SQRT

Syntax `SQRT(n)`

Purpose Returns square root of *n*. The value *n* cannot be negative. SQRT returns a "real" result.

Example

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
  5.09901951
```

TAN

Syntax `TAN`

Purpose Returns the tangent of *n* (an angle expressed in radians).

Example

```
SELECT TAN(135 * 3.14159265359/180)
"Tangent of 135 degrees"  FROM DUAL;

Tangent of 135 degrees
----------------------
                    - 1
```

TANH

Syntax `TANH(n)`

Purpose Returns the hyperbolic tangent of *n*.

Example

```
SELECT TANH(.5) "Hyperbolic tangent of .5"
   FROM DUAL;

Hyperbolic tangent of .5
------------------------
               .462117157
```

TRUNC

Syntax `TRUNC(n[,m])`

Purpose Returns *n* truncated to *m* decimal places; if *m* is omitted, to 0 places. *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

Examples

```
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

  Truncate
----------
      15.7

SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

  Truncate
----------
        10
```

Character Functions

Single-row character functions accept character input and can return either character or number values.

Character Functions Returning Character Values

This section lists character functions that return character values. Unless otherwise noted, these functions all return values with the datatype VARCHAR2 and are limited in length to 4000 bytes. Functions that return values of datatype CHAR are limited in length to 2000 bytes. If the length of the return value exceeds the limit, Oracle truncates it and returns the result without an error message.

CHR

Syntax `CHR(n [USING NCHAR_CS])`

Purpose Returns the character having the binary equivalent to *n* in either the database character set or the national character set.

If the USING NCHAR_CS clause is *not* specified, this function returns the character having the binary equivalent to *n as a VARCHAR2 value* in the database character set.

If the USING NCHAR_CS clause is specified, this function returns the character having the binary equivalent to *n* as a NVARCHAR2 value in the national character set.

Example 1

```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog"
  FROM DUAL;
Dog
---
CAT
```

Example 2

```
SELECT CHR(16705 USING NCHAR_CS) FROM DUAL;

C
-
A
```

CONCAT

Syntax

```
CONCAT(char1, char2)
```

Purpose

Returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||). For information on this operator, see "Concatenation Operator" on page 3-4.

Example

This example uses nesting to concatenate three character strings:

```
SELECT CONCAT( CONCAT(ename, ' is a '), job) "Job"
FROM emp
WHERE empno = 7900;

Job
-----------------
JAMES is a CLERK
```

INITCAP

Syntax

```
INITCAP(char)
```

Purpose

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.

Example

```
SELECT INITCAP('the soap') "Capitals" FROM DUAL;

Capitals
---------
The Soap
```

LOWER

Syntax

```
LOWER(char)
```

Purpose

Returns *char*, with all letters lowercase. The return value has the same datatype as the argument *char* (CHAR or VARCHAR2).

Example

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
   FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax

```
LPAD(char1,n [,char2])
```

Purpose

Returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* 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.

Example

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
     FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax

```
LTRIM(char [,set])
```

Purpose

Removes characters from the left of *char*, with all the leftmost characters that appear in *set* removed; *set* defaults to a single blank. Oracle 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.

Example

```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
     FROM DUAL;

LTRIM exampl
------------
XxyLAST WORD
```

NLS_INITCAP

Syntax

```
NLS_INITCAP(char [, 'nlsparams'] )
```

Purpose

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. The value of '*nlsparams*' can have this form:

```
'NLS_SORT = sort'
```

where *sort* is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. Note that these requirements can result in a return value of a different length than the *char*. If you omit '*nlsparams*', this function uses the default sort sequence for your session. For information on sort sequences, see *Oracle8 Reference*.

Example

```
SELECT NLS_INITCAP
   ('ijsland', 'NLS_SORT = XDutch') "Capitalized"
   FROM DUAL;

Capital
-------
IJsland
```

NLS_LOWER

Syntax

```
NLS_LOWER(char [, 'nlsparams'] )
```

Purpose

Returns *char*, with all letters lowercase. The '*nlsparams*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

Example

```
SELECT NLS_LOWER
   ('CITTA''', 'NLS_SORT = XGerman') "Lowercase"
   FROM DUAL;

Lower
-----
cittá
```

NLS_UPPER

Syntax

```
NLS_UPPER(char [, 'nlsparams'] )
```

Purpose

Returns *char*, with all letters uppercase. The '*nlsparams*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

Example

```
SELECT NLS_UPPER
   ('groβe', 'NLS_SORT = XGerman') "Uppercase"
     FROM DUAL;

Upper
-----
GROSS
```

REPLACE

Syntax

```
REPLACE(char,search_string[,replacement_string])
```

Purpose

Returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, all occurrences of *search_string* are removed. If *search_string* is null, char is returned. This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE allows you to substitute one string for another as well as to remove character strings.

Example

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
     FROM DUAL;

Changes
--------------
BLACK and BLUE
```

RPAD

Syntax

```
RPAD(char1, n [,char2])
```

Purpose Returns *char1*, right-padded to length *n* with *char2*, replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* 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.

Example

```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
     FROM DUAL;

RPAD example
------------------
MORRISONabab
```

RTRIM

Syntax

```
RTRIM(char [,set]
```

Purpose Returns *char*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. RTRIM works similarly to LTRIM.

Example

```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM e.g."
     FROM DUAL;

RTRIM e.g
-------------
BROWNINGyxX
```

SOUNDEX

Syntax

```
SOUNDEX(char)
```

Purpose Returns a character string containing the phonetic representation of *char*. This function allows you to 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:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.

- 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
```

- If two or more letters with the same assigned number are adjacent, remove all but the first.
- Return the first four bytes padded with 0.

Example

```
SELECT ename
     FROM emp
     WHERE SOUNDEX(ename)
          = SOUNDEX('SMYTHE');

ENAME
----------
SMITH
```

SUBSTR

Syntax

```
SUBSTR(char, m [,n])
```

Purpose

Returns a portion of *char*, beginning at character *m*, *n* characters long. If *m* is 0, it is treated as 1. If *m* is positive, Oracle counts from the beginning of *char* to find the first character. If *m* is negative, Oracle counts backwards from the end of *char*. If *n* is omitted, Oracle returns all characters to the end of *char*. If n is less than 1, a null is returned.

Floating-point numbers passed as arguments to *substr* are automatically converted to integers.

Example 1

```
SELECT SUBSTR('ABCDEFG',3.1,4) "Subs"
     FROM DUAL;

Subs
----
CDEF
```

Example 2

```
SELECT SUBSTR('ABCDEFG',-5,4) "Subs"
     FROM DUAL;

Subs
----
CDEF
```

SUBSTRB

Syntax

```
SUBSTR(char, m [,n])
```

Purpose

The same as SUBSTR, except that the arguments *m* and *n* are expressed in bytes, rather than in characters. For a single-byte database character set, SUBSTRB is equivalent to SUBSTR.

Floating-point numbers passed as arguments to *substrb* are automatically converted to integers.

Example

Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2)
     "Substring with bytes"
     FROM DUAL;

Substring with bytes
--------------------
CD
```

TRANSLATE

Syntax

```
TRANSLATE(char, from, to)
```

Purpose

Returns *char* with all occurrences of each character in *from* replaced by its corresponding character in *to*. Characters in *char* that are not in *from* are not replaced. The argument *from* can contain more characters than *to*. In this case, the extra characters at the end of *from* have no corresponding characters in *to*. If these extra characters appear in *char*, they are removed from the return value. You cannot use an empty string for *to* to remove all characters in *from* from the return value. Oracle interprets the empty string as null, and if this function has a null argument, it returns null.

Example 1

The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012 . . . 9' are translated to '9':

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXX') "License"
     FROM DUAL;

License
--------
9XXX999
```

Example 2 The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'0123456789')
"Translate example"
     FROM DUAL;

Translate example
-----------------
2229
```

UPPER

Syntax

```
UPPER(char)
```

Purpose Returns *char*, with all letters uppercase. The return value has the same datatype as the argument *char*.

Example

```
SELECT UPPER('Large') "Uppercase"
     FROM DUAL;

Upper
-----
LARGE
```

Character Functions Returning Number Values

This section lists character functions that return number values.

ASCII

Syntax

```
ASCII(char)
```

Purpose Returns the decimal representation in the database character set of the first character of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code Page 500, this function returns an EBCDIC value. Note that there is no similar EBCDIC character function.

Example

```
SELECT ASCII('Q')
     FROM DUAL;

ASCII('Q')
----------
        81
```

INSTR

Syntax

```
INSTR (char1,char2 [,n[,m]])
```

Purpose Searches *char1* beginning with its *n*th character for the *m*th occurrence of *char2* and returns the position of the character in *char1* that is the first character of this occurrence. If *n* is negative, Oracle counts and searches backward from the end of *char1*. The value of *m* must be positive. The default values of both *n* and *m* are 1, meaning Oracle begins searching at the first character of *char1* for the first occurrence of *char2*. The return value is relative to the beginning of *char1*, regardless of the value of *n*, and is expressed in characters. If the search is unsuccessful (if *char2* does not appear *m* times after the *n*th character of *char1*) the return value is 0.

Example 1

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2)
  "Instring" FROM DUAL;

  Instring
----------
        14
```

Example 2

```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
     FROM DUAL;

Reversed Instring
-----------------
                2
```

INSTRB

Syntax `INSTRB(char1,char2[,n[,m]])`

Purpose The same as INSTR, except that n and the return value are expressed in bytes, rather than in characters. For a single-byte database character set, INSTRB is equivalent to INSTR.

Example This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2)
"Instring in bytes"
FROM DUAL;

Instring in bytes
-----------------
               27
```

LENGTH

Syntax `LENGTH(char)`

Purpose Returns the length of *char* in characters. If *char* has datatype CHAR, the length includes all trailing blanks. If *char* is null, this function returns null.

Example

```
SELECT LENGTH('CANDIDE') "Length in characters"
FROM DUAL;

Length in characters
--------------------
                   7
```

LENGTHB

Syntax `LENGTHB(char)`

Purpose Returns the length of *char* in bytes. If char is null, this function returns null. For a single-byte database character set, LENGTHB is equivalent to LENGTH.

Example This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
     FROM DUAL;

Length in bytes
---------------
             14
```

NLSSORT

Syntax
```
NLSSORT(char [, 'nlsparams'])
```

Purpose Returns the string of bytes used to sort *char*. The value of '*nlsparams*' can have the form

```
'NLS_SORT = sort'
```

where *sort* is a linguistic sort sequence or BINARY. If you omit '*nlsparams*', this function uses the default sort sequence for your session. If you specify BINARY, this function returns *char*. For information on sort sequences, see the discussions of national language support in *Oracle8 Reference*.

Example This function can be used to specify comparisons based on a linguistic sort sequence rather on the binary value of a string:

```
SELECT ename FROM emp
   WHERE NLSSORT (ename, 'NLS_SORT = German')
   > NLSSORT ('S', 'NLS_SORT = German') ORDER BY ename;

ENAME
----------
SCOTT
SMITH
TURNER
WARD
```

Date Functions

Date functions operate on values of the DATE datatype. All date functions return a value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number.

ADD_MONTHS

Syntax
```
ADD_MONTHS(d,n)
```

Purpose Returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Example

```
SELECT TO_CHAR(
     ADD_MONTHS(hiredate,1),
     'DD-MON-YYYY') "Next month"
     FROM emp
     WHERE ename = 'SMITH';

Next Month
-----------
17-JAN-1981
```

LAST_DAY

Syntax

```
LAST_DAY(d)
```

Purpose Returns the date of the last day of the month that contains *d*. You might use this function to determine how many days are left in the current month.

Example 1

```
SELECT SYSDATE,
   LAST_DAY(SYSDATE) "Last",
   LAST_DAY(SYSDATE) - SYSDATE "Days Left"
   FROM DUAL;

SYSDATE   Last       Days Left
--------- --------- ----------
23-OCT-97 31-OCT-97          8
```

Example 2

```
SELECT TO_CHAR(
  ADD_MONTHS(
    LAST_DAY(hiredate),5),
     'DD-MON-YYYY') "Five months"
  FROM emp
  WHERE ename = 'MARTIN';

Five months
-----------
28-FEB-1982
```

MONTHS_BETWEEN

Syntax

```
MONTHS_BETWEEN(d1, d2)
```

Purpose

Returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer; otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

Example

```
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
```

NEW_TIME

Syntax

```
NEW_TIME(d, z1, z2)
```

Purpose

Returns the date and time in time zone *z2* when date and time in time zone *z1* are *d*. The arguments *z1* and *z2* can be any of these text strings:

AST	Atlantic Standard or Daylight Time
ADT	
BST	Bering Standard or Daylight Time
BDT	
CST	Central Standard or Daylight Time
CDT	
EST	Eastern Standard or Daylight Time
EDT	
GMT	Greenwich Mean Time
HST	Alaska-Hawaii Standard Time or Daylight Time.
HDT	
MST	Mountain Standard or Daylight Time
MDT	

NST	Newfoundland Standard Time
PST	Pacific Standard or Daylight Time
PDT	
YST	Yukon Standard or Daylight Time
YDT	

NEXT_DAY

Syntax `NEXT_DAY(d, char)`

Purpose Returns the date of the first weekday named by *char* that is later than the date *d*. The argument *char* must be a day of the week in your session's date language—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 *d*.

Example This example returns the date of the next Tuesday after March 15, 1992.

```
SELECT NEXT_DAY('15-MAR-92','TUESDAY') "NEXT DAY"
     FROM DUAL;

NEXT DAY
---------
17-MAR-92
```

ROUND

Syntax `ROUND(d[,fmt])`

Purpose Returns *d* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is rounded to the nearest day. See "ROUND and TRUNC" on page 3-40 for the permitted format models to use in *fmt*.

Example

```
SELECT ROUND (TO_DATE ('27-OCT-92'),'YEAR')
   "New Year" FROM DUAL;

New Year
---------
01-JAN-93
```

SYSDATE

Syntax SYSDATE

Purpose Returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example

```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
     FROM DUAL;
NOW
-------------------
10-29-1993 20:27:11
```

TRUNC

Syntax 1TRUNC(d,[fmt])

Purpose Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is truncated to the nearest day. See "ROUND and TRUNC" on page 3-40 for the permitted format models to use in *fmt*.

Example

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
  "New Year" FROM DUAL;

New Year
---------
01-JAN-92
```

ROUND and TRUNC

Table 3–11 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 3–11 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.

Table 3–11 (Cont.) Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	ISO Year
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 ISO year.
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. For information on this parameter, see *Oracle8 Reference*.

Conversion Functions

Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention *datatype* TO *datatype*. The first datatype is the input datatype; the last datatype is the output datatype. This section lists the SQL conversion functions.

CHARTOROWID

Syntax

```
CHARTOROWID(char)
```

Purpose

Converts a value from CHAR or VARCHAR2 datatype to ROWID datatype.

Example

```
SELECT ename FROM emp
   WHERE ROWID = CHARTOROWID('AAAAfZAABAAACp8AAO');

ENAME
----------
LEWIS
```

CONVERT

Syntax

```
CONVERT(char, dest_char_set [,source_char_set] )
```

Purpose

Converts a character string from one character set to another.

The *char* argument is the value to be converted.

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.

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.

Example

```
SELECT CONVERT('Groß', 'US7ASCII', 'WE8HP')
"Conversion"
  FROM DUAL;

Conversion
----------
Gross
```

Common character sets include:

US7ASCII	US 7-bit ASCII character set
WE8DEC	DEC West European 8-bit character set
WE8HP	HP West European Laserjet 8-bit character set
F7DEC	DEC French 7-bit character set
WE8EBCDIC500	IBM West European EBCDIC Code Page 500
WE8PC850	IBM PC Code Page 850
WE8ISO8859P1	ISO 8859-1 West European 8-bit character set

HEXTORAW

Syntax `HEXTORAW(char)`

Purpose Converts *char* containing hexadecimal digits to a raw value.

Example

```
INSERT INTO graphics (raw_column)
  SELECT HEXTORAW('7D') FROM DUAL;
```

RAWTOHEX

Syntax `RAWTOHEX(raw)`

Purpose Converts *raw* to a character value containing its hexadecimal equivalent.

Example

```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM graphics;

Graphics
--------
7D
```

ROWIDTOCHAR

Syntax

```
ROWIDTOCHAR(rowid)
```

Purpose

Converts a ROWID value to VARCHAR2 datatype. The result of this conversion is always 18 characters long.

Example

```
SELECT ROWID
     FROM offices
     WHERE
     ROWIDTOCHAR(ROWID) LIKE '%Br1AAB%';

ROWID
------------------
AAAAZ6AABAAABr1AAB
```

TO_CHAR, date conversion

Syntax

```
TO_CHAR(d [, fmt [, 'nlsparams'] ])
```

Purpose

Converts *d* of DATE datatype to a value of VARCHAR2 datatype in the format specified by the date format *fmt*. If you omit *fmt*, *d* is converted to a VARCHAR2 value in the default date format. For information on date formats, see "Format Models" on page 3-63.

The '*nlsparams*' 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 *nlsparams*, this function uses the default date language for your session.

Example

```
SELECT TO_CHAR(HIREDATE, 'Month DD, YYYY')
    "New date format" FROM emp
    WHERE ename = 'BLAKE';

New date format
------------------
May       01, 1981
```

TO_CHAR, number conversion

Syntax

```
TO_CHAR(n [, fmt [, 'nlsparams'] ])
```

Purpose

Converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit *fmt*, *n* is converted to a VARCHAR2 value exactly long enough to hold its significant digits. For information on number formats, see "Format Models" on page 3-63.

The '*nlsparams*' 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. Note that 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 '*nlsparams*' or any one of the parameters, this function uses the default parameter values for your session.

Example 1

In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
     FROM DUAL;

Amount
--------------
  $10,000.00-
```

Example 2

```
SELECT TO_CHAR(-10000,'L99G999D99MI',
'NLS_NUMERIC_CHARACTERS = '',.''
NLS_CURRENCY = ''AusDollars'' ') "Amount"
     FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

Notes:

- In the optional number format *fmt*, L designates local currency symbol and MI designates a trailing minus sign. See Table 3–13 on page 3-67 for a complete listing of number format elements.
- During a conversion of Oracle numbers to string, if a rounding operation occurs that overflows or underflows the Oracle NUMBER range, then ~ or -~ may be returned, representing infinity and negative infinity, respectively. This event typically occurs when you are using TO_CHAR() with a restrictive number format string, causing a rounding operation.

TO_DATE

Syntax

```
TO_DATE(char [, fmt [, 'nlsparams'] ])
```

Purpose

Converts *char* of CHAR or VARCHAR2 datatype to a value of DATE datatype. The *fmt* is a date format specifying the format of *char*. If you omit *fmt*, *char* must be in the default date format. If *fmt* is 'J', for Julian, then *char* must be an integer. For information on date formats, see "Format Models" on page 3-63.

The '*nlsparams*' has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *char* argument. The returned DATE value can have a different century value than the original *char*, depending on *fmt* or the default date format.

For information on date formats, see "Date Format Models" on page 3-69.

Example

```
INSERT INTO bonus (bonus_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_MULTI_BYTE

Syntax `TO_MULTI_BYTE(char)`

Purpose Returns *char* with all of its single-byte characters converted to their corresponding multibyte characters. Any single-byte characters in *char* that have no multibyte equivalents appear in the output string as single-byte characters. This function is only useful if your database character set contains both single-byte and multibyte characters.

TO_NUMBER

Syntax `TO_NUMBER(char [,fmt [, 'nlsparams'] ])`

Purpose Converts *char*, a value of CHAR or VARCHAR2 datatype containing a number in the format specified by the optional format model *fmt*, to a value of NUMBER datatype.

Example 1

```
UPDATE emp SET sal = sal +
   TO_NUMBER('100.00', '9G999D99')
  WHERE ename = 'BLAKE';
```

The '*nlsparams*' string in this function has the same purpose as it does in the TO_CHAR function for number conversions.

Example 2

```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY           = ''AusDollars''
   ') "Amount"
     FROM DUAL;

    Amount
----------
      -100
```

TO_SINGLE_BYTE

Syntax `TO_SINGLE_BYTE(char)`

Purpose Returns *char* with all of its multibyte character converted to their corresponding single-byte characters. Any multibyte characters in *char* that have no single-byte equivalents appear in the output as multibyte characters. This function is only useful if your database character set contains both single-byte and multibyte characters.

TRANSLATE USING

Syntax

```
TRANSLATE(text USING {CHAR_CS | NCHAR_CS })
```

Purpose Converts *text* into the character set specified for conversions between the database character set and the national character set.

The *text* argument is the expression to be converted.

Specifying the USING CHAR_CS argument converts *text* into the database character set. The output datatype is VARCHAR2.

Specifying the USING NCHAR_CS argument converts *text* into the national character set. The output datatype 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 datatype is being used as NCHAR or NVARCHAR2.

Example 1

```
CREATE TABLE t1 (char_col  CHAR(20),
                 nchar_col nchar(20));
INSERT INTO t1
  VALUES ('Hi', N'Bye');
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
Hi           Bye
```

Example 2

```
UPDATE t1 SET
  nchar_col = TRANSLATE(char_col USING NCHAR_CS);
UPDATE t1 SET
  char_col = TRANSLATE(nchar_col USING CHAR_CS);
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
Hi           Hi
```

Example 3

```
UPDATE t1 SET
  nchar_col = TRANSLATE('deo' USING NCHAR_CS);
UPDATE t1 SET
  char_col = TRANSLATE(N'deo' USING CHAR_CS);

CHAR_COL     NCHAR_COL
--------     ---------
deo          deo
```

Other Single-Row Functions

DUMP

Syntax

```
DUMP(expr[,return_format[,start_position[,length]] ] )
```

Purpose

Returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of *expr*. The returned result is always in the database character set. For the datatype corresponding to each code, see Table 2–1 on page 2-6.

The argument *return_format* specifies the format of the return value and can have any of the values listed below.

By default, the return value contains no character set information. To retrieve the character set name of *expr*, specify any of the format values below, plus 1000. For example, a *return_format* of 1008 returns the result in octal, plus provides the character set name of *expr*.

8 returns result in octal notation.

10 returns result in decimal notation.

16 returns result in hexadecimal notation.

17 returns result as single characters.

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, this function returns 'NULL'.

Example 1

```
SELECT DUMP('abc', 1016)
 FROM DUAL;

DUMP('ABC',1016)
----------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63
```

Example 2

```
SELECT DUMP(ename, 8, 3, 2) "OCTAL"
 FROM emp
 WHERE ename = 'SCOTT';

OCTAL
----------------------------
Type=1 Len=5: 117,124
```

Example 3

```
SELECT DUMP(ename, 10, 3, 2) "ASCII"
FROM emp
WHERE ename = 'SCOTT';

ASCII
----------------------------
Type=1 Len=5: 79,84
```

EMPTY_[B | C]LOB

Syntax

```
EMPTY_[B|C]LOB()
```

Purpose

Returns 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.

You cannot use the locator returned from this function as a parameter to the DBMS_LOB package or the OCI.

Examples

```
INSERT INTO lob_tab1 VALUES (EMPTY_BLOB());
UPDATE lob_tab1
  SET clob_col = EMPTY_BLOB();
```

BFILENAME

Syntax

```
BFILENAME ('directory', 'filename')
```

Purpose

Returns a BFILE locator that is associated with a physical LOB binary file on the server's file system. A directory is an alias for a full pathname on the server's file system where the files are actually located; 'filename' is the name of the file in the server's file system.

Neither 'directory' nor 'filename' need to point to an existing object on the file system at the time you specify BFILENAME. However, you must associate a BFILE value with a physical file before performing subsequent SQL, PL/SQL, DBMS_LOB package, or OCI operations. For more information, see CREATE DIRECTORY on page 4-230.

Note: This function does not verify that either the directory or file specified actually exists. Therefore, you can call the CREATE DIRECTORY command after BFILENAME. However, the object must exist by the time you actually use the BFILE locator (for example, as a parameter to one of the OCILob or DBMS_LOB operations such as OCILobFileOpen() or DBMS_LOB.FILEOPEN()).

For more information about LOBs, see *Oracle8 Application Developer's Guide* and *Oracle Call Interface Programmer's Guide*.

Example

```
INSERT INTO file_tbl
   VALUES (BFILENAME ('lob_dir1', 'image1.gif'));
```

GREATEST

Syntax

```
GREATEST(expr [,expr] ...)
```

Purpose

Returns the greatest of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *exprs* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher value. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT GREATEST ('HARRY', 'HARRIOT', 'HAROLD')
   "Great" FROM DUAL;

Great
-----
HARRY
```

LEAST

Syntax

```
LEAST(expr [,expr] ...)
```

Purpose

Returns the least of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST"
     FROM DUAL;

LEAST
------
 HAROLD
```

NLS_CHARSET_DECL_LEN

Syntax

```
NLS_CHARSET_DECL_LEN(bytecnt, csid)
```

Purpose

Returns the declaration width (in number of characters) of an NCHAR column. The *bytecnt* argument is the width of the column. The *csid* argument is the character set ID of the column.

Example

```
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

```
NLS_CHARSET_ID(text)
```

Purpose

Returns the NLS character set ID number corresponding to NLS character set name, *text*. The *text* argument is a run-time VARCHAR2 value. The *text* value 'CHAR_CS' returns the server's database character set ID number. The *text* value 'NCHAR_CS' returns the server's national character set ID number.

Invalid character set names return null.

For a list of character set names, see *Oracle8 Reference*.

Example I

```
SELECT NLS_CHARSET_ID('ja16euc')
  FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
-------------------------
                      830
```

Example 2

```
SELECT NLS_CHARSET_ID('char_cs')
  FROM DUAL;

 NLS_CHARSET_ID('CHAR_CS')
 -------------------------
                         2
```

Example 3

```
SELECT NLS_CHARSET_ID('nchar_cs')
  FROM DUAL;

NLS_CHARSET_ID('NCHAR_CS')
--------------------------
                         2
```

NLS_CHARSET_NAME

Syntax

```
NLS_CHARSET_NAME(n)
```

Purpose

Returns the name of the NLS character set corresponding to ID number *n*. The character set name is returned as a VARCHAR2 value in the database character set.

If *n* is not recognized as a valid character set ID, this function returns null.

For a list of character set IDs, see *Oracle8 Reference*.

Example

```
SELECT NLS_CHARSET_NAME(2)
  FROM DUAL;

NLS_CH
------
WE8DEC
```

NVL

Syntax

```
NVL(expr1, expr2)
```

Purpose

If *expr1* is null, returns *expr2*; if *expr1* is not null, returns *expr1*. The arguments *expr1* and *expr2* can have any datatype. If their datatypes are different, Oracle converts *expr2* to the datatype of *expr1* before comparing them. The datatype of the return value is always the same as the datatype of *expr1*, unless *expr1* is character data, in which case the return value's datatype is VARCHAR2.

Example

```
SELECT ename, NVL(TO_CHAR(COMM), 'NOT
APPLICABLE')
   "COMMISSION" FROM emp
   WHERE deptno = 30;

ENAME      COMMISSION
---------- -----------------------------------
ALLEN      300
WARD       500
MARTIN     1400
BLAKE      NOT APPLICABLE
TURNER     0
JAMES      NOT APPLICABLE
```

UID

Syntax

```
UID
```

Purpose

Returns an integer that uniquely identifies the current user.

USER

Syntax

```
USER
```

Purpose

Returns the current Oracle user with the datatype VARCHAR2. Oracle compares values of this function with blank-padded comparison semantics.

In a distributed SQL statement, the UID and USER functions identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint.

Example

```
SELECT USER, UID FROM DUAL;

USER                                UID
------------------------------ ----------
SCOTT                                19
```

USERENV

Syntax

```
USERENV(option)
```

Purpose

Returns information of VARCHAR2 datatype 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. The argument *option* can have any of these values:

`'ISDBA'`	returns 'TRUE' if you currently have the ISDBA role enabled and 'FALSE' if you do not.
`'LANGUAGE'`	returns the language and territory currently used by your session along with the database character set in this form: `language_territory.characterset`
`'TERMINAL'`	returns the operating system identifier for your current session's terminal. In distributed SQL statements, this option returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECTs, not for remote INSERTs, UPDATEs, or DELETEs.
`'SESSIONID'`	returns your auditing session identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.
`'ENTRYID'`	returns available auditing entry identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.
`'LANG'`	Returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
`'INSTANCE'`	Returns the instance identification number of the current instance.

Example

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

Language
----------------------------------
AMERICAN_AMERICA.WE8DEC
```

VSIZE

Syntax

```
VSIZE(expr)
```

Purpose Returns the number of bytes in the internal representation of *expr*. If *expr* is null, this function returns null.

Example

```
SELECT ename, VSIZE (ename) "BYTES"
  FROM emp
  WHERE deptno = 10;

ENAME           BYTES
---------- ----------
CLARK               5
KING                4
MILLER              6
```

Object Reference Functions

Object reference functions manipulate REFs—references to objects of specified object types. For more information about REFs, see *Oracle8 Concepts* and *Oracle8 Application Developer's Guide*.

DEREF

Syntax

```
DEREF(e)
```

Purpose Returns the object reference of argument *e*. Argument *e* must be an expression that returns a REF to an object.

Example

```
CREATE TABLE tb1(c1 NUMBER, c2 REF t1);
SELECT DEREF(c2) FROM tb1;
```

REFTOHEX

Syntax

```
REFTOHEX(r)
```

Purpose Converts argument *r* to a character value containing its hexadecimal equivalent.

Example

```
CREATE TABLE tb1(c1 NUMBER, c2 REF t1);
SELECT REFTOHEX(c2) FROM tb1;
```

MAKE_REF

Syntax

```
MAKE_REF(table, key [,key...])
```

Purpose Creates a REF to a row of an object view using *key* as the primary key. For more information about object views, see *Oracle8 Application Developer's Guide*.

Example

```
CREATE TYPE t1 AS OBJECT(a NUMBER, b NUMBER);

CREATE TABLE tb1
   (c1 NUMBER, c2 NUMBER, PRIMARY KEY(c1, c2));

CREATE VIEW v1 OF t1 WITH OBJECT OID(a, b) AS
   SELECT * FROM tb1;

SELECT MAKE_REF(v1, 1, 3) FROM DUAL;
```

Group Functions

Group functions return results based on groups of rows, rather than on single rows. In this way, group functions are different from single-row functions. For a discussion of the differences between group functions and single-row functions, see "SQL Functions" on page 3-16.

Many group functions accept these options:

DISTINCT	This option causes a group function to consider only distinct values of the argument expression.
ALL	This option causes a group 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 neither option is specified, the default is ALL.

All group functions except COUNT(*) ignore nulls. You can use the NVL in the argument to a group function to substitute a value for a null.

If a query with a group function returns no rows or only rows with nulls for the argument to the group function, the group function returns null.

AVG

Syntax `AVG([DISTINCT|ALL] n)`

Purpose Returns average value of *n*.

Example

```
SELECT AVG(sal) "Average"
   FROM emp;

   Average
----------
2077.21429
```

COUNT

Syntax

```
COUNT({* | [DISTINCT|ALL] expr})
```

Purpose

Returns the number of rows in the query.

If you specify *expr*, this function returns rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.

If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

Example 1

```
SELECT COUNT(*) "Total"
   FROM emp;

   Total
----------
      18
```

Example 2

```
SELECT COUNT(job) "Count"
   FROM emp;

   Count
----------
      14
```

Example 3

```
SELECT COUNT(DISTINCT job) "Jobs"
     FROM emp;

      Jobs
----------
        5
```

MAX

Syntax

```
MAX([DISTINCT|ALL] expr)
```

Purpose

Returns maximum value of *expr*.

Example

```
SELECT MAX(sal) "Maximum" FROM emp;

   Maximum
----------
      5000
```

MIN

Syntax

```
MIN([DISTINCT|ALL] expr)
```

Purpose

Returns minimum value of *expr*.

Example

```
SELECT MIN(hiredate) "Earliest" FROM emp;

Earliest
---------
17-DEC-80
```

STDDEV

Syntax

```
STDDEV([DISTINCT|ALL] x)
```

Purpose

Returns standard deviation of *x*, a number. Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE group function.

Example

```
SELECT STDDEV(sal) "Deviation"
     FROM emp;

 Deviation
----------
1182.50322
```

SUM

Syntax

```
SUM([DISTINCT|ALL] n)
```

Purpose

Returns sum of values of *n*.

Example

```
SELECT SUM(sal) "Total"
     FROM emp;

     Total
----------
     29081
```

VARIANCE

Syntax `VARIANCE([DISTINCT|ALL]x)`

Purpose Returns variance of *x*, a number. Oracle calculates the variance of *x* using this formula:

$$\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2}{n-1}$$

where:

x_i is one of the elements of *x*.

n is the number of elements in the set *x*. If n is 1, the variance is defined to be 0.

Example

```
SELECT VARIANCE(sal) "Variance"
     FROM emp;

Variance
----------
1389313.87
```

User Functions

You can write your own user functions in PL/SQL to provide functionality that is not available in SQL or SQL functions. User functions are used in a SQL statement anywhere SQL functions can be used; that is, wherever expression can occur.

For example, user functions can be used in the following:

- the select list of a SELECT command
- the condition of a WHERE clause
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- the VALUES clause of an INSERT command
- the SET clause of an UPDATE command

For a complete description on the creation and use of user functions, see *Oracle8 Application Developer's Guide*.

Prerequisites

User functions must be created as top-level PL/SQL functions or declared with a package specification before they can be named within a SQL statement. Create user functions as top-level PL/SQL functions by using the CREATE FUNCTION statement described in CREATE FUNCTION on page 4-232. Specify packaged functions with a package with the CREATE PACKAGE statement described in CREATE PACKAGE on page 4-250.

To call a packaged user function, you must declare the RESTRICT_REFERENCES pragma in the package specification.

Privileges Required

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 SELECT privileges on the view. No separate EXECUTE privileges are needed to select from the view.

Restrictions on User Functions

User functions cannot be used in situations that require an unchanging definition. Thus, a user function cannot:

- be used in a CHECK constraint clause of a CREATE TABLE or ALTER TABLE command
- be used in a DEFAULT clause of a CREATE TABLE or ALTER TABLE command
- contain OUT or IN OUT parameters
- update the database
- read or write package state if the function is a remote function
- use the *parallelism_clause* in SQL commands in the function if the function alters package state
- update variables defined in the function unless the function is a local function and is used in a SELECT list, VALUES clause of an INSERT command, or SET clause of an UPDATE command

Name Precedence

With PL/SQL, the names of database columns take precedence over the names of functions with no parameters. For example, if user SCOTT creates the following two objects in his own schema:

```
CREATE TABLE emp(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 EMP.NEW_SAL:

```
SELECT new_sal FROM emp;
SELECT emp.new_sal FROM emp;
```

To access the function NEW_SAL, you would enter:

```
SELECT scott.new_sal FROM emp;
```

Here are some sample calls to user functions that are allowed in SQL expressions.

```
circle_area (radius)
payroll.tax_rate (empno)
scott.payroll.tax_rate (dependent, empno)@ny
```

Example For example, to call the TAX_RATE user function from schema SCOTT, execute it against the SS_NO and SAL columns in TAX_TABLE, and place the results in the variable INCOME_TAX, specify the following:

```
SELECT scott.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table
    WHERE ss_no = tax_id;
```

Naming Conventions

If only one of the optional schema or package names is given, 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 proceeds as follows:

- Check for the PAYROLL package in the current schema.
- If a PAYROLL package is not found, look for a schema name PAYROLL that contains a top-level TAX_RATE function. If no such function is found, return an error message.
- If the PAYROLL package is found in the current schema, look for a TAX_RATE function in the PAYROLL package. If no such function is found, return an error message.

You can also refer to a stored top-level function using any synonym that you have defined for it.

Format Models

A *format model* is a character literal that describes the format of DATE or NUMBER data stored in a character string. You can use a format model as an argument of the TO_CHAR or TO_DATE function:

- to specify the format for Oracle to use to return a value from the database to you
- to specify the format for a value you have specified for Oracle to store in the database

Note that a format model does not change the internal representation of the value in the database.

This section describes how to use:

- number format models
- date format models
- format model modifiers

Changing the Return Format

You can use a format model to specify the format for Oracle to use to return values from the database to you.

Example 1 The following statement selects the commission values of the employees in Department 30 and uses the TO_CHAR function to convert these commissions into character values with the format specified by the number format model '$9,990.99':

```
SELECT ename employee, TO_CHAR(comm, '$9,990.99') commission
   FROM emp
   WHERE deptno = 30;

EMPLOYEE   COMMISSION
---------- ----------
ALLEN         $300.00
WARD          $500.00
MARTIN      $1,400.00
BLAKE
TURNER          $0.00
JAMES
```

Because of this format model, Oracle returns commissions with leading dollar signs, commas every three digits, and two decimal places. Note that TO_CHAR returns null for all employees with null in the COMM column.

Example 2 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 ename, TO_CHAR(Hiredate,'fmMonth DD, YYYY') hiredate
   FROM emp
   WHERE deptno = 20;

ENAME      HIREDATE
---------- ------------------
SMITH      December 17, 1980
JONES      April 2, 1981
SCOTT      April 19, 1987
ADAMS      May 23, 1987
FORD       December 3, 1981
LEWIS      October 23, 1997
```

With this format model, Oracle returns the hire dates with the month spelled out (as specified by "fm" and discussed in "Format Model Modifiers" on page 3-75), two digits for the day, and the century included in the year.

Supplying the Correct Format

You can use format models to specify the format of a value that you are converting from one datatype to another datatype required for a column. When you insert or update a column value, the datatype of the value that you specify must correspond to the column's datatype. For example, a value that you insert into a DATE column must be a value of the DATE datatype or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE datatype). If the value is in another format, you must use the TO_DATE function to convert the value to the DATE datatype. You must also use a format model to specify the format of the character string.

Example The following statement updates BAKER's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '1992 05 20' to a DATE value:

```
UPDATE emp
  SET hiredate = TO_DATE('1992 05 20','YYYY MM DD')
```

```
WHERE ename = 'BLAKE';
```

Number Format Models

You can use number format models

- in the TO_CHAR function to translate a value of NUMBER datatype to VARCHAR2 datatype
- in the TO_NUMBER function to translate a value of CHAR or VARCHAR2 datatype to NUMBER datatype

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, pound signs (#) replace the value. If a positive value is extremely large and cannot be represented in the specified format, then the infinity sign (~) replaces the value. Likewise, if a negative value is extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~).

Number Format Elements

A number format model is composed of one or more number format elements. Table 3–12 lists the elements of a number format model. Examples are shown in Table 3–13.

- If a number format model does not contain the MI, S, or PR format elements, negative return values automatically contain a leading negative sign and positive values automatically contain a leading space.
- A number format model can contain only a single decimal character (D) or period (.), but it can contain multiple group separators (G) or commas (,).
- A number format model must not begin with a comma (,).

- A group separator or comma cannot appear to the right of a decimal character or period in a number format model.

Table 3–12 Number Format Elements

Element	Example	Description
9	9999	Return value with the specified number of digits with a leading space if positive.
		Return value with the specified number of digits 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.
0	0999	Return leading zeros.
	9990	Return trailing zeros.
$	$9999	Return value with a leading dollar sign.
B	B9999	Return blanks for the integer part of a fixed-point number when the integer part is zero (regardless of "0's in the format model).
MI	9999MI	Return negative value with a trailing minus sign "-".
		Return positive value with a trailing blank.
S	S9999	Return negative value with a leading minus sign "-".
		Return positive value with a leading plus sign "+".
	9999S	Return negative value with a trailing minus sign "-".
		Return positive value with a trailing plus sign "+".
PR	9999PR	Return negative value in <angle brackets>.
		Return positive value with a leading and trailing blank.
D	99D99	Return a decimal character (that is, a period ".") in the specified position.
G	9G999	Return a group separator in the position specified.
C	C999	Return the ISO currency symbol in the specified position.
L	L999	Return the local currency symbol in the specified position.
, (comma)	9,999	Return a comma in the specified position.

Table 3–12 (Cont.) Number Format Elements

Element	Example	Description
. (period)	`99.99`	Return a decimal point (that is, a period ".") in the specified position.
V	`999V99`	Return a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the "V".
EEEE	`9.9EEEE`	Return a value using in scientific notation.
RN	`RN`	Return a value as Roman numerals in uppercase.
rn		Return a value as Roman numerals in lowercase.
		Value can be an integer between 1 and 3999.
FM	`FM90.9`	Return a value with no leading or trailing blanks.

Example Table 3–13 shows the results of the following query for different values of *number* and '*fmt*':

```
SELECT TO_CHAR(number, 'fmt')
  FROM DUAL
```

Table 3–13 Results of Example Number Conversions

number	'fmt'	Result
-1234567890	9999999999S	'1234567890-'
0	99.99	' .00'
+0.1	99.99	' 0.10'
-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'

Table 3–13 Results of Example Number Conversions (Cont.)

number	'fmt'	Result
-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	'1.23E+02'
+123.45	FM999.009	'123.45'
+123.0	FM999.009	'123.00'
+123.45	L999.99	' $123.45'
+123.45	FML99.99	'$123.45'
+1234567890	9999999999S	'1234567890+'

The MI and PR format elements can appear only in the last position of a number format model. The S format element can appear only in the first or last position of a number format model.

The characters returned by some of these format elements are specified by initialization parameters. Table 3–14 lists these elements and parameters.

Table 3–14 Number Format Element Values Determined by Initialization Parameters

Element	Description	Initialization Parameter
D	Decimal character	NLS_NUMERIC_CHARACTER
G	Group separator	NLS_NUMERIC_CHARACTER
C	ISO currency symbol	NLS_ISO_CURRENCY
L	Local currency symbol	NLS_CURRENCY

You can specify the characters returned by these format elements implicitly using the initialization parameter NLS_TERRITORY. For information on these parameters, see *Oracle8 Reference*.

You can change the characters returned by these format elements for your session with the ALTER SESSION command. You can also change the default date format for your session with the ALTER SESSION command. For information, see ALTER SESSION on page 4-58.

Date Format Models

You can use date format models

- in the TO_CHAR function to translate a DATE value that is in a format other than the default date format
- in the TO_DATE function to translate a character value that is in a format other than the default date format

Default Date Format

The default date format is specified either explicitly with the initialization parameter NLS_DATE_FORMAT or implicitly with the initialization parameter NLS_TERRITORY. For information on these parameters, see *Oracle8 Referencee*.

You can change the default date format for your session with the ALTER SESSION command. For information, see ALTER SESSION on page 4-58.

Maximum Length

The total length of a date format model cannot exceed 22 characters.

Date Format Elements

A date format model is composed of one or more date format elements as listed in Table 3–15. 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. Only some of the date format elements can be used in the TO_DATE function as noted in Table 3–15.

Table 3–15 Date Format Elements

Element	Specify in TO_DATE?	Meaning
- / , . ; : 'text'	Yes	Punctuation and quoted text is reproduced in the result.
AD A.D.	Yes	AD indicator with or without periods.

Table 3–15 (Cont.) Date Format Elements

Element	Specify in TO_DATE?	Meaning
AM A.M.	Yes	Meridian indicator with or without periods.
BC B.C.	Yes	BC indicator with or without periods.
CC SCC	No	One greater than the first two digits of a four-digit year; "S" prefixes BC dates with "-". For example, '20' from '1900'.
D	Yes	Day of week (1-7).
DAY	Yes	Name of day, padded with blanks to length of 9 characters.
DD	Yes	Day of month (1-31).
DDD	Yes	Day of year (1-366).
DY	Yes	Abbreviated name of day.
E	No	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
EE	No	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
HH	Yes	Hour of day (1-12).
HH12	No	Hour of day (1-12).
HH24	Yes	Hour of day (0-23).
IW	No	Week of year (1-52 or 1-53) based on the ISO standard.
IYY IY I	No	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	No	4-digit year based on the ISO standard.
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; JAN = 01)

Table 3–15 (Cont.) Date Format Elements

Element	Specify in TO_DATE?	Meaning
MON	Yes	Abbreviated name of month.
MONTH	Yes	Name of month, padded with blanks to length of 9 characters.
PM P.M.	No	Meridian indicator with or without periods.
Q	No	Quarter of year (1, 2, 3, 4; JAN-MAR = 1)
RM	Yes	Roman numeral month (I-XII; JAN = I).
RR	Yes	Given a year with 2 digits, returns a year in the next century if the year is <50 and the last 2 digits of the current year are >=50; returns a year in the preceding century if the year is >=50 and the last 2 digits of the current year are <50.
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you don't want this functionality, simply enter the 4-digit year.
SS	Yes	Second (0-59).
SSSSS	Yes	Seconds past midnight (0-86399).
WW	No	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	No	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
Y,YYY	Yes	Year with comma in this position.
YEAR SYEAR	No	Year, spelled out; "S" prefixes BC dates with "-".
YYYY SYYYY	Yes	4-digit year; "S" prefixes BC dates with "-".
YYY YY Y	Yes	Last 3, 2, or 1 digit(s) of year.

Oracle returns an error if an alphanumeric character is found in the date string where punctuation character is found in the format string. For example:

```
TO_CHAR (TO_DATE('0297','MM/YY'), 'MM/YY')
```

returns an error.

Date Format Elements and National Language Support

The functionality of some date format elements depends on the country and language in which you are using Oracle. For example, these date 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 date format elements are always in English.

The date 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.

For information on these initialization parameters, see *Oracle8 Reference*.

ISO Standard Date Format Elements

Oracle calculates the values returned by the date 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 date format elements YYYY, YYY, YY, Y, and WW, see the discussion of national language support in *Oracle8 Reference*.

The RR Date Format Element

The RR date format element is similar to the YY date format element, but it provides additional flexibility for storing date values in other centuries. The RR date format element allows you to store 21st century dates in the 20th century by

specifying only the last two digits of the year. It will also allow you to store 20th century dates in the 21st century in the same way if necessary.

If you use the TO_DATE function with the YY date format element, the date value returned is always in the current century. If you use the RR date format element instead, the century of the return value varies according to the specified two-digit year and the last two digits of the current year. Table 3–16 summarizes the behavior of the RR date format element.

Table 3–16 The RR Date Element Format

		If the specified two-digit year is	
		0 - 49	**50 - 99**
If the last two digits of the current year are:	**0-49**	The return date is in the current century.	The return date is in the preceding century.
	50-99	The return date is in the next century.	The return date is in the current century.

The following example demonstrates the behavior of the RR date format element.

Example 1 Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR') ,'YYYY') "Year"
     FROM DUAL;

Year
----
1995

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year";
     FROM DUAL;

Year
----
2017
```

Example 2 Assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR') ,'YYYY') "Year";
  FROM DUAL;

Year
----
1995
```

```
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 date format element allows you to write SQL statements that will return the same values after the turn of the century.

Date Format Element Suffixes

Table 3–17 lists suffixes that can be added to date format elements:

Table 3–17 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

When you add one of these suffixes to a date format element, the return value is always in English.

> **Note:** Date suffixes are valid only on output and cannot be used to insert a date into the database.

Capitalization of 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 Date Format Models

You can also 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.

Format Model Modifiers

You can use the FM and FX modifiers in format models for the TO_CHAR function to 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". This modifier suppresses blank padding in the return value of the TO_CHAR function:

- In a date format element of a TO_CHAR function, this modifier suppresses blanks in subsequent character elements (such as MONTH) and suppresses leading and trailing zeroes for subsequent number elements (such as MI) in a date format model. Without FM, the result of a character element is always right padded with blanks to a fixed length, and leading zeroes are always returned for a number element. With FM, because there is no blank padding, the length of the return value may vary
- In a number format element of a TO_CHAR function, this modifier suppresses blanks added to the left of the number, so that the result is left-justified in the output buffer. Without FM, the result is always right-justified in the buffer, resulting in blank-padding to the left of the number.

FX "Format exact". This modifier specifies exact matching for the character argument and date 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 zeroes.

  When FX is enabled, you can disable this check for leading zeroes by using the FM modifier as well.

If any portion of the character argument violates any of these conditions, Oracle returns an error message.

Example 1 The following statement uses a date format model to return a character expression:

```
SELECT TO_CHAR(SYSDATE, 'fmDDTH')||' of '||TO_CHAR
   (SYSDATE, 'Month')||', '||TO_CHAR(SYSDATE, 'YYYY') "Ides"
    FROM DUAL;

Ides
------------------
3RD of April, 1995
```

Note that the statement above also uses the FM modifier. If FM is omitted, the month is blank-padded to nine characters:

```
SELECT TO_CHAR(SYSDATE, 'DDTH')||' of '||
   TO_CHAR(Month, YYYY') "Ides"
   FROM DUAL;

Ides
-----------------------
03RD of April    , 1995
```

Example 2 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.

Example 3 Table 3–18 shows whether the following statement meets the matching conditions for different values of *char* and *'fmt'* using FX:

```
UPDATE table
  SET date_column = TO_DATE(char, 'fmt');
```

Table 3–18 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1993'	'DD-MON-YYYY'	Match
' 15! JAN % /1993'	'DD-MON-YYYY'	Error
'15/JAN/1993'	'FXDD-MON-YYYY'	Error
'15-JAN-1993'	'FXDD-MON-YYYY'	Match
'1-JAN-1993'	'FXDD-MON-YYYY'	Error
'01-JAN-1993'	'FXDD-MON-YYYY'	Match
'1-JAN-1993'	'FXFMDD-MON-YYYY'	Match

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values:

- 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. In other words, 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.
- If a match fails between a date format element and the corresponding characters in the date string, Oracle attempts alternative format elements, as shown in Table 3–19.

Table 3–19 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 3–19 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'YY'	'YYYY'
'RR'	'RRRR'

Expressions

An *expression* is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype NUMBER (the same datatype 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 datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in

- the select list of the SELECT command
- a condition of the WHERE and HAVING clauses
- the CONNECT BY, START WITH, and ORDER BY clauses
- the VALUES clause of the INSERT command
- the SET clause of the UPDATE command

For example, you could use an expression in place of the quoted string 'smith' in this UPDATE statement SET clause:

```
SET ename = 'smith';
```

This SET clause has the expression LOWER(ename) instead of the quoted string 'smith':

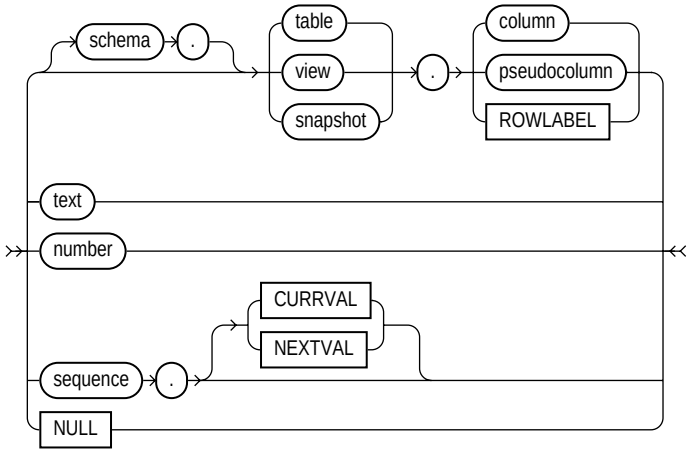
```
SET ename = LOWER(ename);
```

Expressions have several forms. Oracle does not accept all forms of expressions in all parts of all SQL commands. You must use appropriate expression notation whenever *expr* appears in conditions, SQL functions, or SQL commands in other parts of this reference. The description of each command in Chapter 4, "Commands", documents the restrictions on the expressions in the command. The sections that follow describe and provide examples of the various forms of expressions.

Form I

A Form I expression specifies column, pseudocolumn, constant, sequence number, or NULL.

expr_form_I ::=



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 snapshot. Qualifying a public synonym with "PUBLIC" is supported only in data manipulation language (DML) commands, not data definition language (DDL) commands.

The *pseudocolumn* can be either LEVEL, ROWID, or ROWNUM. You can use a pseudocolumn only with a table, not with a view or snapshot. NCHAR and NVARCHAR2 are not valid pseudocolumn or ROWLABEL datatypes. For more information on pseudocolumns, see "Pseudocolumns" on page 2-32.

If you are not using Trusted Oracle, the expression ROWLABEL always returns NULL. For information on using labels and ROWLABEL, see your Trusted Oracle documentation.

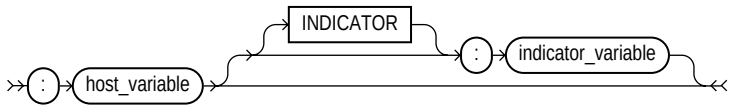
Some valid Form I expressions are:

```
emp.ename
'this is a text string'
10
N'this is an NCHAR string'
```

Form II

A Form II expression specifies a host variable with an optional indicator variable. Note that this form of expression can only appear in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

expr_form_II ::=



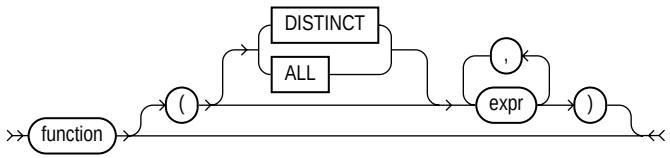
Some valid Form II expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

Form III

A Form III expression specifies a call to a SQL function operating on a single row.

expr_form_III ::=



Some valid Form III expressions are:

```
LENGTH('BLAKE')
ROUND(1234.567*43)
```

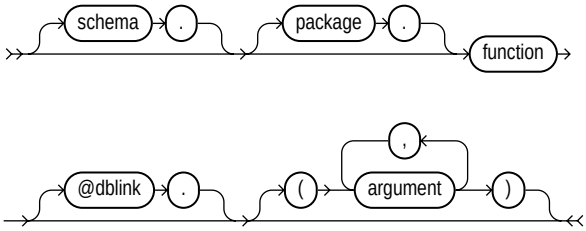
SYSDATE

For information on SQL functions, see “SQL Functions” on page 3-16.

Form IV

A Form IV expression specifies a call to a user function

expr_form_IV ::=



Some valid Form IV expressions are:

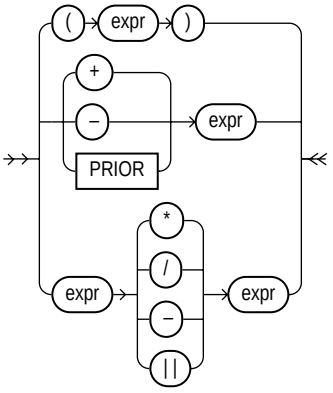
```
circle_area(radius)
payroll.tax_rate(empno)
scott.payrol.tax_rate(dependents, empno)@ny
```

For information on user functions, see “User Functions” on page 3-60.

Form V

A Form V expression specifies a combination of other expressions.

expr_form_V ::=



Note that some combinations of functions are inappropriate and are rejected. For example, the LENGTH function is inappropriate within a group function.

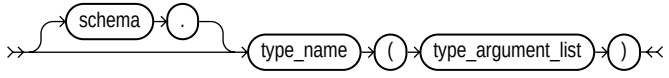
Some valid Form V expressions are:

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate,'DD-MMM-YY')
```

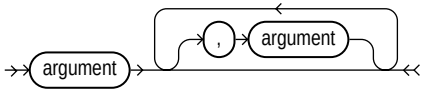
OBJ Form VI

A Form VI expression specifies a call to a type constructor.

expr_form_VI ::=



type_argument_list ::=



If *type_name* is an object type, then the type argument list must be an ordered list of arguments, 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; the maximum number of arguments is 999.

If *type_name* is a VARRAY or nested table type, then the argument list can contain zero or more arguments. Zero arguments imply construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

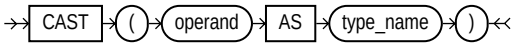
Whether *type_name* is an object type, a VARRAY, or a nested table type, the maximum number of arguments it can contain is 999.

Example

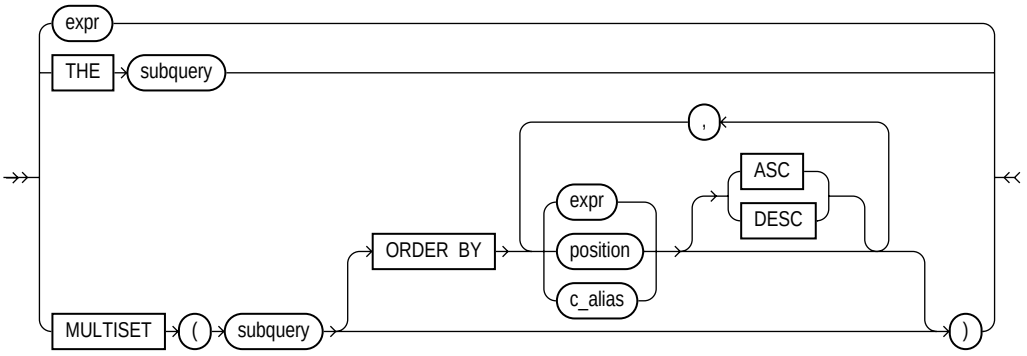
```
CREATE TYPE address_t AS OBJECT
  (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(3), zip NUMBER);
CREATE TYPE address_book_t AS TABLE OF address_t;
DECLARE
  /* Object Type variable initialized via Object Type Constructor */
  myaddr address_t = address_t(500, 'Oracle Parkway', 'Redwood Shores',
                               'CA', 94065);
  /* nested table variable initialized to an empty table via a
     constructor*/
  alladdr address_book_t = address_book_t();
BEGIN
  /* below is an example of a nested table constructor with two elements
     specified, where each element is specified as an object type
     constructor. */
  insert into employee values (666999, address_book_t(address_t(500,
     'Oracle  Parkway', 'Redwood Shores', 'CA', 94065), address_t(400,
     'Mission Street', 'Fremont', 'CA', 94555)));
END;
```

OBJ Form VII

A Form VII expression converts one collection-typed value into another collection-typed value.



operand ::=



CAST allows you to convert collection-typed values of one type into another collection type. You can cast an unnamed collection (such as the result set of a subquery) or a named collection (such as a VARRAY or a nested table) into a type-compatible named collection. The *type_name* must be the name of a collection type and the *operand* must evaluate to a collection value.

To cast a named collection type into another named collection type, the elements of both collections must be of the same type.

If the result set of *subquery* can evaluate to multiple rows, 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, which is not supported in the CAST expression. In other words, scalar subqueries as arguments of the CAST operator are not valid in Oracle8.

The CAST examples that follow use the following user-defined types and tables:

```
CREATE TYPE address_t AS OBJECT
      (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(2));
CREATE TYPE address_book_t AS TABLE OF address_t;
CREATE TYPE address_array_t AS VARRAY(3) OF address_t;
CREATE TABLE emp_address (empno NUMBER, no NUMBER, street CHAR(31),
                          city  CHAR(21), state CHAR(2));
CREATE TABLE employees (empno NUMBER, name CHAR(31));
CREATE TABLE dept (dno NUMBER, addresses address_array_t);
```

Example 1

CAST a subquery:

```
SELECT e.empno, e.name, CAST(MULTISET(SELECT ea.no, ea.street,
                                             ea.city, ea.state
                                        FROM emp_address ea
                                        WHERE ea.empno = e.empno)
                        AS address_book_t)
  FROM employees e;
```

Example 2

CAST converts a VARRAY type column into a nested table. The table values are generated by a flattened subquery. See "Using Flattened Subqueries" on page 4-533.

```
SELECT *
    FROM THE(SELECT CAST(d.addresses AS address_book_t)
                FROM dept d
                  WHERE d.dno = 111) a
     WHERE a.city = 'Redwood Shores';
```

Example 3

The following example casts a MULTISET expression with an ORDER BY clause:

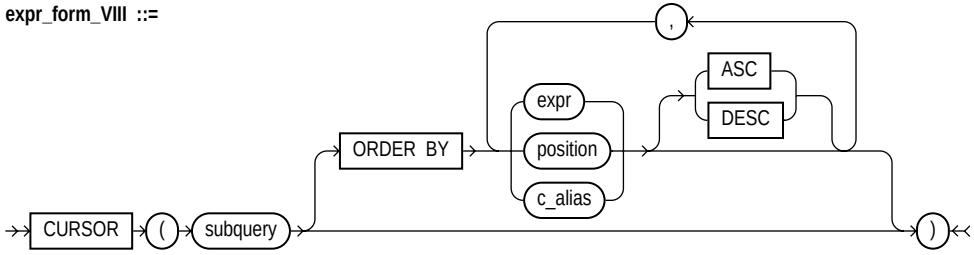
```
CREATE TABLE projects (empid NUMBER, projname VARCHAR2(10));
CREATE TABLE employees (empid NUMBER, ename VARCHAR2(10));
CREATE TYPE projname_table_type AS TABLE OF VARCHAR2(10);
```

An example of a MULTISET expression with the above schema is:

```
SELECT e.name, CAST(MULTISET(SELECT p.projname
                             FROM projects p
                             WHERE p.empid=e.empid
                             ORDER BY p.projname)
  AS projname_table_type)
   FROM employees e;
```

OBJ Form VIII

A Form VIII expression returns a nested CURSOR. This form of expression is similar to the PL/SQL REF cursor.



A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when

- 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)

The following restrictions apply to the CURSOR expression:

- Nested cursors can appear only in a SELECT statement that is not nested in any other query expression, except when it is a subquery of the CURSOR expression itself.
- Nested cursors can appear only in the outermost SELECT list of the query specification.
- Nested cursors cannot appear in views.
- You cannot perform BIND and EXECUTE operations on nested cursors.

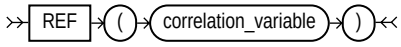
Example

```
SELECT d.deptno, CURSOR(SELECT e.empno, CURSOR(SELECT p.projnum,
                                                      p.projname
                                               FROM   projects p
                                               WHERE  p.empno = e.empno)
                        FROM TABLE(d.employees) e)
  FROM dept d
  WHERE d.dno = 605;
```

OBJ Form IX

A Form IX expression constructs a reference to an object.

expr_form_IX ::=



In a SQL statement, REF takes as its argument a 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. For more information about REFs, see *Oracle8 Concepts*.

Example 1

```
SELECT REF(e)
FROM employee_t e
WHERE e.empno = 10000;
```

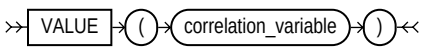
Example 2 This example uses REF in a predicate:

```
SELECT e.name
FROM employee_t
  e INTO :x
WHERE REF(e) = empref1;
```

OBJ Form X

A Form X expression returns the row object.

expr_form_X ::=



In a SQL statement, VALUE takes as its argument a correlation variable (table alias) associated with a row of an object table.

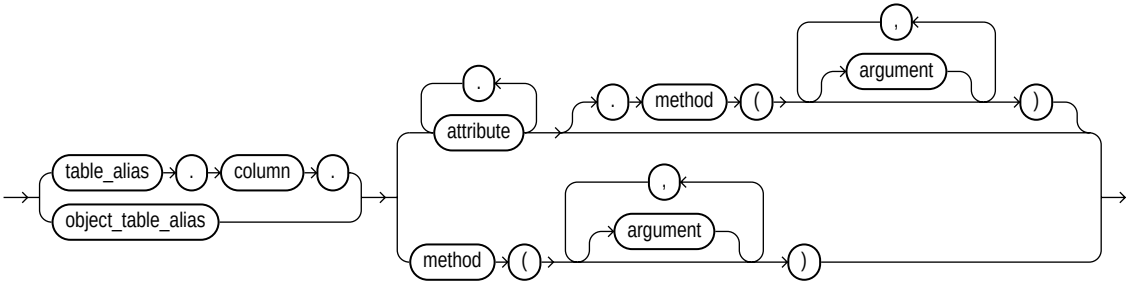
Example

```
SELECT VALUE(e)
  FROM employee e
  WHERE e.name = 'John Smith';
```

OBJ Form XI

A Form XI expression specifies attribute reference and method invocation.

expr_form_XI::=



The `column` parameter can be an object or REF column. Examples in this section use the following user-defined types and tables:

```
CREATE OR REPLACE TYPE employee_t AS OBJECT
    (empid NUMBER,
     name CHAR(31),
     birthdate DATE,
     MEMBER FUNCTION age RETURN NUMBER,
     PRAGMA RESTRICT REFERENCES(age, RNPS, WNPS, WNDS)
    );
CREATE OR REPLACE TYPE BODY employee_t AS
 MEMBER FUNCTION age RETURN NUMBER IS
   var NUMBER;
   BEGIN
     var := months_between(ROUND(SYSDATE, 'YEAR'),
            ROUND(birthdate, 'YEAR'))/12;
     RETURN(var);
   END;
 END; /
CREATE TABLE department (dno NUMBER, manager EMPLOYEE_T);
```

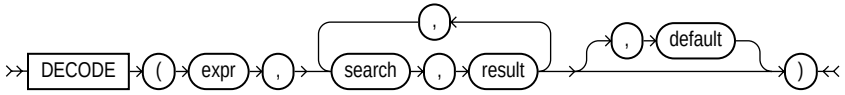
Examples The following examples update and select from the object columns and method defined above.

```
UPDATE department d
  SET d.manager.empid = 100;

SELECT d.manager.name, d.manager.age()
  FROM department d;
```

Decoded Expression

A decoded expression uses the special DECODE syntax:

decode_expr ::=



To evaluate this expression, Oracle compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, Oracle returns the corresponding *result*. If no match is found, Oracle returns *default*, or, if *default* is omitted, returns null. If *expr* and *search* contain character data, Oracle compares them using nonpadded comparison semantics. For information on these semantics, see the section"Datatype Comparison Rules" on page 2-24.

The *search*, *result*, and *default* values can be derived from expressions. Oracle 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 datatype of the first *search* value before comparing. Oracle automatically converts the return value to the same datatype as the first *result*. If the first *result* has the datatype CHAR or if the first *result* is null, then Oracle converts the return value to the datatype VARCHAR2. For information on datatype conversion, see "Data Conversion" on page 2-28.

In a DECODE expression, Oracle considers two nulls to be equivalent. If *expr* is null, Oracle returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE expression, including *expr*, *search*es, *result*s, and *default* is 255.

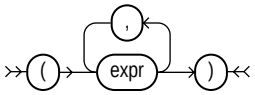
Example This expression decodes the value DEPTNO. If DEPTNO is 10, the expression evaluates to 'ACCOUNTING'; if DEPTNO is 20, it evaluates to 'RESEARCH'; etc. If DEPTNO is not 10, 20, 30, or 40, the expression returns 'NONE'.

```
DECODE (deptno,10, 'ACCOUNTING',
               20, 'RESEARCH',
               30, 'SALES',
               40, 'OPERATION',
                   'NONE')
```

List of Expressions

A list of expressions is a parenthesized series of expressions separated by a comma.

expr_list ::=



An expression list can contain up to 1000 expressions. Some valid expression lists are:

```
10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
(LENGTH('MOOSE') * 57, -SQRT(144) + 72, 69)
```

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or unknown. You must use this syntax whenever *condition* appears in SQL commands in Chapter 4, "Commands".

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 command:

- WHERE
- START WITH
- CONNECT BY
- HAVING

A condition could be said to be of the "logical" datatype, although Oracle does not formally support such a datatype.

The following simple condition always evaluates to TRUE:

```
1 = 1
```

The following more complex condition adds the SAL value to the COMM value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 2500:

```
NVL(sal, 0) + NVL(comm, 0) > 2500
```

Logical operators can combine multiple conditions into a single condition. For example, you can use the AND operator to combine two conditions:

```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

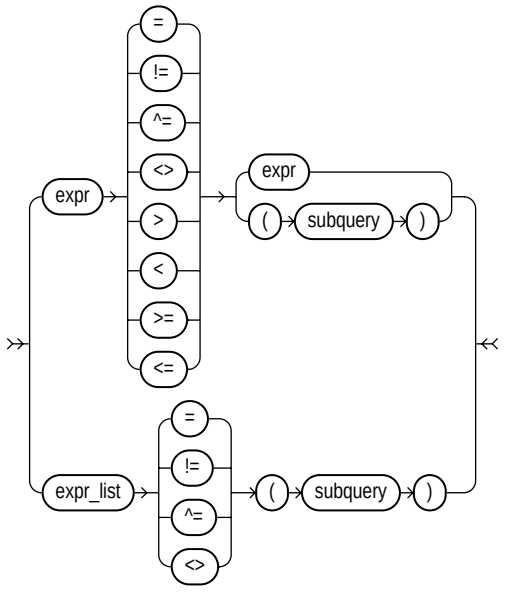
```
name = 'SMITH'
emp.deptno = dept.deptno
hiredate > '01-JAN-88'
job IN ('PRESIDENT', 'CLERK', 'ANALYST')
sal BETWEEN 500 AND 1000
comm IS NULL AND sal = 2000
```

Conditions can have several forms. The description of each command in Chapter 4, "Commands", documents the restrictions on the conditions in the command. The sections that follow describe the various forms of conditions.

Form I

A Form I condition specifies a comparison with expressions or subquery results.

condition_form_I ::=

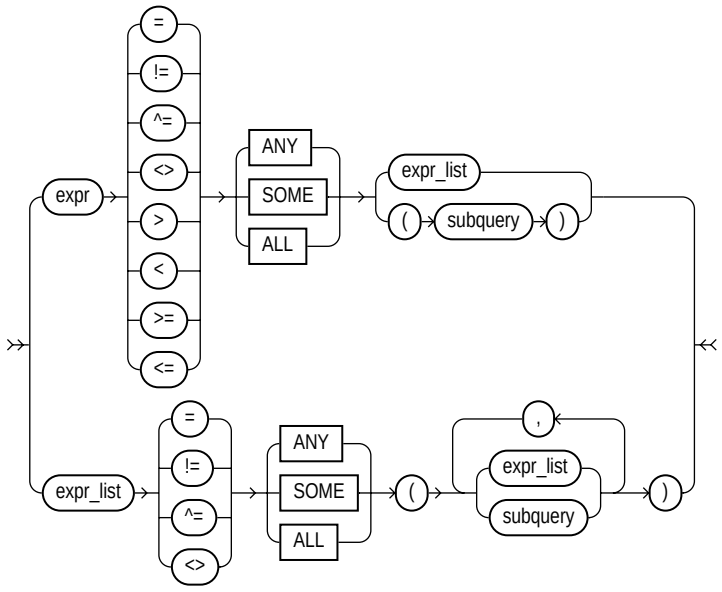


For information on comparison operators, see “Comparison Operators” on page 3-5.

Form II

A Form II condition specifies a comparison with any or all members in a list or subquery.

condition_form_II ::=

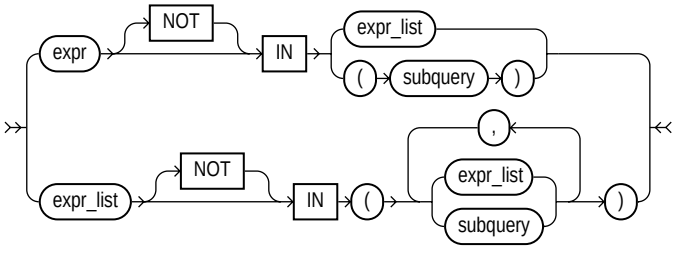


See “Subqueries” on page 4-530.

Form III

A Form III condition tests for membership in a list or subquery.

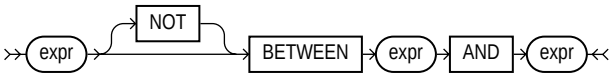
condition_form_III ::=



Form IV

A Form IV condition tests for inclusion in a range.

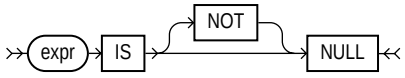
condition_form_IV ::=



Form V

A Form V condition tests for nulls.

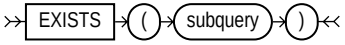
condition_form_V ::=



Form VI

A Form VI condition tests for existence of rows in a subquery.

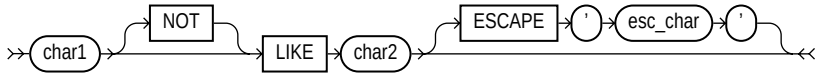
condition_form_VI ::=



Form VII

A Form VII condition specifies a test involving pattern matching.

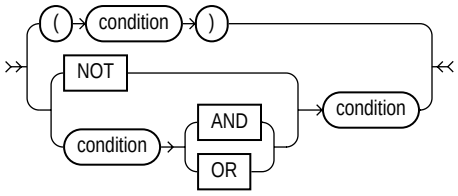
condition_form_VII ::=



Form VIII

A Form VIII condition specifies a combination of other conditions.

condition_form_VIII ::=



4

Commands

This chapter describes, in alphabetical order, Oracle SQL commands and clauses.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

The description of each command or clause contains the following sections:

Purpose	describes the basic uses of the command.
Prerequisites	lists privileges you must have and steps that you must take before using the command. In addition to the prerequisites listed, most commands also require that the database be opened by your instance, unless otherwise noted.
Syntax	shows the keywords and parameters that make up the command.
Keywords and Parameters	describes the purpose of each keyword and parameter. The conventions for keywords and parameters used in this chapter are explained in the Preface of this reference. Usage notes: Optional sections following "Keywords and Parameters" provide examples and discuss how and when to use the command.
Related Topics	lists related commands, clauses, and sections of this and other manuals.

Summary of SQL Commands

The tables in the following sections provide a functional summary of SQL commands and are divided into these categories:

- Data Definition Language (DDL) Commands
- Data Manipulation Language (DML) Commands
- Transaction Control Commands
- Session Control Commands
- System Control Command

Data Definition Language (DDL) Commands

Data definition language (DDL) commands enable 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 object being acted upon. For example, an ALTER TABLE command fails if another user has an open transaction on the specified table.

The GRANT, REVOKE, ANALYZE, AUDIT, and COMMENT commands do not required exclusive access to the object being acted upon. For example, you can analyze a table while other users are updating the table.

Oracle implicitly commits the current transaction before and after every DDL statement.

Many DDL statements may cause Oracle to recompile or reauthorize schema objects. For information on how Oracle recompiles and reauthorizes schema objects and the circumstances under which a DDL statement would cause this, see *Oracle8 Concepts*.

DDL commands are not directly supported by PL/SQL, but may be available using packaged procedures supplied by Oracle corporation. For more information, see *PL/SQL User's Guide and Reference*.

Table 4–1 lists the DDL commands.

Table 4–1 Data Definition Language Commands

Command	Purpose
ALTER CLUSTER	Change the storage characteristics of a cluster.
	Allocate an extent for a cluster.
ALTER DATABASE	Open / mount the database.
	Convert an Oracle7 data dictionary when migrating to Oracle8.
	Prepare to downgrade to an earlier release of Oracle.
	Choose ARCHIVELOG / NOARCHIVELOG mode.
	Perform media recovery.
	Add / drop / clear redo log file groups members.
	Rename a datafile / redo log file member.
	Back up the current control file.
	Back up SQL commands (that can be used to re-create the database) to the trace file.
	Create a new datafile.
	Resize one or more datafiles.
	Create a new datafile in place of an old one for recovery purposes.
	Enable / disable autoextending the size of datafiles.
	Take a datafile online / offline.
	Enable / disable a thread of redo log file groups.
	Change the database's global name.
ALTER FUNCTION	Recompile a stored function.
ALTER INDEX	Redefine an index's future storage allocation.
ALTER PACKAGE	Recompile a stored package.
ALTER PROCEDURE	Recompile a stored procedure.
ALTER PROFILE	Add or remove a resource limit to or from a profile.
ALTER RESOURCE COST	Specify a formula to calculate the total cost of resources used by a session.
ALTER ROLE	Change the authorization needed to access a role.

Table 4–1 (Cont.) Data Definition Language Commands

Command	Purpose
ALTER ROLLBACK SEGMENT	Change a rollback segment's storage characteristics.
	Bring a rollback segment online / offline.
	Shrink a rollback segment to an optimal or given size.
ALTER SEQUENCE	Redefine value generation for a sequence.
ALTER SNAPSHOT	Change a snapshot's storage characteristics, automatic refresh time, or automatic refresh mode.
ALTER SHAPSHOT LOG	Change a snapshot log's storage characteristics.
ALTER TABLE	Add a column / integrity constraint to a table.
	Redefine a column, to change a table's storage characteristics.
	Enable / disable / drop an integrity constraint.
	Enable / disable table locks on a table.
	Enable / disable all triggers on a table.
	Allocate an extent for the table.
	Allow / disallow writing to a table.
	Modify the degree of parallelism for a table.
ALTER TABLESPACE	Add / rename datafiles.
	Change storage characteristics.
	Take a tablespace online / offline.
	Begin / end a backup.
	Allow / disallow writing to a tablespace.
ALTER TRIGGER	Enable / disable a database trigger.
ALTER TYPE	Change a user-defined type.
ALTER USER	Change a user's password, default tablespace, temporary tablespace, tablespace quotas, profile, or default roles.
ALTER VIEW	Recompile a view.
ANALYZE	Collect performance statistics, validate structure, or identify chained rows for a table, cluster, or index.
AUDIT	Choose auditing for specified SQL commands or operations on schema objects.

Table 4–1 (Cont.) Data Definition Language Commands

Command	Purpose
COMMENT	Add a comment about a table, view, snapshot, or column to the data dictionary.
CREATE CLUSTER	Create a cluster that can contain one or more tables.
CREATE CONTROLFILE	Re-create a control file.
CREATE DATABASE	Create a database.
CREATE DATABASE LINK	Create a link to a remote database.
CREATE DIRECTORY	Create a directory database object for administering access to and use of BFILEs stored outside the database.
CREATE FUNCTION	Create a stored function.
CREATE INDEX	Create an index for a table or cluster.
CREATE LIBRARY	Create a library from which SQL and PL/SQL can call external third-generation language (3GL) functions and procedures.
CREATE PACKAGE	Create the specification of a stored package.
CREATE PACKAGE BODY	Create the body of a stored package
CREATE PROCEDURE	Create a stored procedure.
CREATE PROFILE	Create a profile and specify its resource limits.
CREATE ROLE	Create a role.
CREATE ROLLBACK SEGMENT	Create a rollback segment.
CREATE SCHEMA	Issue multiple CREATE TABLE, CREATE VIEW, and GRANT statements in a single transaction.
CREATE SEQUENCE	Create a sequence for generating sequential values.
CREATE SHAPSHOT	Create a snapshot of data from one or more remote master tables.
CREATE SNAPSHOT LOG	Create a snapshot log containing changes made to the master table of a snapshot.
CREATE SYNONYM	Create a synonym for a schema object.
CREATE TABLE	Create a table, defining its columns, integrity constraints, and storage allocation.
CREATE TABLESPACE	Create a place in the database for storage of schema objects, rollback segments, and temporary segments, naming the datafiles that make up the tablespace.

Table 4–1 (Cont.) Data Definition Language Commands

Command	Purpose
CREATE TRIGGER	Create a database trigger.
CREATE TYPE	Create an object type, named varying array (VARRAY), nested table type, or an incomplete object type.
CREATE USER	Create a database user.
CREATE VIEW	Define a view of one or more tables or views.
DROP CLUSTER	Remove a cluster from the database.
DROP DATABASE LINK	Remove a database link.
DROP DIRECTORY	Remove a directory from the database.
DROP FUNCTION	Remove a stored function from the database.
DROP INDEX	Remove an index from the database.
DROP LIBRARY	Remove a library object from the database.
DROP PACKAGE	Remove a stored package from the database.
DROP PROCEDURE	Remove a stored procedure from the database.
DROP PROFILE	Remove a profile from the database.
DROP ROLE	Remove a role from the database.
DROP ROLLBACK SEGMENT	Remove a rollback segment from the database.
DROP SEQUENCE	Remove a sequence from the database.
DROP SNAPSHOT	Remove a snapshot from the database.
DROP SNAPSHOT LOG	Remove a snapshot log from the database.
DROP SYNONYM	Remove a synonym from the database.
DROP TABLE	Remove a table from the database.
DROP TABLESPACE	Remove a tablespace from the database.
DROP TRIGGER	Remove a trigger from the database.
DROP TYPE	Remove a user-defined type from the database.
DROP USER	Remove a user and the objects in the user's schema from the database.
DROP VIEW	Remove a view from the database.
GRANT	Grant system privileges, roles, and object privileges to users and roles.

Table 4–1 (Cont.) Data Definition Language Commands

Command	Purpose
NOAUDIT	Disable auditing by reversing, partially or completely, the effect of a prior AUDIT statement.
RENAME	Change the name of a schema object.
REVOKE	Revoke system privileges, roles, and object privileges from users and roles.
TRUNCATE	Remove all rows from a table or cluster and free the space that the rows used.

Data Manipulation Language (DML) Commands

Data manipulation language (DML) commands query and manipulate data in existing schema objects. These commands do not implicitly commit the current transaction.

Table 4–2 Data Manipulation Language Commands

Command	Purpose
DELETE	Remove rows from a table.
EXPLAIN PLAN	Return the execution plan for a SQL statement.
INSERT	Add new rows to a table.
LOCK TABLE	Lock a table or view, limiting access to it by other users.
SELECT	Select data in rows and columns from one or more tables.
UPDATE	Change data in a table.

All DML commands except the EXPLAIN PLAN command are supported in PL/SQL.

Transaction Control Commands

Transaction control commands manage changes made by DML commands.

Table 4–3 Transaction Control Commands

Command	Purpose
`COMMIT`	Make permanent the changes made by statements issued since the beginning of the current transaction.
`ROLLBACK`	Undo all changes since the beginning of a transaction or since a savepoint.
`SAVEPOINT`	Establish a point back to which you may roll.
`SET TRANSACTION`	Establish properties for the current transaction.

All transaction control commands except certain forms of the COMMIT and ROLLBACK commands are supported in PL/SQL. For information on the restrictions, see COMMIT on page 4-185 and ROLLBACK on page 4-484.

Session Control Commands

Session control commands dynamically manage the properties of a user session. These commands do not implicitly commit the current transaction.

PL/SQL does not support session control commands.

Table 4–4 Session Control Commands

Command	Purpose
`ALTER SESSION`	Enable/disable the SQL trace facility.
	Enable/disable global name resolution.
	Change the values of the session's NLS parameters.
	In a parallel server, indicate that the session must access database files as if the session were connected to another instance.
	Close a database link.
	Send advice to remote databases for forcing an in-doubt distributed transaction.
	Permit or prohibit procedures and stored procedures from issuing COMMIT and ROLLBACK statements.
	Change the goal of the cost-based optimization approach.
`SET ROLE`	Enable/disable roles for the current session.

System Control Command

The single system control command dynamically manages the properties of an Oracle instance. This command does not implicitly commit the current transaction.

ALTER SYSTEM is not supported in PL/SQL.

Table 4–5 System Control Command

Command	Purpose
`ALTER SYSTEM`	Alter the Oracle instance by performing a specialized function.

Embedded SQL Commands

Embedded SQL commands 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 Precompiler Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*
- *SQL*Module for Ada Programmer's Guide*

ALTER CLUSTER

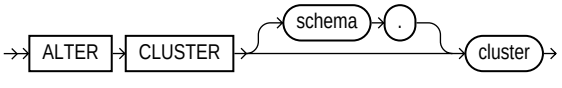
Purpose

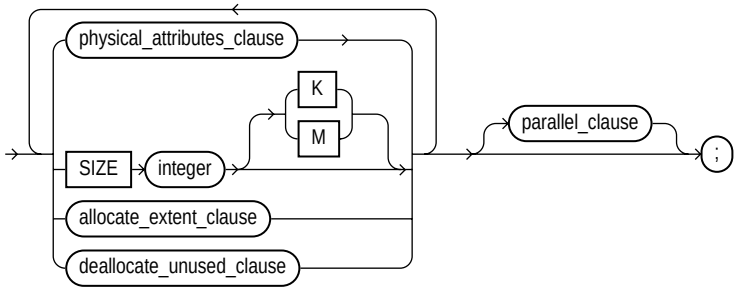
Redefines storage and parallelism characteristics of a cluster. See also "Altering Clusters" on page 4-13.

Prerequisites

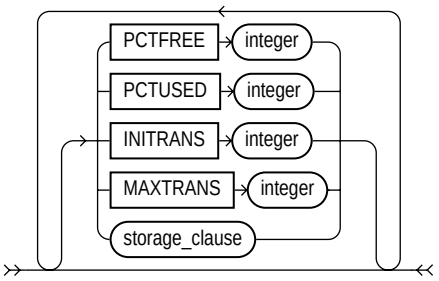
The cluster must be in your own schema or you must have ALTER ANY CLUSTER system privilege.

Syntax

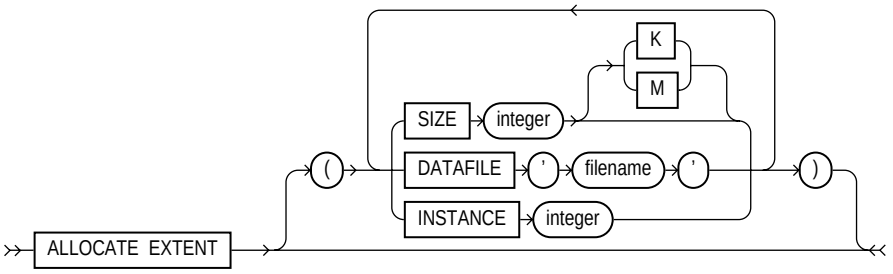




physical_attributes_clause ::=



allocate_extent_clause ::=



deallocate_unused_clause: See the DEALLOCATE UNUSED clause on page 4-372.
parallel_clause: See the PARALLEL clause on page 4-465.

Keywords and Parameters

schema is the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.

cluster is the name of the cluster to be altered.

physical_attributes_clause changes the values of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters of the cluster. See CREATE CLUSTER on page 4-207.

storage_clause changes the storage characteristics for the cluster. See the STORAGE clause on page 4-523.

SIZE determines how many cluster keys will be stored in data blocks allocated to the cluster. You can change the SIZE parameter only for an indexed cluster, not for a hash cluster. For a description of the SIZE parameter, see CREATE CLUSTER on page 4-207.

allocate_extent_clause explicitly allocates a new extent for the cluster.

SIZE specifies the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes. If you omit this parameter, Oracle determines the size based on the values of the cluster's STORAGE parameters.

DATAFILE specifies one of the datafiles in the cluster's tablespace to contain the new extent. If you omit this parameter, Oracle chooses the datafile.

	INSTANCE	makes the new extent available to the specified instance. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit this parameter, the extent is available to all instances. Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode. Explicitly allocating an extent with this clause does not cause Oracle to evaluate the cluster's storage parameters and determine a new size for the next extent to be allocated. You can allocate a new extent only for an indexed cluster, not a hash cluster.
deallocate_unused_clause	explicitly deallocates unused space at the end of the cluster and makes the freed space available for other segments. Only unused space above the high-water mark can be freed. If KEEP is omitted, all unused space is freed. For syntax and complete information, see the DEALLOCATE UNUSED clause on page 4-372.	
	KEEP	specifies the number of bytes above the high-water mark that the cluster will have after deallocation. If the number of remaining extents are 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.
parallel_clause	specifies the degree of parallelism for creating the cluster and the default degree of parallelism for queries on the cluster once created. For syntax and complete information, see the PARALLEL clause on page 4-465.	

Altering Clusters

You can perform these tasks with the ALTER CLUSTER command:

- change the MAXTRANS parameter value for data blocks in the cluster
- change the SIZE, PCTUSED, PCTFREE, and INITRANS parameter values for future data blocks in the cluster
- change future storage characteristics with the STORAGE characteristics NEXT, PCTINCREASE, and MAXEXTENTS
- explicitly allocate an extent
- explicitly deallocate space from unused extents

You cannot perform these tasks with the ALTER CLUSTER command:

- change the number or the name of columns in the cluster key
- change the values of the STORAGE parameters INITIAL and MINEXTENTS
- change the tablespace in which the cluster is stored

- remove tables from a cluster (see DROP CLUSTER on page 4-386 and DROP TABLE on page 4-405)

Example I The following statement alters the CUSTOMER cluster in the schema SCOTT:

```
ALTER CLUSTER scott.customer
    SIZE 512
    STORAGE (MAXEXTENTS 25);
```

Oracle allocates 512 bytes for each cluster key value. Assuming a data block size of 2 kilobytes, future data blocks within this cluster contain 4 cluster keys per data block, or 2 kilobytes divided by 512 bytes.

The cluster can have a maximum of 25 extents.

Example II The following statement deallocates unused space from CUSTOMER cluster, keeping 30 kilobytes of unused space for future use:

```
ALTER CLUSTER scott.customer
    DEALLOCATE UNUSED KEEP 30 K;
```

Related Topics

CREATE CLUSTER on page 4-207
CREATE TABLE on page 4-306
DEALLOCATE UNUSED clause on page 4-372
DROP CLUSTER on page 4-386
DROP TABLE on page 4-405
STORAGE clause on page 4-523
PARALLEL clause on page 4-465

ALTER DATABASE

Purpose

To alter an existing database in one of these ways:

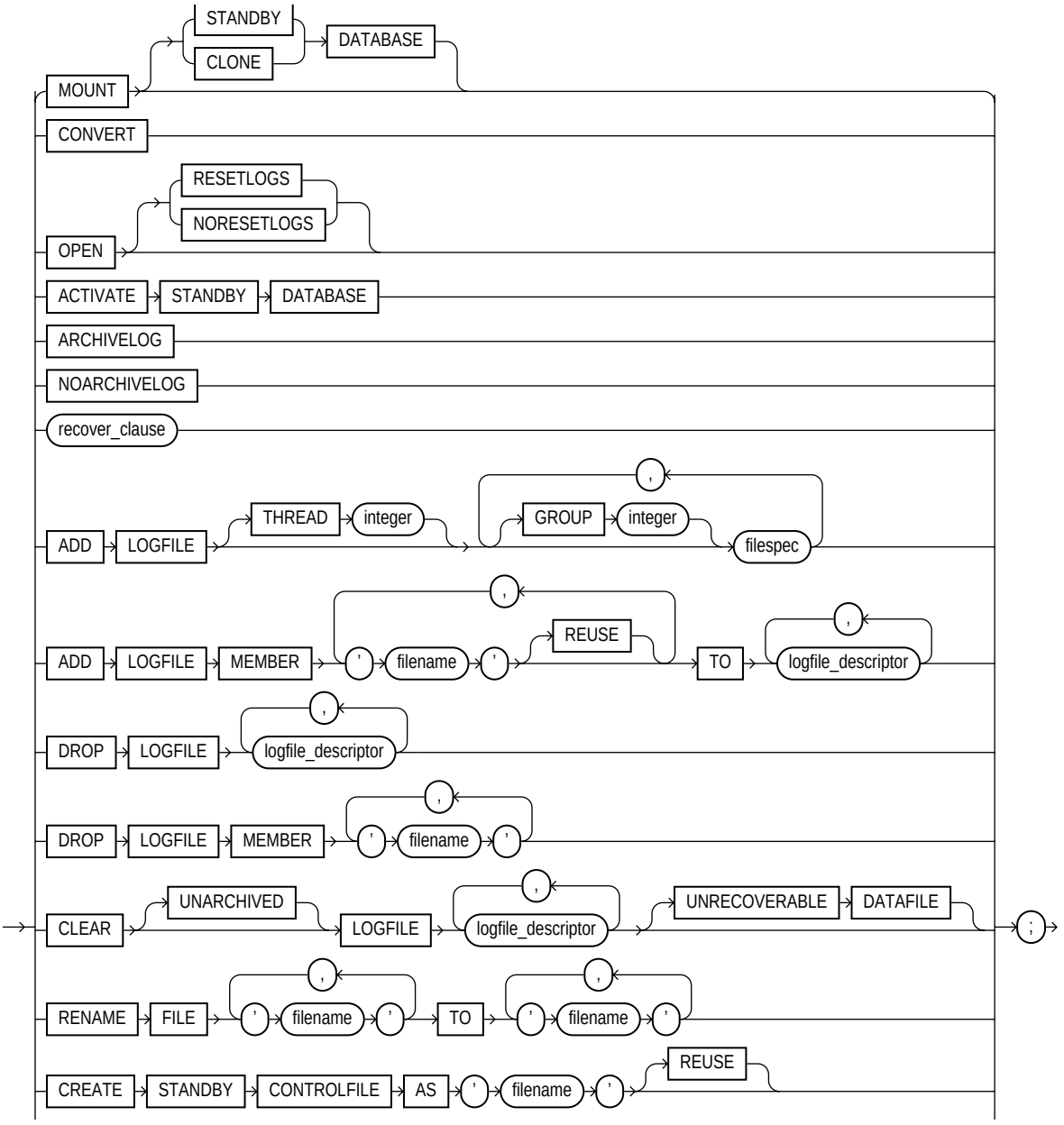
- mount the database, clone, or standby database
- convert an Oracle7 data dictionary when migrating to Oracle8
- open the database
- choose ARCHIVELOG or NOARCHIVELOG mode for redo log file groups
- perform media recovery
- add or drop a redo log file group or a member of a redo log file group
- clear and initialize an online redo log file
- rename a redo log file member or a datafile
- back up the current control file
- back up SQL commands (that can be used to re-create the database) to the database's trace file
- take a datafile online or offline
- enable or disable a thread of redo log file groups
- change the database's global name
- prepare to downgrade to an earlier release of Oracle
- resize one or more datafiles
- create a new datafile in place of an old one for recovery purposes
- enable or disable the autoextending of the size of datafiles

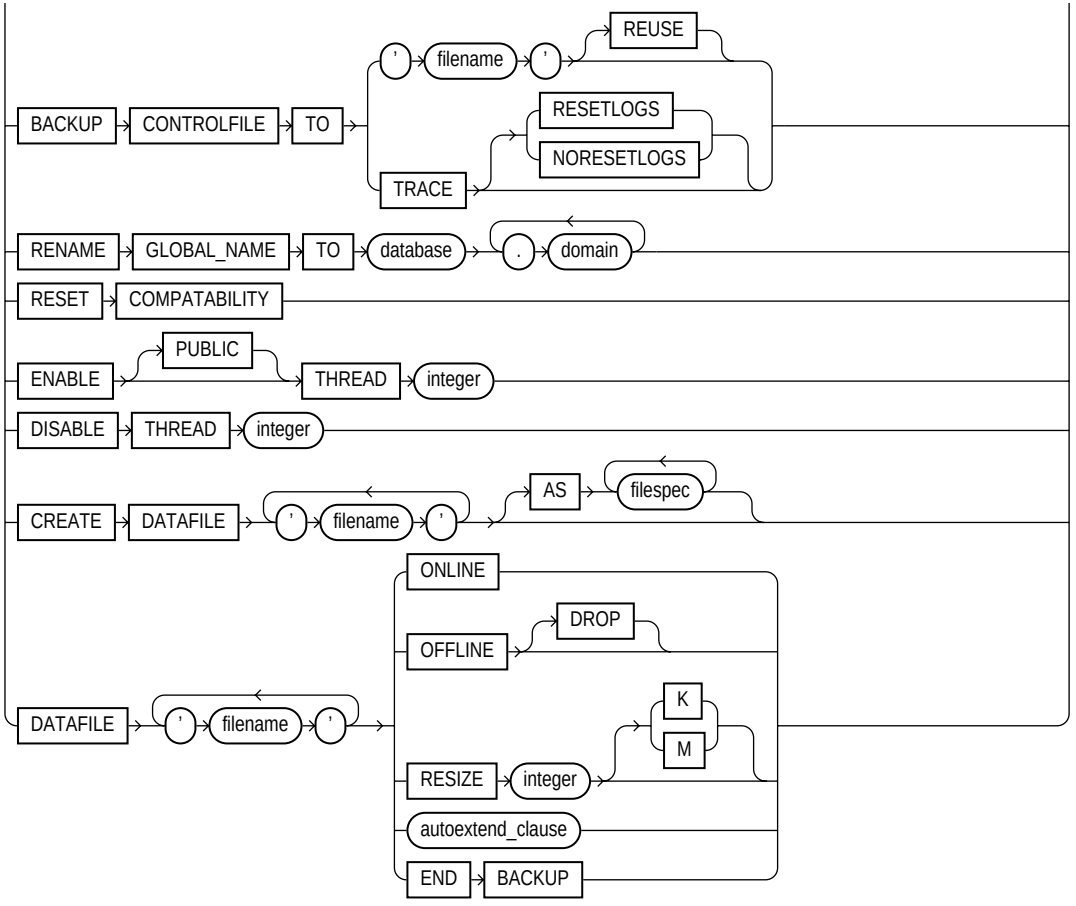
For illustrations of some of these purposes, see "Examples" on page 4-23.

Prerequisites

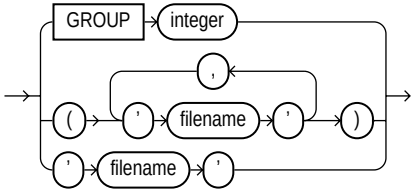
You must have ALTER DATABASE system privilege.

Syntax

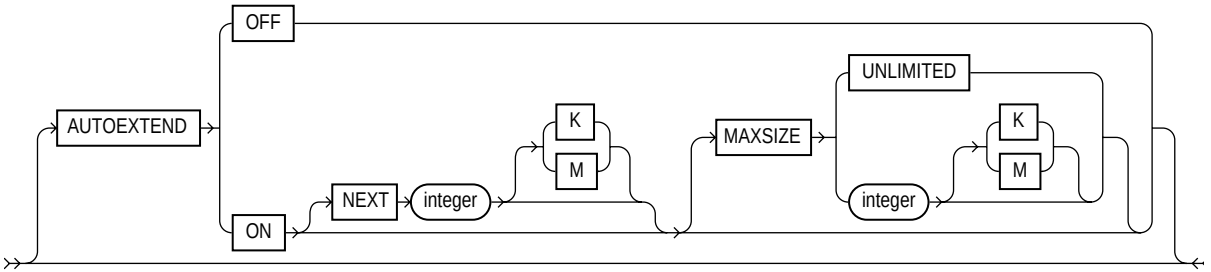

STANDBY
CLONE
DATABASE
MOUNT
CONVERT
RESETLOGS
NORESETLOGS
OPEN
ACTIVATE
STANDBY
DATABASE
ARCHIVELOG
NOARCHIVELOG
recover_clause
,
ADD
LOGFILE
THREAD
integer
GROUP
integer
filespec
,
ADD
LOGFILE
MEMBER
'
filename
'
REUSE
TO
,
logfile_descriptor
DROP
LOGFILE
,
logfile_descriptor
DROP
LOGFILE
MEMBER
,
'
filename
'
CLEAR
UNARCHIVED
LOGFILE
,
logfile_descriptor
UNRECOVERABLE
DATAFILE
;
RENAME
FILE
,
'
filename
'
TO
,
'
filename
'
CREATE
STANDBY
CONTROLFILE
AS
'
filename
'
REUSE



logfile_descriptor::=



autoextend_clause::=



recover_clause: See the RECOVER clause on page 4-469.

Keywords and Parameters

database identifies the database to be altered. The database name can contain only ASCII characters. If you omit database, Oracle 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. Note that the database identifier is not related to the Net8 database specification.

You can use the following options only when the database is not mounted by your instance:

MOUNT mounts the database.

- STANDBY DATABASE mounts the standby database. For more information, see the *Oracle8 Administrator's Guide.*
- CLONE DATABASE mounts the clone database. For more information, see the *Oracle8 Backup and Recovery Guide.*

CONVERT completes the conversion of the Oracle7 data dictionary. After you use this option, the Oracle7 data dictionary no longer exists in the Oracle database. Use this option only when you are migrating to Oracle8. For more information on using this option, see *Oracle8 Migration*.

OPEN opens the database, making it available for normal use. You must mount the database before you can open it. You cannot open a standby database that has not been activated.

- RESETLOGS resets the current log sequence number to 1 and discards any redo information that was not applied during recovery, ensuring that it will never be applied. This effectively discards all changes that are in the redo log, but not in the database. You must use this option to open the database after performing media recovery with an incomplete recovery using the RECOVER UNTIL clause (see RECOVER clause on page 4-469) or with a backup control file. After opening the database with this option, you should perform a complete database backup.

NORESETLOGS leaves the log sequence number and redo log files in their current state.

You can specify the above options only after performing incomplete media recovery or complete media recovery with a backup control file. In any other case, Oracle uses the NORESETLOGS automatically.

ACTIVATE STANDBY DATABASE changes the state of a standby database to an active database. For more information, see *Oracle8 Administrator's Guide*.

Use the following options only if your instance has the database mounted in parallel server disabled mode, but not open:

ARCHIVELOG establishes ARCHIVELOG mode for redo log file groups. In this mode, the contents of a redo log file group must be archived before the group can be reused. This option prepares for the possibility of media recovery. You can use this option only after shutting down your instance normally or immediately with no errors and then restarting it, mounting the database in parallel server disabled mode.

NOARCHIVELOG establishes NOARCHIVELOG mode for redo log files. In this mode, the contents of a redo log file group need not be archived so that the group can be reused. This mode does not prepare for recovery after media failure.

You can use any of the following options when your instance has the database mounted, open or closed, and the files involved are not in use:

recover_clause performs media recovery. For syntax and more information, see the RECOVER clause on page 4-469. You recover the entire database only when the database is closed. You can recover tablespaces or datafiles when the database is open or closed, provided the tablespaces or datafiles to be recovered are offline. You cannot perform media recovery if you are connected to Oracle through the multithreaded server architecture. You can also perform media recovery with the Server Manager recovery dialog box.

ADD LOGFILE adds one or more redo log file groups to the specified thread, making them available to the instance assigned the thread.

THREAD is required only if you are using Oracle with the Parallel Server option in parallel mode. If you omit the THREAD parameter, the redo log file group is added to the thread assigned to your instance.

GROUP uniquely identifies the redo log file group among all groups in all threads and can range from 1 to the MAXLOGFILES value. You cannot add multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

	filespec	Each *filespec* specifies a redo log file group containing one or more members, or copies. See the syntax description of *filespec* in "Filespec" on page 4-431.

ADD LOGFILE MEMBER adds new members to existing redo log file groups. Each new member is specified by *'filename'*. If the file already exists, it must be the same size as the other group members, and you must specify the REUSE option. If the file does not exist, Oracle creates a file of the correct size. You cannot add a member to a group if all of the group's members have been lost through media failure.

You can specify an existing redo log file group in one of these ways:

GROUP	Specify the value of the GROUP *parameter* that identifies the redo log file group.
list of filenames	List all members of the redo log file group. You must fully specify each filename according to the conventions of your operating system.

DROP LOGFILE drops all members of a redo log file group. You can specify a redo log file group in the same manner as the ADD LOGFILE MEMBER clause. You cannot drop a redo log file group if it needs archiving or is the currently active group; nor can you drop a redo log file group if doing so would cause the redo thread to contain less than two redo log file groups.

DROP LOGFILE MEMBER drops one or more redo log file members. Each 'filename' must fully specify a member using the conventions for filenames on your operating system.

You cannot use this clause to drop all members of a redo log file group that contains valid data. To perform this operation, use the DROP LOGFILE clause.

CLEAR LOGFILE reinitializes an online redo log, optionally without archiving the redo log. CLEAR LOGFILE is similar to adding and dropping a redo log, except that the command may be issued even if there are only two logs for the thread and also may be issued for the current redo log of a closed thread.

UNARCHIVED	You must specify UNARCHIVED if you want to reuse a redo log that was not archived. **WARNING:** Specifying UNARCHIVED makes backups unusable if the redo log is needed for recovery.

You cannot 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, 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 command is interrupted by a system or instance failure, then the database may hang. If so, this command must be reissued once 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.

UNRECOVERABLE DATAFILE — You must specify UNRECOVERABLE DATAFILE if the database has a datafile that is offline (not for drop) and if the unarchived log to be cleared is needed to recover the datafile before bringing it back online. In this case, you must drop the datafile and the entire tablespace once the CLEAR LOGFILE command completes.

RENAME FILE — renames datafiles or redo log file members. This clause renames only files in the control file; it does not actually rename them on your operating system. You must specify each filename using the conventions for filenames on your operating system.

CREATE STANDBY CONTROLFILE — creates a control file to be used to maintain a standby database. For more information, see *Oracle8 Administrator's Guide.*

BACKUP CONTROLFILE — backs up the current control file.

TO '*filename*' — specifies the file to which the control file is backed up. You must fully specify the *filename* using the conventions for your operating system. If the specified file already exists, you must specify the REUSE option.

TO TRACE — writes SQL statements to the database's trace file rather than making a physical backup of the control file. The SQL commands can be used to start up the database, re-create the control file, and recover and open the database appropriately, based on the created control file.

You can copy the commands from the trace file into a script file, edit the commands as necessary, and use the database if all copies of the control file are lost (or to change the size of the control file).

RESETLOGS — specifies that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN RESETLOGS.

NORESETLOGS — specifies that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN NORESETLOGS.

RENAME GLOBAL_NAME — changes the global name of the database. 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.

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.

For more information on global names, see *Oracle8 Distributed Database Systems*.

RESET COMPATIBILITY — marks the database to be reset to an earlier version of Oracle when the database is next restarted.

Note: RESET COMPATIBILITY works only if you have successfully disabled Oracle features that affect backward compatibility. For more information on downgrading to an earlier version of Oracle, see *Oracle8 Migration*.

You can use the following options only when your instance has the database open:

ENABLE THREAD — in a parallel server, enables the specified thread of redo log file groups. The thread must have at least two redo log file groups before you can enable it.

- PUBLIC — makes the enabled thread available to any instance that does not explicitly request a specific thread with the initialization parameter THREAD. If you omit the PUBLIC option, the thread is available only to the instance that explicitly requests it with the initialization parameter THREAD.

DISABLE THREAD — disables the specified thread, making it unavailable to all instances. You cannot disable a thread if an instance using it has the database mounted.

You can use any of the following options when your instance has the database mounted, open or closed, and the files involved are not in use:

CREATE DATAFILE — creates a new empty datafile in place of an old one. You can use this option to re-create a datafile that was lost with no backup. The '*filename*' must identify a file that is or was once part of the database. The *filespec* specifies the name and size of the new datafile. If you omit the AS clause, Oracle creates the new file with the name and size as the file specified by '*filename*'.

During recovery, all archived redo logs written to since the original datafile was created must be applied to the new, empty version of the lost datafile.

Oracle 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.

You cannot create a new file based on the first datafile of the SYSTEM tablespace.

DATAFILE — affects your database files as follows:

ONLINE	brings the datafile online.
OFFLINE	takes the datafile offline. If the database is open, you must perform media recovery on the datafile before bringing it back online, because a checkpoint is not performed on the datafile before it is taken offline. DROP takes a datafile offline when the database is in NOARCHIVELOG mode.
RESIZE	attempts to change the size of the datafile to the specified absolute size in bytes. You can also use K or M to specify this size in kilobytes or megabytes. There is no default, so you must specify a size.
autoextend_clause	enables or disables the automatic extension of a datafile. If you do not specify this clause, datafiles are not automatically extended. OFF disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER DATABASE AUTOEXTEND commands. ON enables autoextend. NEXT specifies the size in bytes of the next increment of disk space to be automatically allocated to the datafile when more extents are required. You can also use K or M to specify this size in kilobytes or megabytes. The default is one data block. MAXSIZE specifies the maximum disk space allowed for automatic extension of the datafile. UNLIMITED sets no limit on allocating disk space to the datafile.
END BACKUP	avoids media recovery on database startup after an online tablespace backup was interrupted by a system failure or instance failure or SHUTDOWN ABORT.

WARNING: Do not use ALTER TABLESPACE ... END BACKUP if you have restored any of the files affected from a backup. Media recovery is fully described in the *Oracle8 Backup and Recovery Guide* and *Oracle8 Administrator's Guide*.

Examples

For more information on using the ALTER DATABASE command for database maintenance, see the *Oracle8 Administrator's Guide*.

Example I The following statement adds a redo log file group with two members and identifies it with a GROUP parameter value of 3:

```
ALTER DATABASE stocks
  ADD LOGFILE GROUP 3
```

```
    ('diska:log3.log' ,
     'diskb:log3.log') SIZE 50K;
```

Example II The following statement adds a member to the redo log file group added in the previous example:

```
ALTER DATABASE stocks
   ADD LOGFILE MEMBER 'diskc:log3.log'
   TO GROUP 3;
```

Example III The following statement drops the redo log file member added in the previous example:

```
ALTER DATABASE stocks
    DROP LOGFILE MEMBER 'diskc:log3.log';
```

Example IV The following statement renames a redo log file member:

```
ALTER DATABASE stocks
    RENAME FILE 'diskb:log3.log' TO 'diskd:log3.log';
```

The above 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 'DISKB:LOG3.LOG' to 'DISKD:LOG3.LOG'. You must perform this operation through your operating system.

Example V The following statement drops all members of the redo log file group 3:

```
ALTER DATABASE stocks DROP LOGFILE GROUP 3;
```

Example VI The following statement adds a redo log file group containing three members to thread 5 and assigns it a GROUP parameter value of 4:

```
ALTER DATABASE stocks
    ADD LOGFILE THREAD 5 GROUP 4
        ('diska:log4.log',
        'diskb:log4:log',
        'diskc:log4.log' );
```

Example VII The following statement disables thread 5 in a parallel server:

```
ALTER DATABASE stocks
    DISABLE THREAD 5;
```

Example VIII The following statement enables thread 5 in a parallel server, making it available to any Oracle instance that does not explicitly request a specific thread:

```
ALTER DATABASE stocks
    ENABLE PUBLIC THREAD 5;
```

Example IX The following statement creates a new datafile 'DISK1:DB1.DAT' based on the file 'DISK2:DB1.DAT':

```
ALTER DATABASE
    CREATE DATAFILE 'disk1:db1.dat' AS 'disk2:db1.dat';
```

Example XI The following statement changes the global name of the database and includes both the database name and domain:

```
ALTER DATABASE
    RENAME GLOBAL_NAME TO sales.australia.acme.com;
```

Example XII The following statement attempts to change the size of datafile 'DISK1:DB1.DAT':

```
ALTER DATABASE
    DATAFILE 'disk1:db1.dat' RESIZE 10 M;
```

For examples of performing media recovery, see *Oracle8 Administrator's Guide* and *Oracle8 Backup and Recovery Guide.*

Example XIII The following statement clears a log file:

```
ALTER DATABASE
    CLEAR LOGFILE 'disk3:log.dbf';
```

Related Topics

CREATE DATABASE on page 4-219
RECOVER clause on page 4-469
"Filespec" on page 4-431

ALTER FUNCTION

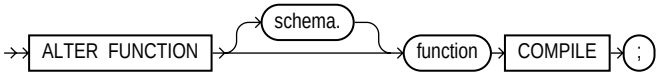
Purpose

To recompile a standalone stored function. See also "Recompiling Standalone Functions" on page 4-26.

Prerequisites

The function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.
function	is the name of the function to be recompiled.
COMPILE	causes Oracle to recompile the function. The COMPILE keyword is required.

Recompiling Standalone Functions

You can use the ALTER FUNCTION command to explicitly recompile a function that is invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The ALTER FUNCTION command is similar to ALTER PROCEDURE on page 4-41. For information on how Oracle recompiles functions and procedures, see *Oracle8 Concepts*.

Note: This command does not change the declaration or definition of an existing function. To redeclare or redefine a function, use the CREATE FUNCTION command with the OR REPLACE option; see CREATE FUNCTION on page 4-232.

Example To explicitly recompile the function GET_BAL owned by the user MERRIWEATHER, issue the following statement:

```
ALTER FUNCTION merriweather.get_bal
COMPILE;
```

If Oracle encounters no compilation errors while recompiling GET_BAL, GET_BAL becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling GET_BAL results in compilation errors, Oracle returns an error message and GET_BAL remains invalid.

Oracle also invalidates all objects that depend upon GET_BAL. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

Related Topics

ALTER PROCEDURE on page 4-41
CREATE FUNCTION on page 4-232

ALTER INDEX

Purpose

Use ALTER INDEX to:

- change storage allocation for, rebuild, or rename an index
- rename, split, remove, mark as unusable, rebuild, or modify physical or logging attributes of a partition of a partitioned index
- modify the physical, parallel, or logging attributes of a nonpartitioned index
- modify the default physical, parallel, or logging attributes of index partitions
- rebuild an index to store the bytes of the index block in reverse order
- OBJ modify a nested table index

For illustrations of some of these purposes, see "Examples" on page 4-35.

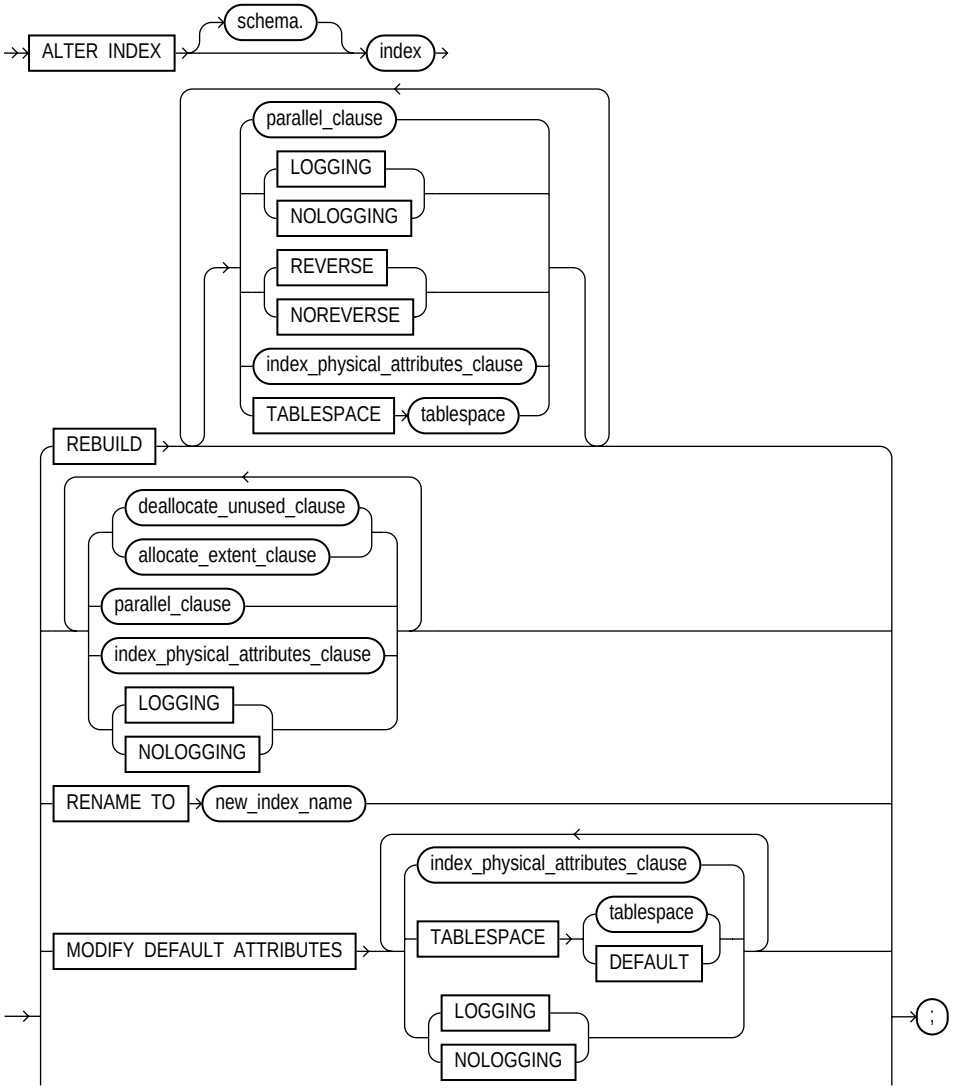
Prerequisites

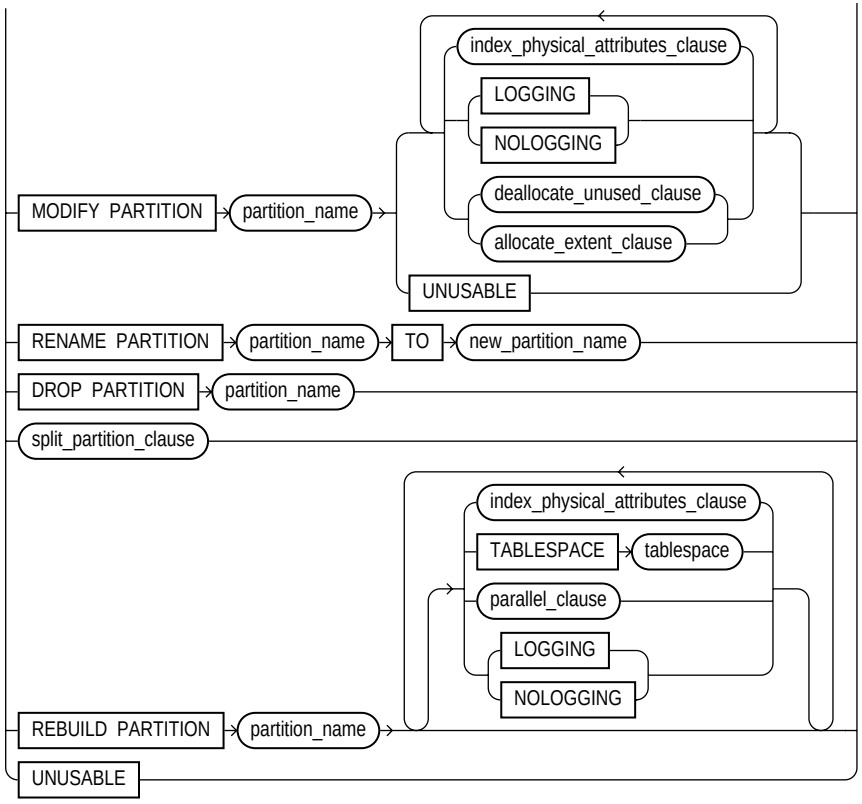
The index must be in your own schema or you must have ALTER ANY INDEX system privilege.

Schema object privileges are granted on the parent index, not on individual index partitions. The following index partition operations require tablespace quota:

- modify
- rebuild
- split

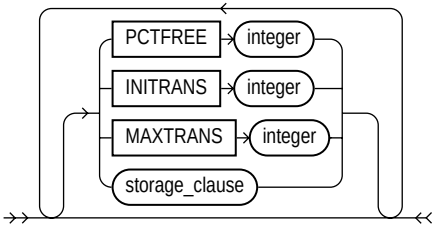
Syntax


ALTER INDEX
schema.
index
parallel_clause
LOGGING
NOLOGGING
REVERSE
NOREVERSE
index_physical_attributes_clause
TABLESPACE
tablespace
REBUILD
deallocate_unused_clause
allocate_extent_clause
parallel_clause
index_physical_attributes_clause
LOGGING
NOLOGGING
RENAME TO
new_index_name
index_physical_attributes_clause
MODIFY DEFAULT ATTRIBUTES
TABLESPACE
tablespace
DEFAULT
LOGGING
NOLOGGING
;



parallel_clause: See PARALLEL clause on page 4-465.

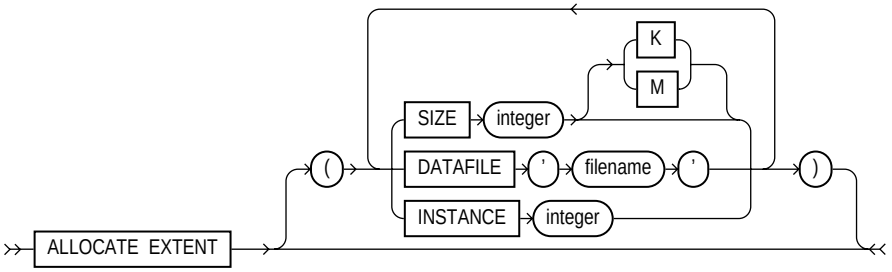
index_physical_attributes_clause ::=



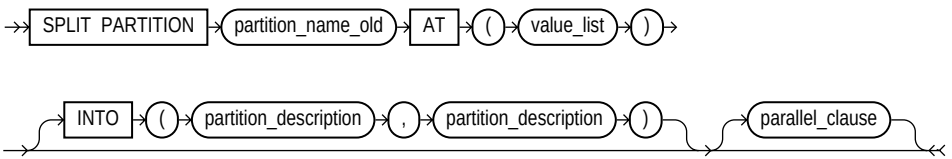
storage_clause: See STORAGE clause on page 4-523.

deallocate_unused_clause: See DEALLOCATE UNUSED clause on page 4-372.

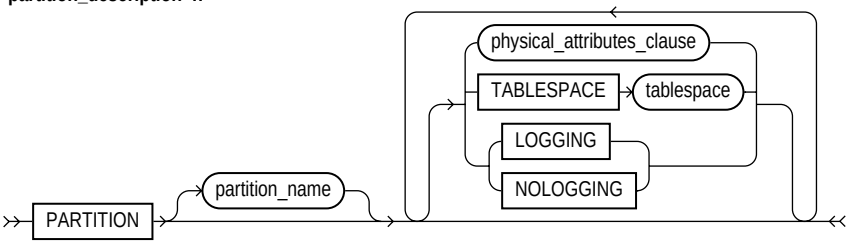
allocate_extent_clause ::=



split_partition_clause ::=



partition_description ::=



Keywords and Parameters

schema is the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index is the name of the index to be altered.

The following operations can be performed only on partitioned indexes:

- drop partition
- split partition
- rename partition
- rebuild partition
- modify partition

Of these, drop partition and split partition can be performed only on global indexes.

REBUILD — re-creates an existing index.

parallel_clause — specifies that rebuilding the index, or some queries against the index or the index partition, are performed either in serial or parallel execution. For information about the syntax of this option and this clause, see the PARALLEL clause on page 4-465. For more information about parallelized operations see *Oracle8 Parallel Server Concepts and Administration*.

LOGGING / NOLOGGING — specifies whether ALTER INDEX...REBUILD (and ALTER INDEX...SPLIT) operations will be logged.

REVERSE — stores the bytes of the index block in reverse order, excluding 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.

index_physical_attributes_clause — changes the values of the PCTFREE, INITRANS, and MAXTRANS parameters for a nonpartitioned index, index partition, or all partitions of a partitioned index, or default values of these parameters for a partitioned index. See these parameters in CREATE TABLE on page 4-306.

Note: You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or to modify a partition (ALTER INDEX ... MODIFY PARTITION). You can change it in all other forms of the ALTER INDEX command.

storage_clause — changes 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. See the STORAGE clause on page 4-523.

TABLESPACE — specifies the tablespace where the rebuilt index or index partition will be stored. The default is the default tablespace of the user issuing the command.

deallocate_unused_clause explicitly deallocates unused space at the end of the index and make the freed space available for other segments. Only unused space above the high-water mark can be freed. If KEEP is omitted, all unused space is freed. See the DEALLOCATE UNUSED clause on page 4-372.

KEEP	specifies the number of bytes above the high-water mark that the index will have after deallocation. If the number of remaining extents are 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.

allocate_extent_clause explicitly allocates a new extent for the index.

SIZE	specifies the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes. If you omit this parameter, Oracle determines the size based on the values of the index's STORAGE parameters.
DATAFILE	specifies one of the data files in the index's tablespace to contain the new extent. If you omit this parameter, Oracle chooses the data file.
INSTANCE	makes the new extent available to the specified instance. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit this parameter, the extent is available to all instances. Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

Explicitly allocating an extent with this clause does affect the size for the next extent to be allocated as specified by the NEXT and PCTINCREASE storage parameters.

LOGGING / NOLOGGING LOGGING / NOLOGGING specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against a nonpartitioned index, index partition, or all partitions of a partitioned index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

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 logged. Therefore, if you cannot afford to lose this index, you must take a backup after the operation in NOLOGGING mode.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the an operation in LOGGING mode will re-create the index. However, media recovery from a backup taken before an operation in NOLOGGING mode will not re-create the index.

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.

For more information about the LOGGING option and parallel DML, see *Oracle8 Concepts* and the *Oracle8 Parallel Server Concepts and Administration*.

Note: The LOGGING / NOLOGGING keywords replace the RECOVERABLE / UNRECOVERABLE option. That option is still available as a valid keyword in Oracle8 when altering or rebuilding nonpartitioned indexes, but its use is not recommended.

RENAME TO — renames *index* to *new_index_name*. The *new_index_name* is a single identifier and does not include the schema name.

MODIFY DEFAULT ATTRIBUTES — is a valid option only for a partitioned index. Use this option to specify new values for the default attributes of a partitioned index.

- TABLESPACE — specifies the tablespace where the default tablespace of a partitioned index will be stored. The default is the default tablespace of the user issuing the command.
- LOGGING / NOLOGGING — specifies the default logging attribute of a partitioned index.

Note: 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 PARTITION — modifies the real physical attributes, logging option, or storage characteristics of index partition *partition_name*; *partition_name* is the name of the index partition to be altered. It must be a partition in *index*.

UNUSABLE — marks the index or index partition(s) as 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.

RENAME PARTITION — renames index *partition_name* to *new_partition_name*.

DROP PARTITION — removes a partition and the data in it from a partitioned global index. Dropping a partition of a global index marks the index's next partition as unusable. You cannot drop the highest partition of a global index.

split_partition_clause — splits a global partitioned index into two partitions, adding a new partition to the index.

Splitting a partition marked as unusable results in two partitions, both marked as unusable. You must rebuild the partitions before you can use them.

Splitting a usable partition results in two partitions populated with index data, both marked as usable.

	AT (*value_list*)	specifies the new noninclusive upper bound for *split_partition_1*. The *value_list* must compare 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	describes the two partitions resulting from the split.
	partition_ description, *partition_ description*	specifies the names and physical attributes of the two partitions resulting from the split.
REBUILD PARTITION	rebuilds one partition of an index. You can also use this option to move an index partition to another tablespace or to change a create-time physical attribute. For more information about partition maintenance operations, see the *Oracle8 Administrator's Guide*.	

Examples

Example I This statement alters SCOTT'S CUSTOMER index so that future data blocks within this index use 5 initial transaction entries and an incremental extent of 100 kilobytes:

```
ALTER INDEX scott.customer
    INITRANS 5
    STORAGE (NEXT 100K);
```

Example II The following example drops index partition IX_ANTARTICA:

```
ALTER INDEX sales_area_ix
  DROP PARTITION ix_antarctica;
```

Example III This statement alters the real attributes of every partition of local partitioned index SALES_IX3. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100 K:

```
ALTER INDEX sales_ix3 INITRANS 5 STORAGE ( NEXT 100K );
```

Example III(a) This statement alters the default attributes of local partitioned index SALES_IX3. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100 K:

```
ALTER INDEX sales_ix3
  MODIFY DEFAULT ATTRIBUTES INITRANS 5 STORAGE ( NEXT 100K );
```

Example IV The following statement marks the IDX_ACCTNO index as UNUSABLE:

```
ALTER INDEX idx_acctno UNUSABLE;
```

Example V The following statement changes the maximum number of extents for partition BRIX_NY:

```
ALTER INDEX branch_ix MODIFY PARTITION brix_ny
  STORAGE( MAXEXTENTS 30 ) LOGGING;
```

Example VI The following example marks partition IDX_FEB96 of index IDX_ACCTNO as UNUSABLE:

```
ALTER INDEX idx_acctno MODIFY PARTITION idx_feb96 UNUSABLE;
```

Example VII The following statement sets the parallel attributes for index ARTIST_IX:

```
ALTER INDEX artist_ix PARALLEL (DEGREE 4, INSTANCES 3);
```

Example VIII The following statement sets the parallel attributes for index ARTIST_IX so that scans on the index will not be parallelized:

```
ALTER INDEX artist_ix NOPARALLEL;
```

Example IX The following statement rebuilds partition P063 in index ARTIST_IX. The rebuilding of the index partition will not be logged:

```
ALTER INDEX artist_ix
  REBUILD PARTITION p063 NOLOGGING;
```

Example X The following example renames an index:

```
ALTER INDEX emp_ix1 RENAME TO employee_ix1;
```

Example XI The following example renames an index partition:

```
ALTER INDEX employee_ix2 RENAME PARTITION emp_ix2_p3
  TO employee_ix2_p3;
```

Example XII The following example splits partition PARTNUM_IX_P6 in partitioned index PARTNUM_IX into PARTNUM_IX_P5 and PARTNUM_IX_P6:

```
ALTER INDEX partnum_ix
  SPLIT PARTITION partnum_ix_p6 AT ( 5001 )
  INTO ( PARTITION partnum_ix_p5 TABLESPACE ts017 LOGGING,
```

```
        PARTITION partnum_ix_p6 TABLESPACE ts004 );
```

Note that the second partition retains the name of the old partition.

Example XIII The following statement rebuilds index EMP_IX so that the bytes of the index block are stored in REVERSE order:

```
ALTER INDEX emp_ix REBUILD REVERSE;
```

Related Topics

CREATE INDEX on page 4-237
CREATE TABLE on page 4-306
PARALLEL clause on page 4-465
STORAGE clause on page 4-523
DEALLOCATE UNUSED clause on page 4-372

ALTER PACKAGE

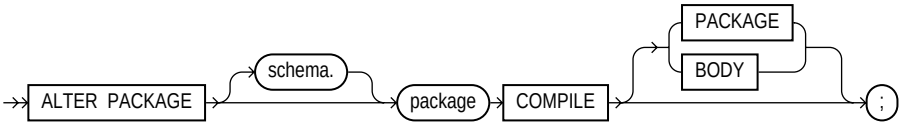
Purpose

To recompile a stored package. See also "Recompiling Stored Packages" on page 4-38.

Prerequisites

The package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.
package	is the name of the package to be recompiled.
COMPILE	recompiles the package specification or body. The COMPILE keyword is required.
PACKAGE	recompiles the package body and specification.
BODY	recompiles only the package body.
	The default option is PACKAGE.

Recompiling Stored Packages

You can use the ALTER PACKAGE command to explicitly recompile either a package specification and body or only a package body. 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 command recompiles all package objects together. You cannot use the ALTER

PROCEDURE command or ALTER FUNCTION command to individually recompile a procedure or function that is part of a package.

Note: This command 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 command with the OR REPLACE option.

Recompiling Package Specifications

You might want to recompile a package specification to check for compilation errors after modifying the specification. When you issue an ALTER PACKAGE statement with the COMPILE PACKAGE option, Oracle recompiles the package specification and body regardless of whether it is invalid. When you recompile a package specification, Oracle invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. Note that the body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

Recompiling Package Bodies

You might want to recompile a package body after modifying it. When you issue an ALTER PACKAGE statement with the COMPILE BODY option, Oracle recompiles the package body regardless of whether it is invalid. When you recompile a package body, Oracle first recompiles the objects on which the body depends, if any of those objects are invalid. If Oracle recompiles the body successfully, the body becomes valid. If recompiling the body results in compilation errors, Oracle returns an error and the body remains invalid. You can then debug the body using the predefined package DBMS_OUTPUT. Note that recompiling a package body does not invalidate objects that depend upon the package specification.

For more information on debugging packages, see *Oracle8 Application Developer's Guide*. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8 Concepts*.

Example I This statement explicitly recompiles the specification and body of the ACCOUNTING package in the schema BLAIR:

```
ALTER PACKAGE blair.accounting
COMPILE PACKAGE;
```

If Oracle encounters no compilation errors while recompiling the ACCOUNTING specification and body, ACCOUNTING becomes valid. BLAIR can subsequently call or reference all package objects declared in the specification of ACCOUNTING without run-time recompilation. If recompiling ACCOUNTING results in compilation errors, Oracle returns an error message and ACCOUNTING remains invalid.

Oracle also invalidates all objects that depend upon ACCOUNTING. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

Example II To recompile the body of the ACCOUNTING package in the schema BLAIR, issue the following statement:

```
ALTER PACKAGE blair.accounting
COMPILE BODY;
```

If Oracle encounters no compilation errors while recompiling the package body, the body becomes valid. BLAIR can subsequently call or reference all package objects declared in the specification of ACCOUNTING without run-time recompilation. If recompiling the body results in compilation errors, Oracle returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of ACCOUNTING, Oracle does not invalidate dependent objects.

Related Topics

CREATE PACKAGE on page 4-250
CREATE PACKAGE BODY on page 4-254

ALTER PROCEDURE

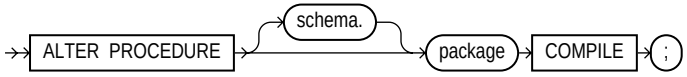
Purpose

To recompile a stand-alone stored procedure. See also “Recompiling Stored Procedures” on page 4-41.

Prerequisites

The procedure must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.
procedure	is the name of the procedure to be recompiled.
COMPILE	causes Oracle to recompile the procedure. The COMPILE keyword is required.

Recompiling Stored Procedures

The ALTER PROCEDURE command is quite similar to the ALTER FUNCTION command. The following discussion of explicitly recompiling procedures also applies to functions.

You can use the ALTER PROCEDURE command to explicitly recompile a procedure that is invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

When you issue an ALTER PROCEDURE statement, Oracle recompiles the procedure regardless of whether it is valid or invalid.

You can use the ALTER PROCEDURE command only to recompile a standalone procedure. To recompile a procedure that is part of a package, recompile the entire package using the ALTER PACKAGE command.

When you recompile a procedure, Oracle first recompiles objects upon which the procedure depends, if any of those objects are invalid. Oracle also invalidates any local objects that depend upon the procedure, such as procedures that call the recompiled procedure or package bodies that define procedures that call the recompiled procedure. If Oracle recompiles the procedure successfully, the procedure becomes valid. If recompiling the procedure results in compilation errors, then Oracle returns an error and the procedure remains invalid. You can then debug procedures using the predefined package DBMS_OUTPUT. For information on debugging procedures, see *Oracle8 Application Developer's Guide*. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8 Concepts*.

Note: This command does not change the declaration or definition of an existing procedure. To redeclare or redefine a procedure, use the CREATE PROCEDURE command with the OR REPLACE option.

Example To explicitly recompile the procedure CLOSE_ACCT owned by the user HENRY, issue the following statement:

```
ALTER PROCEDURE henry.close_acct
COMPILE;
```

If Oracle encounters no compilation errors while recompiling CLOSE_ACCT, CLOSE_ACCT becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling CLOSE_ACCT results in compilation errors, Oracle returns an error and CLOSE_ACCT remains invalid.

Oracle also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that call CLOSE_ACCT. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

Related Topics

ALTER FUNCTION on page 4-26
ALTER PACKAGE on page 4-38
CREATE PROCEDURE on page 4-259

ALTER PROFILE

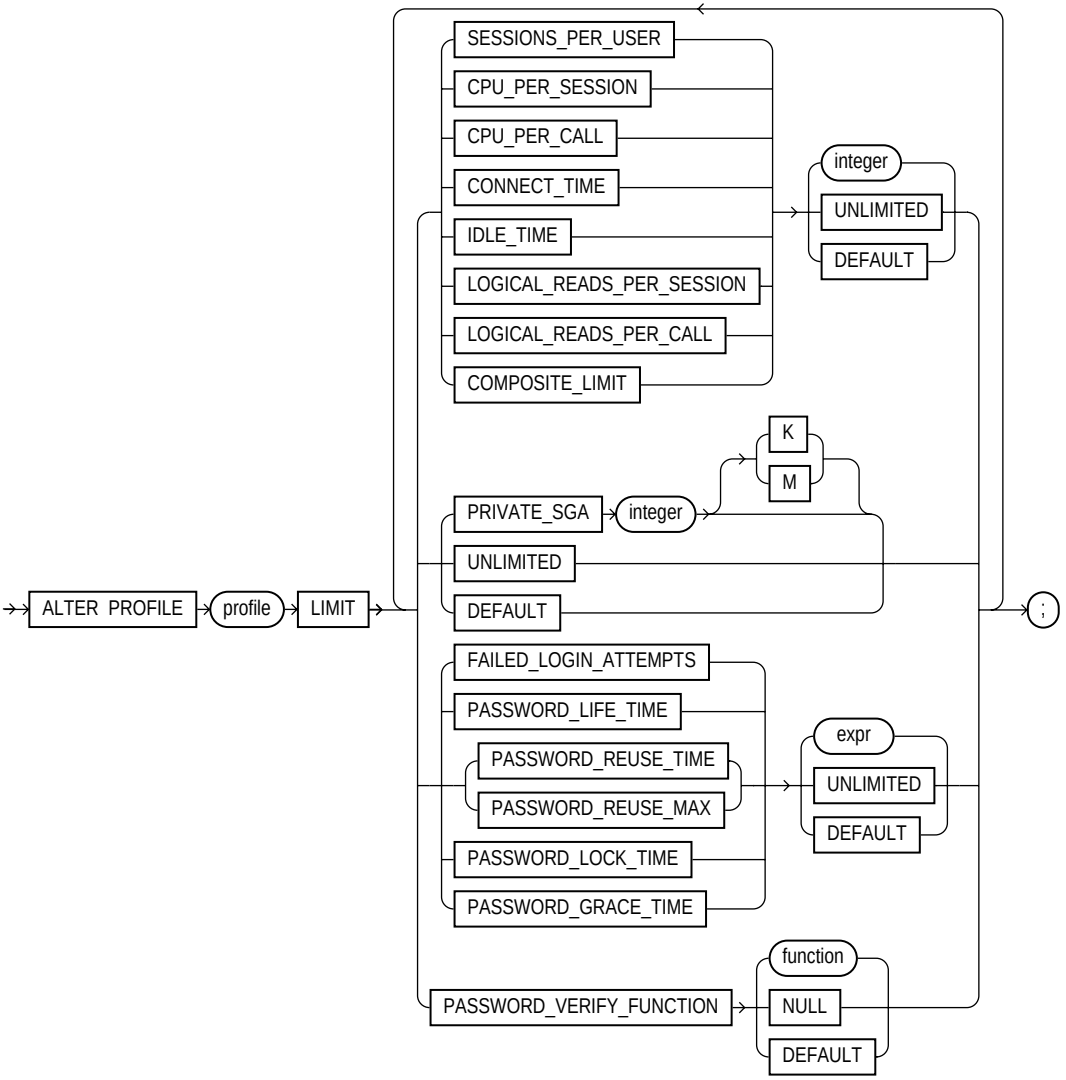
Purpose

To add, modify, or remove a resource limit or password management in a profile. See also “Examples” on page 4-45.

Prerequisites

You must have ALTER PROFILE system privilege to change profile resource limits. To modify password limits and protection, you must have ALTER PROFILE and ALTER USER system privileges. See also “Using Password History” on page 4-45.

Syntax



Keywords and Parameters

profile is the name of the profile to be altered.

integer defines a new limit for a resource in this profile.

For information on parameter resource limits for ALTER PROFILE, see CREATE PROFILE on page 4-265.

Note:

- You cannot remove a limit from the DEFAULT profile.
- You can use fractions of days for all parameters with days as units. Fractions are expressed as *x/y*. For example, 1 hour is 1/24 and 1 minute is 1/1440.

Using Password History

Changes made to a profile with an ALTER PROFILE statement affect users only in their subsequent sessions, not in their current sessions.

The following restrictions apply when specifying password history parameters:

- If PASSWORD_REUSE_TIME is set to an integer value, PASSWORD_REUSE_MAX must be set to UNLIMITED. If PASSWORD_REUSE_MAX is set to an integer value, PASSWORD_REUSE_TIME must be set to UNLIMITED.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to UNLIMITED, then Oracle uses neither of these password resources.
- If PASSWORD_REUSE_MAX is set to DEFAULT and PASSWORD_REUSE_TIME is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_MAX value defined in the default profile.
- If PASSWORD_REUSE_TIME is set to DEFAULT and PASSWORD_REUSE_MAX is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_TIME value defined in the default profile.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to DEFAULT, then Oracle uses whichever value is defined in the default profile.

Examples

Example I The following example makes a password unavailable for reuse for 90 days:

```
ALTER PROFILE prof
LIMIT PASSWORD_REUSE_TIME 90
PASSWORD_REUSE_MAX UNLIMITED;
```

Example II The following statement defaults the PASSWORD_REUSE_TIME value to its defined value in the DEFAULT profile:

```
ALTER PROFILE prof
```

```
LIMIT PASSWORD_REUSE_TIME DEFAULT
PASSWORD_REUSE_MAX UNLIMITED;
```

Example III The following example alters profile PROF with FAILED_LOGIN_ ATTEMPTS set to 5 and PASSWORD_LOCK_TIME set to 1:

```
ALTER PROFILE prof LIMIT
FAILED_LOGIN_ATTEMPTS 5
PASSWORD_LOCK_TIME 1;
```

This command causes PROF's account to become locked for 1 day after 5 unsuccessful login attempts.

Example IV The following example modifies profile PROF's PASSWORD_LIFE_ TIME to 60 days and PASSWORD_GRACE_TIME to 10 days:

```
ALTER PROFILE prof LIMIT
PASSWORD_LIFE_TIME 60
PASSWORD_GRACE_TIME 10;
```

Example V This statement defines a new limit of 5 concurrent sessions for the ENGINEER profile:

```
ALTER PROFILE engineer LIMIT SESSIONS_PER_USER  5;
```

If the ENGINEER profile does not currently define a limit for SESSIONS_PER_ USER, the above statement adds the limit of 5 to the profile. If the profile already defines a limit, the above statement redefines it to 5. Any user assigned the ENGINEER profile is subsequently limited to 5 concurrent sessions.

Example VI This statement defines unlimited idle time for the ENGINEER profile:

```
ALTER PROFILE engineer LIMIT IDLE_TIME UNLIMITED;
```

Any user assigned the ENGINEER profile is subsequently permitted unlimited idle time.

Example VII This statement removes the IDLE_TIME limit from the ENGINEER profile:

```
ALTER PROFILE engineer LIMIT IDLE_TIME DEFAULT;
```

Any user assigned the ENGINEER profile is subject in their subsequent sessions to the IDLE_TIME limit defined in the DEFAULT profile.

Example VIII 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

Related Topics

CREATE PROFILE on page 4-265

ALTER RESOURCE COST

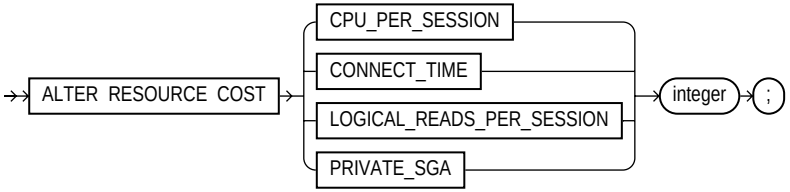
Purpose

To specify a formula to calculate the total resource cost used in a session. For any session, this cost is limited by the value of the COMPOSITE_LIMIT parameter in the user's profile. See also "Altering Resource Costs" on page 4-48.

Prerequisites

You must have ALTER RESOURCE COST system privilege.

Syntax



Keywords and Parameters

CPU_PER_SESSION	is the amount of CPU time used by a session measured in hundredth of seconds.
CONNECT_TIME	is the amount of CPU time used by a session measured in hundredths of seconds.
CPU_PER_SESSION	is the elapsed time of a session measured in minutes.
LOGICAL_READS_PER_ SESSION	is the number of data blocks read during a session including blocks read from both memory and disk.
PRIVATE_SGA	The number of bytes of private space in the system global area (SGA) used by a session. This limit only applies if you are using the multithreaded server architecture and allocating private space in the SGA for your session.
integer	is the weight of each resource.

Altering Resource Costs

The ALTER RESOURCE COST command specifies the formula by which Oracle calculates the total resource cost used in a session. Oracle calculates the total resource cost by multiplying the amount of each resource used in the session by the

resource's weight and summing the products for all four resources. Both the products and the total cost are expressed in units called *service units*.

Although Oracle monitors the use of other resources, only these four can contribute to the total resource cost for a session. For information on all resources, see CREATE PROFILE on page 4-265.

The weight that you assign to each resource determines how much the use of that resource contributes to the total resource cost. Using a resource with a lower weight contributes less to the cost than using a resource with a higher weight. If you do not assign a weight to a resource, 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.

Once you have specified a formula for the total resource cost, you can limit this cost for a session with the COMPOSITE_LIMIT parameter of the CREATE PROFILE command. If a session's cost exceeds the limit, Oracle aborts the session and returns an error. For information on establishing resource limits, see CREATE PROFILE on page 4-265. If you use the ALTER RESOURCE COST command to change the weight assigned to each resource, Oracle uses these new weights to calculate the total resource cost for all current and subsequent sessions.

Example I 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:

$T = (100 * CPU) + CON$

where:

T	is the total resource cost for the session expressed in service units.
CPU	is the CPU time used by the session measured in hundredths of seconds.
CON	is the elapsed time of a session measured in minutes.

Because the above 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, a session exceeds this limit whenever *T* exceeds 500. For example, a session using 0.04 seconds of CPU time and 101 minutes of elapsed time exceeds the limit. A session 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:

$T = (100 * CPU) + (2 * LOG)$

where:

T CPU	are the same as in the previous formula.
LOG	is the number of data blocks read during the session.

This ALTER RESOURCE COST statement changes the formula in these ways:

- Because the statement omits a weight for the CPU_PER_SESSION resource and the resource was already assigned a weight, the resource remains in the formula with its original weight.
- Because the statement assigns a weight to the LOGICAL_READS_PER_ SESSION resource, this resource now appears in the formula.
- Because the statement assigns a weight of 0 to the CONNECT_TIME resource, this resource no longer appears in the formula.
- Because the statement omits a weight for the PRIVATE_SGA resource and the resource was not already assigned a weight, the resource still does not appear in the formula.

Related Topics

CREATE PROFILE on page 4-265

ALTER ROLE

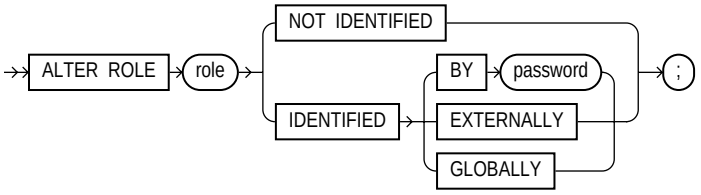
Purpose

To change the authorization needed to enable a role. See also “Changing Authorizations” on page 4-51.

Prerequisites

You must either have been granted the role with the ADMIN OPTION or have ALTER ANY ROLE system privilege.

Syntax



Keywords and Parameters

The keywords and parameters in the ALTER ROLE command all have the same meaning as in the CREATE ROLE command; see CREATE ROLE on page 4-272.

Changing Authorizations

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.

If a user with ALTER ANY ROLE changes a role that is IDENTIFIED GLOBALLY to any of the following, then Oracle grants the role with the ADMIN OPTION:

- to IDENTIFIED BY *password*
- to IDENTIFIED EXTERNALLY
- to NOT IDENTIFIED

Example I The following example changes the role ANALYST to IDENTIFIED GLOBALLY:

```
ALTER ROLE analyst IDENTIFIED GLOBALLY;
```

Example II This statement changes the password on the TELLER role to LETTER:

```
ALTER ROLE teller
IDENTIFIED BY letter;
```

Users granted the TELLER role must subsequently enter the new password “letter” to enable the role.

Related Topics

CREATE ROLE on page 4-272
SET ROLE on page 4-516

ALTER ROLLBACK SEGMENT

Purpose

To alter a rollback segment by

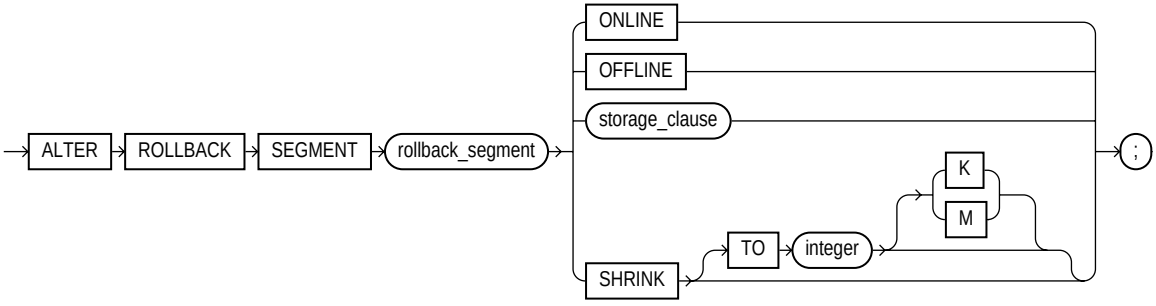
- bringing it online
- taking it offline
- changing its storage characteristics
- shrinking it to an optimal or given size

For more information, see "Altering Rollback Segments" on page 4-54.

Prerequisites

You must have ALTER ROLLBACK SEGMENT system privilege.

Syntax



storage_clause: See STORAGE clause on page 4-523.

Keywords and Parameters

rollback_segment	specifies the name of an existing rollback segment.
ONLINE	brings the rollback segment online.
OFFLINE	takes the rollback segment offline.
storage_clause	changes the rollback segment's storage characteristics. See the STORAGE clause on page 4-523 for syntax and additional information.
SHRINK	attempts to shrink the rollback segment to an optimal or given size.

Altering Rollback Segments

When you create a rollback segment, it is initially offline. An offline rollback segment is not available for transactions.

The ONLINE option 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.

The OFFLINE option takes the rollback segment offline. If the rollback segment does not contain information necessary to roll back any active transactions, Oracle takes it offline immediately. If the rollback segment does contain information for active transactions, Oracle makes the rollback segment unavailable for future transactions and takes it offline after all the active transactions are committed or rolled back. Once the rollback segment is offline, it can be brought online by any instance.

You cannot take the SYSTEM rollback segment offline.

You can tell whether a rollback segment is online or offline by querying the data dictionary view DBA_ROLLBACK_SEGS. Online rollback segments are indicated by a STATUS value of IN_USE. Offline rollback segments are indicated by a STATUS value of AVAILABLE.

For more information on making rollback segments available and unavailable, see *Oracle8 Administrator's Guide*.

The STORAGE clause of the ALTER ROLLBACK SEGMENT command affects future space allocation in the rollback segment. You cannot change the values of the INITIAL and MINEXTENTS for an existing rollback segment.

The SHRINK clause of the ALTER ROLLBACK SEGMENT command initiates an attempt to reduce the specified rollback segment to an optimum size. If size is not specified, then the size defaults to the OPTIMAL value of the STORAGE clause of the CREATE ROLLBACK SEGMENT command that created the rollback segment.

If the OPTIMAL value was not specified, then the size defaults to the MINEXTENTS value of the STORAGE clause of the CREATE ROLLBACK SEGMENT command. The specified size in a SHRINK clause is valid for the execution of the command; thereafter, OPTIMAL reverts to the OPTIMAL value of the CREATE ROLLBACK SEGMENT command. Regardless of whether a size is specified or not, the rollback segment cannot shrink to less than two extents.

You can query the DBA_ROLLBACK_SEGS view to determine the actual size of a rollback segment after attempting to shrink a rollback segment.

For a parallel server, you can shrink only rollback segments that are online to your instance.

The SHRINK option is an *attempt* to shrink the size of the rollback segment; the success and amount of shrinkage depends on the following:

- available free space in the rollback segment
- how active transactions are holding space in the rollback segment

Example I This statement brings the rollback segment RSONE online:

```
ALTER ROLLBACK SEGMENT rsone ONLINE;
```

Example II This statement changes the STORAGE parameters for RSONE:

```
ALTER ROLLBACK SEGMENT rsone
STORAGE (NEXT 1000 MAXEXTENTS 20);
```

Example III This statement attempts to resize a rollback segment to an optimum size of 100 megabytes:

```
ALTER ROLLBACK SEGMENT rsone
SHRINK TO 100 M;
```

Related Topics

CREATE ROLLBACK SEGMENT on page 4-275
CREATE TABLESPACE on page 4-328
STORAGE clause on page 4-523

ALTER SEQUENCE

Purpose

To change the sequence by

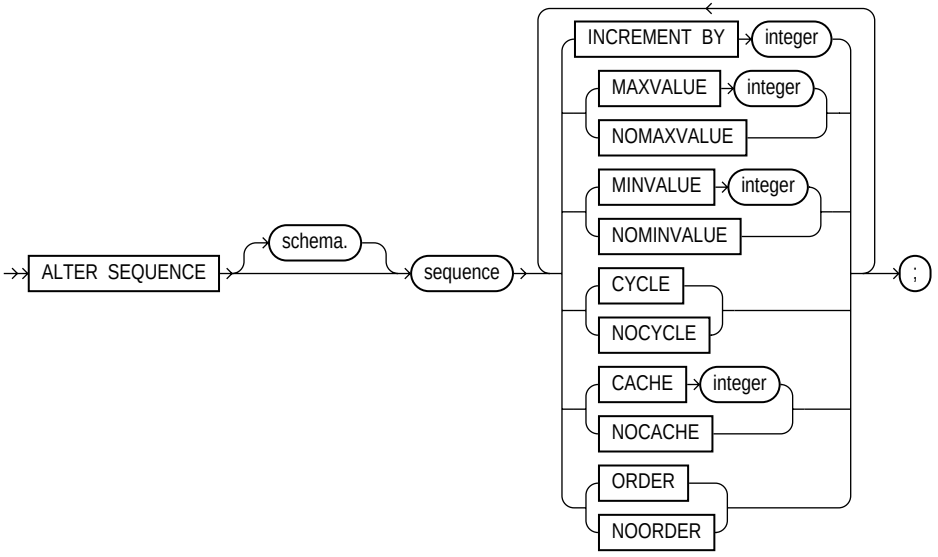
- changing the increment between future sequence values
- setting or eliminating the minimum or maximum value
- changing the number of cached sequence numbers
- specifying whether the sequence continues to generate numbers after reaching the maximum or minimum value
- specifying whether sequence numbers must be ordered

For illustrations of some of these purposes, see "Examples" on page 4-57.

Prerequisites

The sequence must be in your own schema or you must have ALTER privilege on the sequence or you must have ALTER ANY SEQUENCE system privilege.

Syntax



Keywords and Parameters

The keywords and parameters in this command serve the same purpose that they do in CREATE SEQUENCE on page 4-281.

Note:

- The sequence must be dropped and re-created to restart the sequence at a different number. Only future sequence numbers are affected by the ALTER SEQUENCE command.
- Some validations are performed. For example, a new MAXVALUE cannot be imposed that is less than the current sequence number.

Examples

Example I This statement sets a new maximum value for the ESEQ sequence:

```
ALTER SEQUENCE eseq
MAXVALUE 1500;
```

Example II This statement turns on CYCLE and CACHE for the ESEQ sequence:

```
ALTER SEQUENCE eseq
CYCLE
CACHE 5;
```

Related Topics

CREATE SEQUENCE on page 4-281
DROP SEQUENCE on page 4-401

ALTER SESSION

Purpose

To alter your current session in one of the following ways:

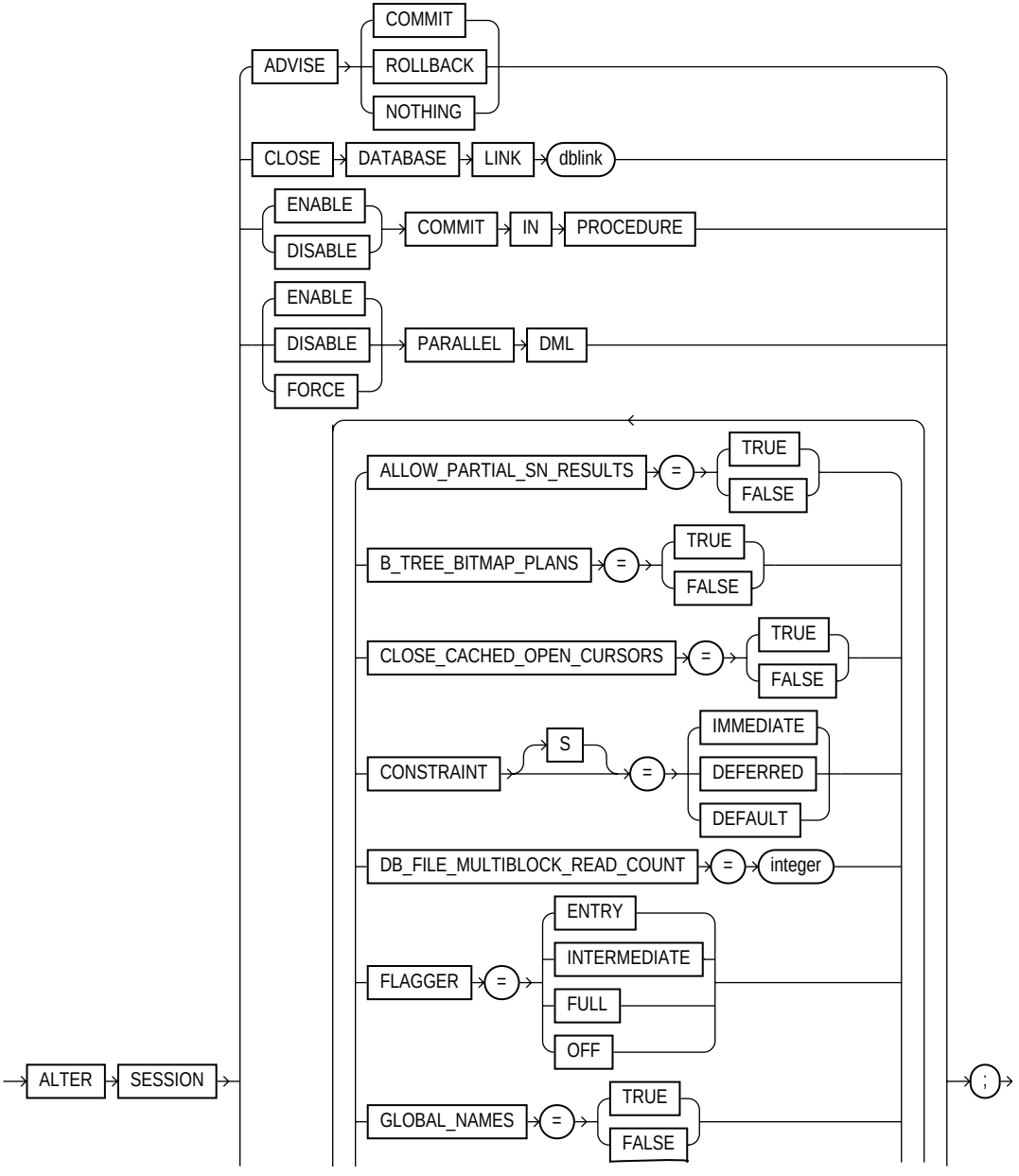
- enable or disable the SQL trace facility
- enable or disable global name resolution
- change the values of NLS parameters
- specify the size of the cache used to hold frequently used cursors
- enable or disable the closing of cached cursors on COMMIT or ROLLBACK
- in a parallel server, indicate that the session must access database files as if the session were connected to another instance
- enable, disable, and change the behavior of hash join operations
- change the handling of remote procedure call dependencies
- change transaction level handling
- close a database link
- send advice to remote databases to force an in-doubt distributed transaction
- permit or prohibit stored procedures and functions from issuing COMMIT and ROLLBACK statements
- change the goal of the cost-based optimization approach
- in a parallel server, enable DML statements to be considered for parallel execution
- to insert, update, or delete from tables with indexes or index partitions marked as unusable
- allow deferrable constraints to be checked either immediately following every DML statement or at the end of a transaction

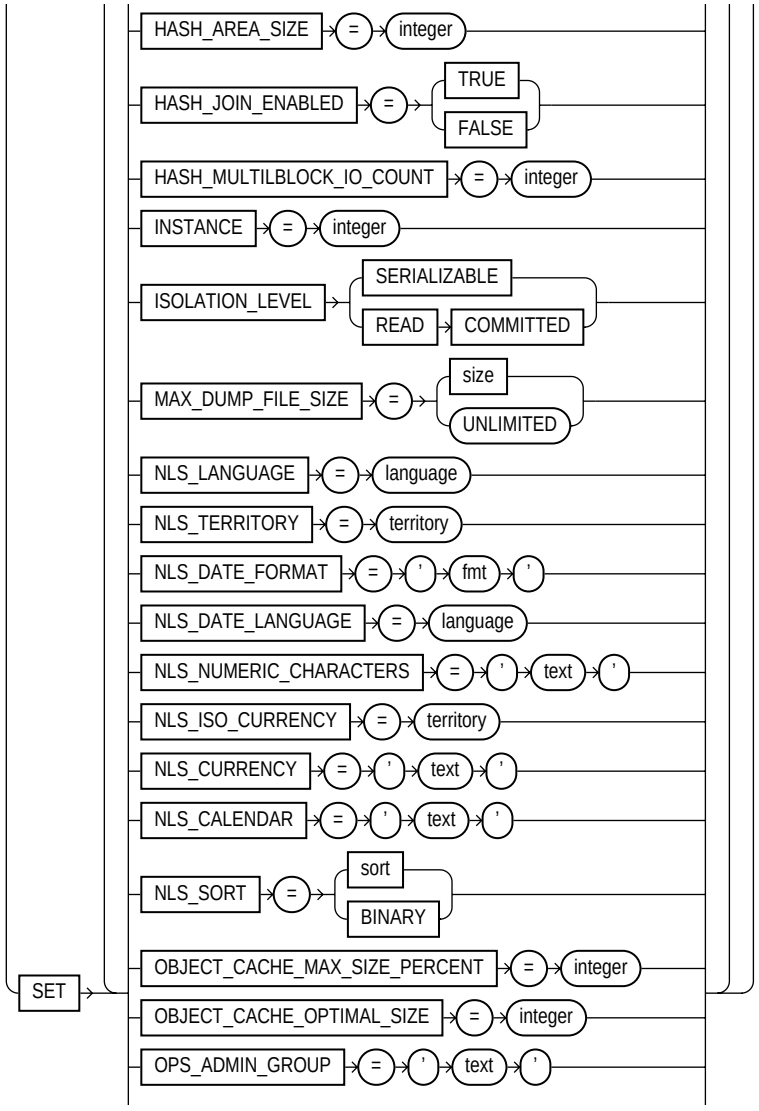
Prerequisites

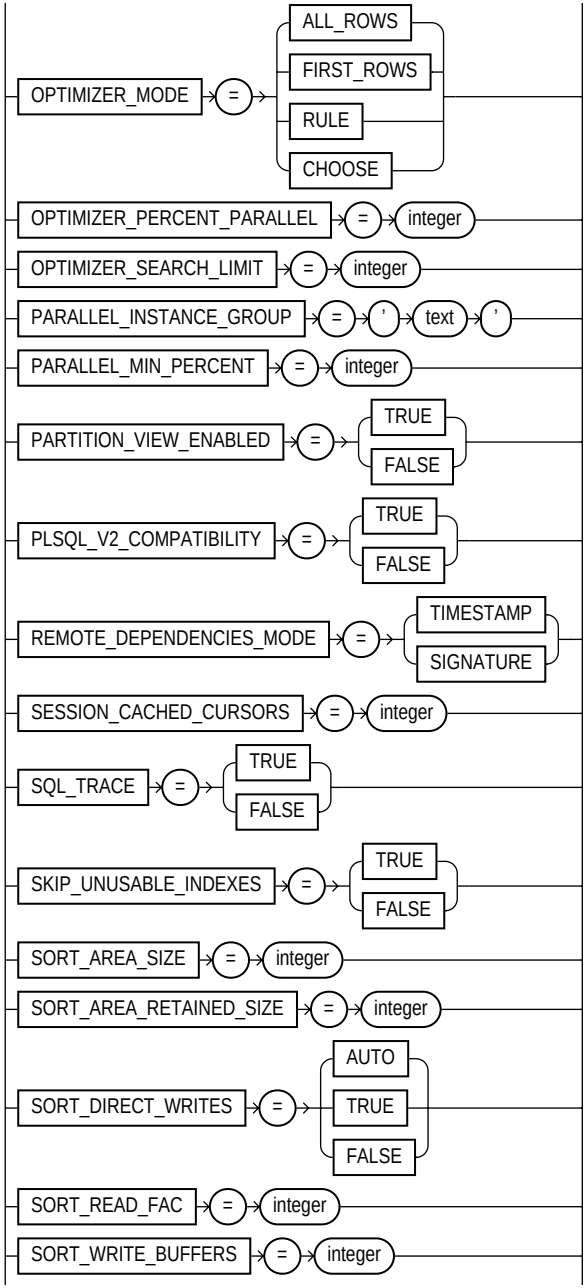
To enable and disable the SQL trace facility or to change the default label format, you must have ALTER SESSION system privilege.

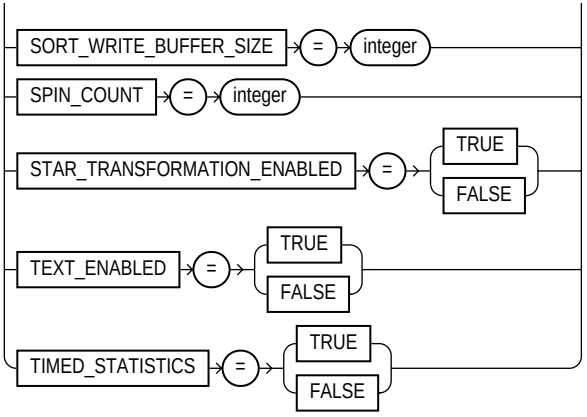
To perform the other operations of this command, you do not need any privileges.

Syntax


ALTER
SESSION
ADVISE
COMMIT
ROLLBACK
NOTHING
CLOSE
DATABASE
LINK
dblink
ENABLE
DISABLE
COMMIT
IN
PROCEDURE
ENABLE
DISABLE
FORCE
PARALLEL
DML
ALLOW_PARTIAL_SN_RESULTS
=
TRUE
FALSE
B_TREE_BITMAP_PLANS
=
TRUE
FALSE
CLOSE_CACHED_OPEN_CURSORS
=
TRUE
FALSE
CONSTRAINT
S
=
IMMEDIATE
DEFERRED
DEFAULT
DB_FILE_MULTIBLOCK_READ_COUNT
=
integer
FLAGGER
=
ENTRY
INTERMEDIATE
FULL
OFF
GLOBAL_NAMES
=
TRUE
FALSE
;


SET
HASH_AREA_SIZE
=
integer
HASH_JOIN_ENABLED
=
TRUE
FALSE
HASH_MULTILBLOCK_IO_COUNT
=
integer
INSTANCE
=
integer
ISOLATION_LEVEL
SERIALIZABLE
READ
COMMITTED
MAX_DUMP_FILE_SIZE
=
size
UNLIMITED
NLS_LANGUAGE
=
language
NLS_TERRITORY
=
territory
NLS_DATE_FORMAT
=
'
fmt
'
NLS_DATE_LANGUAGE
=
language
NLS_NUMERIC_CHARACTERS
=
'
text
'
NLS_ISO_CURRENCY
=
territory
NLS_CURRENCY
=
'
text
'
NLS_CALENDAR
=
'
text
'
NLS_SORT
=
sort
BINARY
OBJECT_CACHE_MAX_SIZE_PERCENT
=
integer
OBJECT_CACHE_OPTIMAL_SIZE
=
integer
OPS_ADMIN_GROUP
=
'
text
'


OPTIMIZER_MODE
=
ALL_ROWS
FIRST_ROWS
RULE
CHOOSE
OPTIMIZER_PERCENT_PARALLEL
=
integer
OPTIMIZER_SEARCH_LIMIT
=
integer
PARALLEL_INSTANCE_GROUP
=
'
text
'
PARALLEL_MIN_PERCENT
=
integer
PARTITION_VIEW_ENABLED
=
TRUE
FALSE
PLSQL_V2_COMPATIBILITY
=
TRUE
FALSE
REMOTE_DEPENDENCIES_MODE
=
TIMESTAMP
SIGNATURE
SESSION_CACHED_CURSORS
=
integer
SQL_TRACE
=
TRUE
FALSE
SKIP_UNUSABLE_INDEXES
=
TRUE
FALSE
SORT_AREA_SIZE
=
integer
SORT_AREA_RETAINED_SIZE
=
integer
SORT_DIRECT_WRITES
=
AUTO
TRUE
FALSE
SORT_READ_FAC
=
integer
SORT_WRITE_BUFFERS
=
integer



Keywords and Parameters

ADVISE — sends advice to a remote database to force a distributed transaction. This advice appears on the remote database in the ADVICE column of the DBA_2PC_PENDING data dictionary view in the event of an in-doubt distributed transaction. (See also "Forcing In-Doubt Distributed Transactions" on page 4-73.)The following are advice options:

- COMMIT — places the value 'C' in DBA_2PC_PENDING.ADVICE.
- ROLLBACK — places the value 'R' in DBA_2PC_PENDING.ADVICE.
- NOTHING — places the value ' ' in DBA_2PC_PENDING.ADVICE.

CLOSE DATABASE LINK — closes the database link *dblink*, eliminating your session's connection to the remote database. The database link cannot be currently in use by an active transaction or an open cursor. For more information, see "Closing Database Links" on page 4-72.

COMMIT IN PROCEDURE

- ENABLE — permits procedures and stored functions to issue these statements.
- DISABLE — prohibits procedures and stored functions from issuing these statements.

See also "Transaction Control in Procedures and Stored Functions" on page 4-74.

PARALLEL DML — specifies whether all subsequent DML transactions in the session will be considered for parallel execution. (See also "Parallel DML" on page 4-74.)

You can execute this option only between committed transactions. Uncommitted transactions must either be committed or rolled back prior to executing this command.

	ENABLE	executes the session's DML statements in parallel mode if a parallel hint or a parallel clause is specified.
	DISABLE	executes the session's DML statements serially. This is the default mode.
	FORCE	forces parallel execution of subsequent DML statements in the session if none of the parallel DML restrictions are violated. If no parallel clause or hint is specified, then a default level of parallelism (for both degree and instances) is used. **Note**: Using FORCE 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 default degree and default instances) with the CREATE TABLE statement.
SET	sets the session parameters that follow.	
CLOSE_OPEN_ CACHED_ CURSORS	controls whether cursors opened and cached in memory by PL/SQL are automatically closed at each COMMIT or ROLLBACK.	
	TRUE	causes open cursors to be closed at each COMMIT or ROLLBACK.
	FALSE	signifies that cursors opened by PL/SQL are held open so that subsequent executions need not open a new cursor.
CONSTRAINT[S]	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; equivalent to issuing the SET CONSTRAINTS ALL IMMEDIATE command at the beginning of each transaction in your session. See the IMMEDIATE parameter of SET CONSTRAINT(S) on page 4-514.
	DEFERRED	indicates that the conditions specified by the deferrable constraint are checked when the transaction is committed; equivalent to issuing the SET CONSTRAINTS ALL DEFERRED command at the beginning of each transaction in your session. See the DEFERRED parameter of SET CONSTRAINT(S) on page 4-514.
	DEFAULT	restores all constraints at the beginning of each transaction to their initial state of DEFERRED or IMMEDIATE.
FLAGGER	specifies FIPS flagging. See also "FIPS Flagging" on page 4-71.	
	ENTRY	flags for SQL92 Entry level.
	INTERMEDIATE	flags for SQL92 Intermediate level.
	FULL	flags for SQL92 Full level.

	OFF	turns off flagging
GLOBAL_ NAMES	controls the enforcement of global name resolution for your session. For information on enabling and disabling global name resolution with this parameter, see ALTER SYSTEM on page 4-88.	
	TRUE	enables global name resolution.
	FALSE	disables global name resolution.
HASH_JOIN_ ENABLED	enables or disables the use of the hash join operation in queries. The default is TRUE, which enables hash joins.	
HASH_AREA_ SIZE	specifies in bytes the amount of memory to use for hash join operations. The default is twice the value of the SORT_AREA_SIZE initialization parameter.	
HASH_ MULTIBLOCK_ IO_COUNT	specifies the number of data blocks to read and write during a hash join operation. The value multiplied by the DB_BLOCK_SIZE initialization parameter should not exceed 64 K. The default value for this parameter is 1. If the multithreaded server is used, the value is always 1, and any value given here is ignored.	
INSTANCE	in a parallel server, accesses database files as if the session were connected to the instance specified by *integer*. For more information, see "Accessing the Database as if Connected to Another Instance in a Parallel Server" on page 4-72.	
ISOLATION_ LEVEL	specifies how transactions containing database modifications are handled.	
	SERIALIZABLE	Transactions in the session use the serializable transaction isolation mode as specified in SQL92. That is, if a serializable transaction attempts to execute a DML statement that updates rows that are 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. The COMPATIBLE initialization parameter must be set to 7.3.0 or higher for SERIALIZABLE mode to work.
	READ COMMITTED	Transactions in the session will use the default Oracle transaction behavior. Thus, 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.
MAX_DUMP_ FILE_SIZE	specifies the upper limit of trace dump file size. Specify the maximum *size* as either a nonnegative integer that represents the number of blocks, or as 'UNLIMITED'. If 'UNLIMITED' is specified, no upper limit is imposed.	

For more information on the following NLS parameters, see "Using NLS Parameters" on page 4-67.

NLS_ LANGUAGE	changes the language in which Oracle returns errors and other messages. This parameter also implicitly specifies new values for these items:

	■ language for day and month names and abbreviations and spelled values of other elements ■ sort sequences ■ B.C. and A.D. indicators ■ A.M. and P.M. meridian indicators
NLS_ TERRITORY	implicitly specifies new values for these items: ■ default date format ■ decimal character and group separators ■ local currency symbol ■ ISO currency symbol ■ first day of the week for D date format element
NLS_DATE_ FORMAT	explicitly specifies a new default date format. The '*fmt*' value must be a date format model as specified in the section "Date Format Models" on page 3-69.
NLS_DATE_ LANGUAGE	explicitly changes the language for day and month names and abbreviations and spelled values of other date format elements.
NLS_NUMERIC_ CHARACTERS	explicitly specifies a new decimal character and group separator. The '*text*' value must have this form: `dg'` where: *d* is the new decimal character, and *g* is the new group separator. The decimal character and the group separator must be two different single-byte characters, and cannot be a numeric value or any of the following characters: "+" plus, "–" minus (or hyphen), "<" less-than, or ">" greater-than.
NLS_ISO_ CURRENCY	explicitly specifies the territory whose ISO currency symbol should be used.
NLS_ CURRENCY	explicitly specifies a new local currency symbol. The symbol cannot exceed 10 characters.
NLS_SORT	changes the sequence into which Oracle sorts character values. *sort* specifies the name of a linguistic sort sequence. BINARY specifies a binary sort. The default sort for all character sets is binary.
NLS_ CALENDAR	explicitly specifies a new calendar type.

OPTIMIZER_ MODE	specifies the approach and mode of the optimizer for your session. For more information on optimizer mode, see "Changing the Optimization Approach and Mode" on page 4-71.	
	ALL_ROWS	specifies the cost-based approach and optimizes for best throughput.
	FIRST_ROWS	specifies the cost-based approach and optimizes for best response time.
	RULE	specifies the rule-based approach.
	CHOOSE	causes the optimizer to choose an optimization approach based on the presence of statistics in the data dictionary.
PARTITION_ VIEW_ENABLED	When set to TRUE, this parameter causes the optimizer to skip unnecessary table accesses in a partition view. For more information, see *Oracle8 Reference*.	
PLSQL_V2_ COMPATABILITY	modifies the compile-time behavior of PL/SQL programs to allow language constructs that are illegal in Oracle8 (PL/SQL V3), but were legal in Oracle7 (PL/SQL V2). See the *PL/SQL User's Guide and Reference* and *Oracle8 Reference* for more information about this session parameter.	
	TRUE	enables Oracle8 PL/SQL V3 programs to execute Oracle7 PL/SQL V2 constructs.
	FALSE	disallows illegal Oracle7 PL/SQL V2 constructs. This is the default.
REMOTE_ DEPENDENCIES _MODE	specifies how dependencies of remote stored procedures are handled by the session. For more information, refer *Oracle8 Application Developer's Guide*.	
SESSION_ CACHED_ CURSORS	specifies the size of the session cache for holding frequently used cursors. *integer* specifies how many cursors can be retained in the cache. For more information on this parameter, see "Caching Session Cursors" on page 4-71.	
SKIP_UNUSABLE_INDEXES		
	controls the use and reporting of tables with unusable indexes or index partitions.	
	TRUE	disables error reporting of indexes marked as unusable. Allows inserts, deletes, and updates to tables with unusable indexes or index partitions.
	FALSE	enables error reporting of indexes marked as unusable. Does not allow inserts, deletes, and updates to tables with unusable indexes or index partitions. This is the default.
SQL_TRACE	controls the SQL trace facility for your session. See also "Enabling and Disabling the SQL Trace Facility" on page 4-67.	
	TRUE	enables the SQL trace facility.
	FALSE	disables the SQL trace facility.

Enabling and Disabling the SQL Trace Facility

The SQL trace facility generates performance statistics for the processing of SQL statements. You can enable and disable the SQL trace facility for all sessions on an Oracle instance with the initialization parameter SQL_TRACE. When you begin a session, Oracle enables or disables the SQL trace facility based on the value of this parameter. You can subsequently enable or disable the SQL trace facility for your own session with the SQL_TRACE option of the ALTER SESSION command.

For more information on the SQL trace facility, including how to format and interpret its output, see *Oracle8 Tuning*.

Example I. To enable the SQL trace facility for your session, issue the following statement:

```
ALTER SESSION
SET SQL_TRACE = TRUE;
```

Using NLS Parameters

Oracle contains support for use in different nations and with different languages. When you start an instance, Oracle establishes support based on the values of initialization parameters that begin with "NLS". For information on these parameters, see *Oracle8 Reference*. You use the NLS clauses of the ALTER SESSION command to change NLS characteristics dynamically for your session. You can query the dynamic performance table V$NLS_PARAMETERS to see the current NLS attributes for your session. The sections that follow describe the use of specific NLS parameters.

Language for Error Messages

You can specify a new language for error messages with the NLS_LANGUAGE parameter. Note that this parameter also implicitly changes other language-related items. Oracle provides error messages in a wide range of languages on many platforms.

Example II. The following statement changes the language for error messages to the French:

```
ALTER SESSION
  SET NLS_LANGUAGE = French
```

Oracle returns error messages in French:

```
SELECT * FROM emp
```

```
ORA-00942: Table ou vue n'existe pas
```

> **Note:** The language you select must already have been installed. Refer to your operating system specific installation instructions.

Default Date Format

You can specify a new default date format either explicitly with the NLS_DATE_FORMAT parameter or implicitly with the NLS_TERRITORY parameter. For information on the default date format models, see the section "Date Format Models" on page 3-69.

Example III. 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 uses the new default date format:

```
SELECT TO_CHAR(SYSDATE) Today
FROM DUAL
TODAY
-------------------
1997 08 12 14:25:56
```

Language for Months and Days

You can specify a new language for names and abbreviations of months and days either explicitly with the NLS_DATE_LANGUAGE parameter or implicitly with the NLS_LANGUAGE parameter.

Example IV. The following statement changes the language for date format elements to the French:

```
ALTER SESSION
SET NLS_DATE_LANGUAGE = French

SELECT TO_CHAR(SYSDATE, 'Day DD Month YYYY') Today
FROM DUAL

TODAY
---------------------------
Mardi    28 Février   1997
```

Decimal Character and Group Separator

You can specify new values for these number format elements either explicitly with the NLS_NUMERIC_CHARACTERS parameter or implicitly with the NLS_TERRITORY parameter:

D (decimal character)	is the character that separates the integer and decimal portions of a number.
G (group separator)	is the character that separates groups of digits in the integer portion of a number.

For information on how to use number format models, see "Number Format Models" on page 3-65.

The decimal character and the group separator must be single-byte character and cannot be the same character. If the decimal character is not a period (.), you must use single quotation marks to enclose to enclose all number values that appear in expressions in your SQL statements. When not using a period for the decimal point, you should always use the TO_NUMBER function to ensure that a valid number is retrieved.

Example V. The following statement dynamically changes the decimal character to comma (,) and the group separator to period (.):

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

Oracle returns these new characters when you use their number format elements:

```
SELECT TO_CHAR( SUM(sal), 'L999G999D99') Total FROM emp ;

TOTAL
-------------
FF29.025,00
```

ISO Currency Symbol

You can specify a new value for the C number format element (the ISO currency symbol) either explicitly with the NLS_ISO_CURRENCY parameter or implicitly with the NLS_TERRITORY parameter. The value that you specify for these parameters is a territory whose ISO currency symbol becomes the value of the C number format element.

Example VI. 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(sal), 'L999G999D99') Total
FROM emp;

TOTAL
-------------
USD29,025.00
```

Local Currency Symbol

You can specify a new value for the L number format element, (the local currency symbol) either explicitly with the NLS_CURRENCY parameter or implicitly with the NLS_TERRITORY parameter.

Example VII. The following statement dynamically changes the local currency symbol to 'DM':

```
ALTER SESSION
SET NLS_CURRENCY = 'DM';

SELECT TO_CHAR( SUM(sal), 'L999G999D99') Total
FROM emp;

TOTAL
-------------
DM29.025,00
```

Linguistic Sort Sequence

You can specify a new linguistic sort sequence or a binary sort either explicitly with the NLS_SORT parameter or implicitly with the NLS_LANGUAGE parameter.

Example VIII. The following statement dynamically changes the linguistic sort sequence to Spanish:

```
ALTER SESSION
SET NLS_SORT = XSpanish;
```

Oracle sorts character values based on their position in the Spanish linguistic sort sequence.

Changing the Optimization Approach and Mode

The Oracle optimizer can use either of these approaches to optimize a SQL statement:

cost-based The optimizer optimizes a SQL statement by considering statistics describing the tables, indexes, and clusters accessed by the statement as well as the information considered with the rule-based approach.

rule-based The optimizer optimizes a SQL statement based on the indexes and clusters associated with the accessed tables, the syntactic constructs of the statement, and a heuristically ranked list of these constructs.

With the cost-based approach, the optimizer can optimize a SQL statement with one of these goals:

best throughput is the minimal time necessary to return all rows accessed by the statement.

best response time is the minimal time necessary to return the first row accessed by the statement.

When you start your instance, the optimization approach is established by the initialization parameter OPTIMIZER_MODE. If this parameter establishes the cost-based approach, the default goal is best throughput.

For information on how to choose a goal for the cost-based approach based on the characteristics of your application, see the *Oracle8 Tuning*.

FIPS Flagging

FIPS flagging causes an error message to be generated when a SQL statement is issued that is an extension of ANSI SQL92. In Oracle, there is currently no difference between Entry, Intermediate, or Full level flagging. Once flagging is set in a session, a subsequent ALTER SESSION SET FLAGGER command will work, but generates the message, ORA-00097. This allows FIPS flagging to be altered without disconnecting the session.

Caching Session Cursors

If an application repeatedly issues parse calls on the same set of SQL statements, the reopening of the session cursors can affect performance. The ALTER SESSION SET SESSION_CACHED_CURSORS command allows frequently used session

cursors to be stored in a session cache even if they are closed. This is particularly useful for some Oracle tools. For example, Oracle Forms applications close all session cursors associated with a form when switching to another form; in this case, frequently used cursors would not have to be reparsed.

Oracle uses the shared SQL area to determine whether more than three parse requests were issued on a given statement. If so, Oracle moves the cursor into the session cursor cache. Subsequent requests to parse that SQL statement by the same session will find the cursor in the session cursor cache.

Session cursors are cached automatically if the initialization parameter SESSION_CACHED_CURSORS is set to a positive value. This parameter specifies the maximum number of session cursors to be kept in the cache. A least recently used algorithm ages out entries in the cache to make room for new entries when needed. You use the ALTER SESSION SET SESSION_CACHED_CURSORS command to dynamically enable session cursor caching.

For more information on session cursor caching, see *Oracle8 Tuning*.

Accessing the Database as if Connected to Another Instance in a Parallel Server

For optimum performance, each instance of a parallel server uses its own private rollback segments, freelist groups, and so on. A database is usually designed for a parallel server so that users connect to a particular instance and access data that is partitioned primarily for their use. If the users for that instance must connect to another instance, the data partitioning can be lost. The ALTER SESSION SET INSTANCE command allows users to access an instance as if they were connected to their usual instance.

Closing Database Links

A database link allows you to access a remote database in DELETE, INSERT, LOCK TABLE, SELECT, and UPDATE statements. When you issue a statement that uses a database link, Oracle creates a session for you on the remote database using the database 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.

You can use the CLOSE DATABASE LINK clause of the ALTER SESSION command to close a database link explicitly if you do not plan to use it again in your session. You may want to close a database link explicitly if the network overhead associated with leaving it open is costly. Before closing a database link, you must first close all cursors that use the link and then end your current transaction if it uses the link.

Example. This example updates the employee table on the SALES database using a database link, commits the transaction, and explicitly closes the database link:

```
UPDATE emp@sales
SET sal = sal + 200
WHERE empno = 9001;
COMMIT;
ALTER SESSION
CLOSE DATABASE LINK sales;
```

Forcing In-Doubt Distributed Transactions

If a network or machine failure occurs during the commit process for a distributed transaction, the state of the transaction may be unknown or in doubt. The transaction can be manually committed or rolled back on each database involved in the transaction with the FORCE clause of the COMMIT or ROLLBACK commands.

Before committing a distributed transaction, you can use the ADVISE clause of the ALTER SESSION command to send advice to a remote database in the event a distributed transaction becomes in doubt. If the transaction becomes in doubt, the advice appears in the ADVICE column of the DBA_2PC_PENDING view on the remote database. The administrator of that database can then use this advice to decide whether to commit or roll back the transaction on the remote database. For more information on distributed transactions and how to decide whether to commit or roll back in-doubt distributed transactions, see *Oracle8 Distributed Database Systems*.

You issue 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. This allows you to send different advice to different databases.

Example. This transaction inserts an employee record into the EMP table on the database identified by the database link SITE1 and deletes an employee record from the EMP table on the database identified by SITE2:

```
ALTER SESSION
ADVISE COMMIT

INSERT INTO emp@site1
VALUES (8002, 'FERNANDEZ', 'ANALYST', 7566,
TO_DATE('04-OCT-1992', 'DD-MON-YYYY'), 3000, NULL, 20)

ALTER SESSION
ADVISE ROLLBACK;
```

```
DELETE FROM emp@site2
WHERE empno = 8002;
COMMIT;
```

This transaction has two ALTER SESSION statements with the ADVISE clause. If the transaction becomes in-doubt, SITE1 is sent the advice 'COMMIT' by virtue of the first ALTER SESSION statement and SITE2 is sent the advice 'ROLLBACK' by virtue of the second.

Transaction Control in Procedures and Stored Functions

Procedures and stored functions are written in PL/SQL, and they can issue COMMIT and ROLLBACK statements. If your application performs record management that would be disrupted by a COMMIT or ROLLBACK statement not issued directly by the application itself, you may want to prevent procedures and stored functions called during your session from issuing these statements. You can do this with the following statement:

```
ALTER SESSION DISABLE COMMIT IN PROCEDURE;
```

If you subsequently call a procedure or a stored function that issues a COMMIT or ROLLBACK statement, Oracle returns an error and does not commit or roll back the transaction.

You can subsequently allow procedures and stored functions to issue COMMIT and ROLLBACK statements in your session by issuing the following statement:

```
ALTER SESSION ENABLE COMMIT IN PROCEDURE;
```

This command does not apply to database triggers. Triggers can never issue COMMIT or ROLLBACK statements.

Note: Some applications (such as SQL*Forms) automatically prohibit COMMIT and ROLLBACK statements in procedures and stored functions. Refer to your application documentation.

Parallel DML

When parallel DML is enabled for your session, all DML portions of statements issued are considered for parallel execution. Even with parallel DML enabled, however, some DML operations are restricted from parallelization, while others may still execute serially unless parallel hints and clauses are specified. For a detailed description of parallel DML features and hints, see *Oracle8 Tuning*.

The following restrictions apply to parallel DML operations:

- DML operations on clustered tables are not parallelized.
- DML operations with embedded functions that either write or read database or package states are not parallelized.
- DML operations on tables with triggers that could fire are not parallelized.
- DML operations on tables or schema objects containing object types, LONGs, or LOB datatypes are not parallelized.

Parallel DML mode can be modified only between committed transactions. Issuing this command following an uncommitted transaction will generate an error. Uncommitted transactions must be either committed or rolled back prior to issuing the ALTER SESSION ENABLE | DISABLE | FORCE PARALLEL DML command.

Example I. Issue the following statement to enable parallel DML mode for the current session:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Example II. The following example modifies the current session to check all deferrable constraints immediately following each DML statement:

```
ALTER SESSION SET CONSTRAINTS IMMEDIATE;
```

Example III. The following statement modifies the current session to allow inserts into local index partitions marked as unusable:

```
ALTER SESSION SET SKIP_UNUSABLE_INDEXES=TRUE;
```

Related Topics

ALTER SESSION on page 4-58
SET CONSTRAINT(S) on page 4-514
PL/SQL User's Guide and Reference
Oracle8 Reference
Oracle8 Tuning

ALTER SNAPSHOT

Purpose

To alter a snapshot in one of the following ways:

- changing its storage characteristics
- changing its automatic refresh mode and times

For illustrations of some of these purposes, see “Examples” on page 4-81.

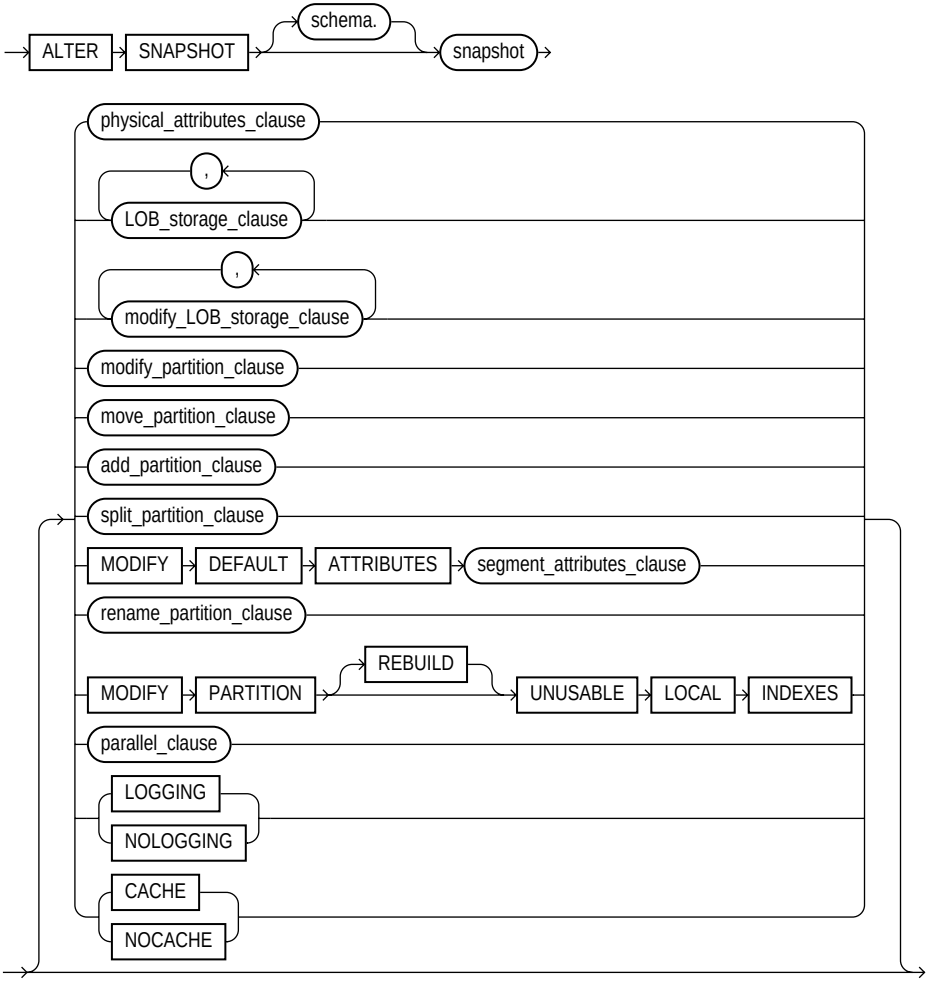
For more information on snapshots, including refreshing snapshots, see CREATE SNAPSHOT on page 4-286.

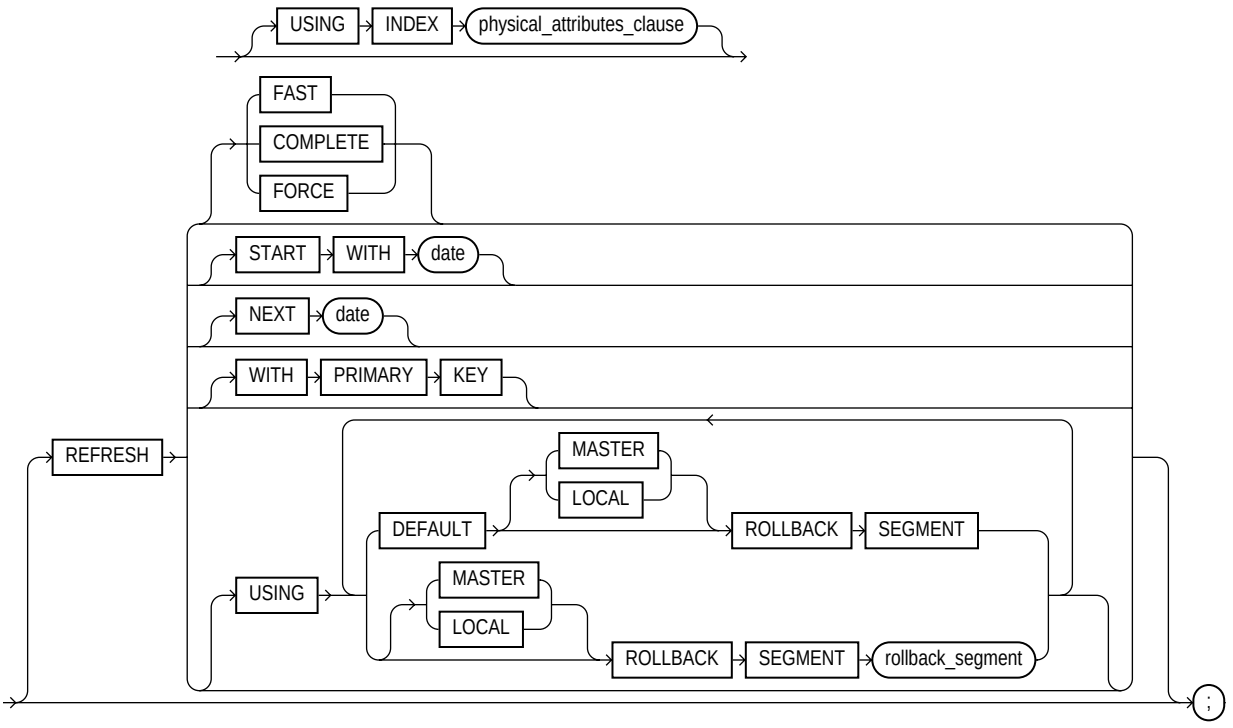
Prerequisites

To alter a snapshot’s storage parameters, the snapshot must be contained in your own schema, or you must have the ALTER ANY SNAPSHOT system privilege.

For detailed information about the prerequisites for ALTER SNAPSHOT, see *Oracle8 Replication*.

Syntax


ALTER
SNAPSHOT
schema.
snapshot
physical_attributes_clause
,
LOB_storage_clause
,
modify_LOB_storage_clause
modify_partition_clause
move_partition_clause
add_partition_clause
split_partition_clause
MODIFY
DEFAULT
ATTRIBUTES
segment_attributes_clause
rename_partition_clause
MODIFY
PARTITION
REBUILD
UNUSABLE
LOCAL
INDEXES
parallel_clause
LOGGING
NOLOGGING
CACHE
NOCACHE



parallel_clause: See the PARALLEL clause on page 4-465
storage_clause: See the STORAGE clause on page 4-523

For the syntax of the following clauses, see ALTER TABLE on page 4-106:

- **physical_attributes_clause**
- **LOB_storage_clause**
- **modify_LOB_storage_clause**
- **modify_partition_clause**
- **move_partition_clause**
- **add_partition_clause**
- **split_partition_clause**
- **modify_default_attributes_clause**
- **rename_partition_clause**

Keywords and Parameters

schema is the schema containing the snapshot. If you omit schema, Oracle assumes the snapshot is in your own schema.

snapshot is the name of the snapshot to be altered.

modify_default_attributes specifies new values for the default attributes of a partitioned table. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

physical_attributes_clause change the values of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics for the internal table that Oracle uses to maintain the snapshot's data. For more information, see CREATE TABLE on page 4-306 and the STORAGE clause on page 4-523.

LOGGING / NOLOGGING specifies the logging attribute. For information about specifying this option, see ALTER TABLE on page 4-106.

CACHE / NOCACHE for data that will be accessed frequently, specifies whether 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 option is useful for small lookup tables. For information about specifying this option, see ALTER TABLE on page 4-106.

LOB_storage_clause specifies the LOB storage characteristics. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

modify_LOB_storage_clause modifies the physical attributes of the LOB attribute *lob_item* or LOB object attribute. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

For more information on the following partitioning clauses, see "Partitioned Snapshots" on page 4-83.

modify_partition_clause modifies the real physical attributes of a table partition. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

move_partition_clause moves table partition *partition_name* to another segment. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

add_partition_clause adds a new partition *new_partition_name* to the "high" end of a partitioned table. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

split_partition_clause creates two new partitions, each with a new segment and new physical attributes, and new initial extents. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

rename_partition_clause renames table partition *partition_name* to *new_partition_name*. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

parallel_clause specifies the degree of parallelism for the snapshot. See the PARALLEL clause on page 4-1022. When this clause is set for master tables, performance for snapshot creation and refresh may improve (depending on the snapshot definition query).

MODIFY PARTITION UNUSABLE LOCAL INDEXES

marks all the local index partitions associated with *partition_name* as unusable.

MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES

rebuilds the unusable local index partitions associated with *partition_name*.

USING INDEX changes the value of INITRANS, MAXTRANS, and STORAGE parameters for the index Oracle uses to maintain the snapshot's data. If USING INDEX is not specified then default values are used for the index.

REFRESH changes the mode and times for automatic refreshes.

- FAST — specifies a fast refresh, or a refresh using the snapshot log associated with the master table.
- COMPLETE — specifies a complete refresh, or a refresh that re-creates the snapshot during each refresh.
- FORCE — specifies a fast refresh if one is possible or complete refresh if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time.

If you omit the FAST, COMPLETE, and FORCE options, Oracle uses FORCE by default.

- START WITH — specifies a date expression for the next automatic refresh time.
- NEXT — specifies a new date expression for calculating the interval between automatic refreshes.

START WITH and NEXT values must evaluate to times in the future.

- WITH PRIMARY KEY — changes a ROWID snapshot to a primary key snapshot. Primary key snapshots allow snapshot master tables to be reorganized without impacting the snapshot's ability to continue to fast refresh. The master table must contain an enabled primary key constraint. See also "Primary Key Snapshots" on page 4-82.

USING MASTER ROLLBACK SEGMENT changes remote master rollback segment to be used during snapshot refresh; *rollback_segment* is the name of the rollback segment to be used. (To change the local snapshot rollback segment, use the DBMS_REFRESH package in *Oracle8 Replication*.) See also "Specifying Rollback Segments" on page 4-82,

- DEFAULT — specifies that Oracle will choose which rollback segment to use. If you specify DEFAULT, you cannot specify *rollback_segment*.

MASTER	specifies the remote rollback segment to be used at the remote master for the individual snapshot.
LOCAL	specifies the remote rollback segment to be used for the local refresh group that contains the snapshot.

Examples

Example I. The following statement changes the automatic refresh mode for the HQ_EMP snapshot to FAST:

```
ALTER SNAPSHOT hq_emp
REFRESH FAST;
```

The next automatic refresh of the snapshot will be a fast refresh provided it is a simple snapshot and its master table has a snapshot log that was created before the snapshot was created or last refreshed.

Because the REFRESH clause does not specify START WITH or NEXT values, the refresh intervals established by the REFRESH clause when the HQ_EMP snapshot was created or last altered are still used.

Example II. The following statement stores a new interval between automatic refreshes for the BRANCH_EMP snapshot:

```
ALTER SNAPSHOT branch_emp
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 BRANCH_EMP snapshot was created or last altered.

At the time of the next automatic refresh, Oracle refreshes the snapshot, evaluates the NEXT expression SYSDATE+7 to determine the next automatic refresh time, and continues to refresh the snapshot automatically once a week.

Because the REFRESH clause does not explicitly specify a refresh mode, Oracle continues to use the refresh mode specified by the REFRESH clause of a previous CREATE SNAPSHOT or ALTER SNAPSHOT statement.

Example III. The following statement specifies a new refresh mode, next refresh time, and new interval between automatic refreshes of the SF_EMP snapshot:

```
ALTER SNAPSHOT sf_emp
REFRESH COMPLETE
```

```
START WITH TRUNC(SYSDATE+1) + 9/24
NEXT SYSDATE+7;
```

The START WITH value establishes the next automatic refresh for the snapshot to be 9:00 AM tomorrow. At that point, Oracle performs a fast refresh of the snapshot, evaluates the NEXT expression, and subsequently refreshes the snapshot every week.

Specifying Rollback Segments

You can specify the rollback segments to be used during a refresh for both the master site and the local site. The master rollback segment is stored on a per-snapshot basis and is validated during snapshot creation and refresh. If the snapshot is complex, the master rollback segment, if specified, is ignored.

You can change local snapshot rollback segments using the DBMS_REFRESH package and is stored at the refresh group level. For information about the DBMS_REFRESH package, see *Oracle8 Replication*. If the auto-refresh parameters (START WITH and NEXT) are specified, a new refresh group is automatically created to refresh the snapshot with a background process. The local rollback segment, if specified, is associated with this new refresh group. An error is raised if the auto-refresh parameters are not specified, but a local rollback segment is.

Note: To direct Oracle to select the rollback segment automatically after one has been specified using CREATE SNAPSHOT or ALTER SNAPSHOT, specify the DEFAULT option with ALTER SNAPSHOT.

Example I. The following example changes the remote master rollback segment used during snapshot refresh to MASTER_SEG:

```
ALTER SNAPSHOT inventory
  REFRESH USING MASTER ROLLBACK SEGMENT master_seg;
```

Example II. The following example changes the remote master rollback segment used during snapshot refresh to one chosen by Oracle:

```
ALTER SNAPSHOT sales REFRESH USING DEFAULT MASTER ROLLBACK SEGMENT;
```

Primary Key Snapshots

To change a ROWID snapshot to a primary key snapshot you must:

- include all columns of the primary key in the snapshot definition
- have an enabled primary key constraint defined on the snapshot master
- ensure that the updatable snapshot log is empty for all updatable snapshots

To fast refresh primary key snapshots you must first create a snapshot master log specifying WITH PRIMARY KEY. The snapshot master log can also store ROWIDs. The snapshot master log must be created before the snapshot is created in order for the snapshots to use the log to fast refresh.

For detailed information about primary key snapshots, see *Oracle8 Replication*.

Note: Primary key snapshots cannot be altered to ROWID snapshots. You must drop the primary key snapshot and re-create it as a ROWID snapshot.

Example I. The following example changes a ROWID to a primary key snapshot:

```
ALTER SNAPSHOT emp_rs REFRESH WITH PRIMARY KEY;
```

Partitioned Snapshots

Partitioned snapshots are the same as partitioned tables because snapshots are basically tables. The options have the same syntax and semantics as the partitioned table options for CREATE TABLE and ALTER TABLE. The only difference is that the following operations are not allowed on snapshots and snapshot logs:

- DROP PARTITION
- TRUNCATE PARTITION
- EXCHANGE PARTITION

You cannot perform bulk deletions by dropping or truncating partitions on master tables. Thus, after dropping or truncating a partition, all snapshots must be refreshed manually. A fast refresh will probably produce incorrect results, but Oracle will not raise an error.

Related Topics

CREATE SNAPSHOT on page 4-286
DROP SNAPSHOT on page 4-402
STORAGE clause on page 4-523

ALTER SNAPSHOT LOG

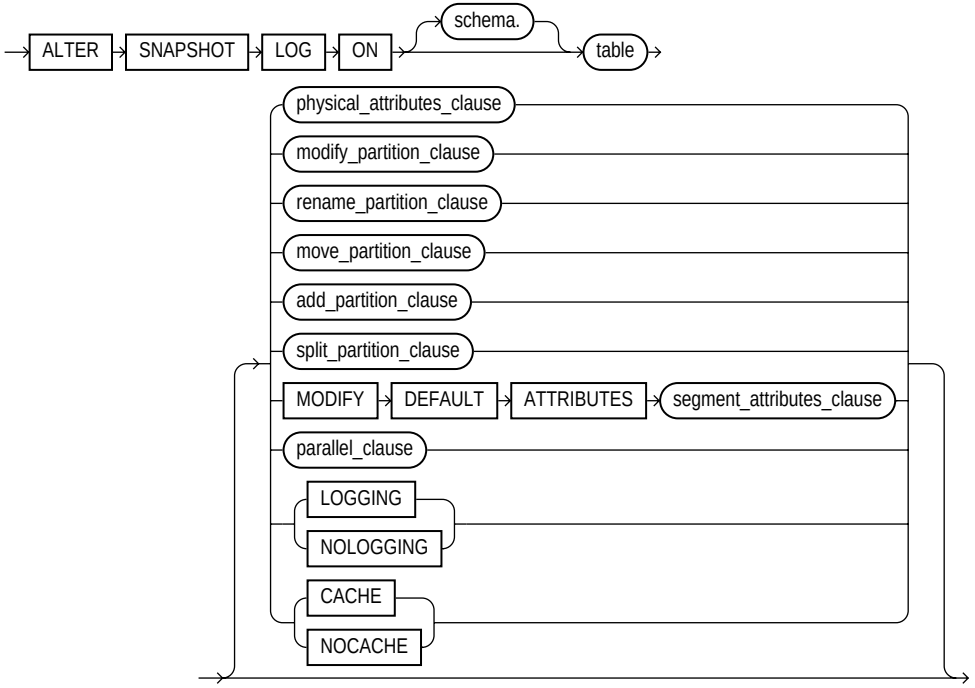
Purpose

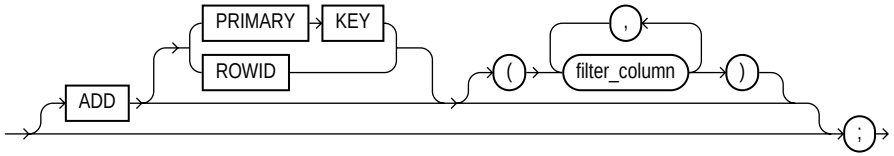
Changes the storage characteristics of a snapshot log. For more information on snapshot logs, see CREATE SNAPSHOT on page 4-286.

Prerequisites

Only the owner of the master table or a user with the SELECT privilege for the master table can alter a snapshot log. For detailed information about the prerequisites for ALTER SNAPSHOT LOG, see *Oracle8 Replication*.

Syntax





For the syntax of the following clauses, see ALTER TABLE on page 4-106:

- **physical_attributes_clause**
- **modify_partition_clause**
- **rename_partition_clause**
- **move_partition_clause**
- **add_partition_clause**
- **split_partition_clause**
- **modify_default_attributes**

Keywords and Parameters

schema	is the schema containing the master table. If you omit *schema*, Oracle assumes the snapshot log is in your own schema.
table	is the name of the master table associated with the snapshot log to be altered.
physical_ attributes_clause	changes the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters for the table, partition, the overflow data segment, or the default characteristics of a partitioned table. See the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters of CREATE TABLE on page 4-306. See the example under "Modifying Physical Attributes" on page 4-86.
rename_partition_ clause	renames table partition *partition_name* to *new_partition_name*. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.
modify_partition_ clause	modifies the real physical attributes of a table partition. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.
move_partition_ clause	moves table partition *partition_name* to another segment. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.
add_partition_ clause	adds a new partition *new_partition_name* to the "high" end of a partitioned table. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106.

split_partition_clause	creates two new partitions, each with a new segment and new physical attributes, and new initial extents. For information about specifying the parameters of this clause, see ALTER TABLE on page 4-106. For more information see "Partitioned Snapshot Logs" on page 4-87.
modify_default_attributes_clause	is a valid option only for a partitioned index. Use this option to specify new values for the default attributes of a partitioned index.
parallel_clause	specifies the degree of parallelism for the snapshot. See the PARALLEL clause on page -1022. When this clause is set for master tables, performance during snapshot creation and refresh may improve (depending on the snapshot definition query).
LOGGING / NOLOGGING	specifies the logging attribute. For information about specifying this option, see ALTER TABLE on page 4-106.
CACHE / NOCACHE	for data that will be accessed frequently, specifies whether 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 option is useful for small lookup tables. For information about specifying this option, see ALTER TABLE on page 4-106.
ADD	changes the snapshot log so that it records the primary key values or ROWID values when rows in the snapshot master table are updated. This clause can also be used to record additional filter columns. PRIMARY KEY — specifies that the primary-key values of all rows updated should be recorded in the snapshot log. ROWID — specifies that the ROWID values of all rows updated should be recorded in the snapshot log. *filter_column(s)* — are non-primary-key columns referenced by snapshots. For information about filter columns, see *Oracle8 Replication*. For more information, see "Adding Primary Key, ROWID, and Filter Columns" on page 4-87.

Modifying Physical Attributes

Example. The following statement changes the MAXEXTENTS value of a snapshot log:

```
ALTER SNAPSHOT LOG ON dept
STORAGE MAXEXTENTS 50;
```

Adding Primary Key, ROWID, and Filter Columns

Snapshot logs can be altered to additionally record primary key, ROWID, or filter column information when snapshot master tables are updated. To stop recording any of this information, you must first drop the snapshot log and then re-create it.

Example. The following example alters an existing ROWID snapshot log to also record primary key information:

```
ALTER SNAPSHOT LOG ON sales ADD PRIMARY KEY;
```

Partitioned Snapshot Logs

Partitioned snapshot logs are the same as partitioned tables, because snapshot logs are basically tables. The options have the same syntax and semantics as the partitioned table options for CREATE TABLE and ALTER TABLE. The only difference is that the following operations are not allowed on snapshots and snapshot logs:

- DROP PARTITION
- TRUNCATE PARTITION
- EXCHANGE PARTITION

You cannot perform bulk deletions by dropping or truncating partitions on master tables. Therefore, after dropping or truncating a partition, all snapshots must be manually refreshed. A fast refresh will probably produce incorrect results, but Oracle will not raise an error.

Related Topics

ALTER TABLE on page 4-106
CREATE SNAPSHOT on page 4-286
CREATE SNAPSHOT LOG on page 4-297
DROP SNAPSHOT LOG on page 4-403

ALTER SYSTEM

Purpose

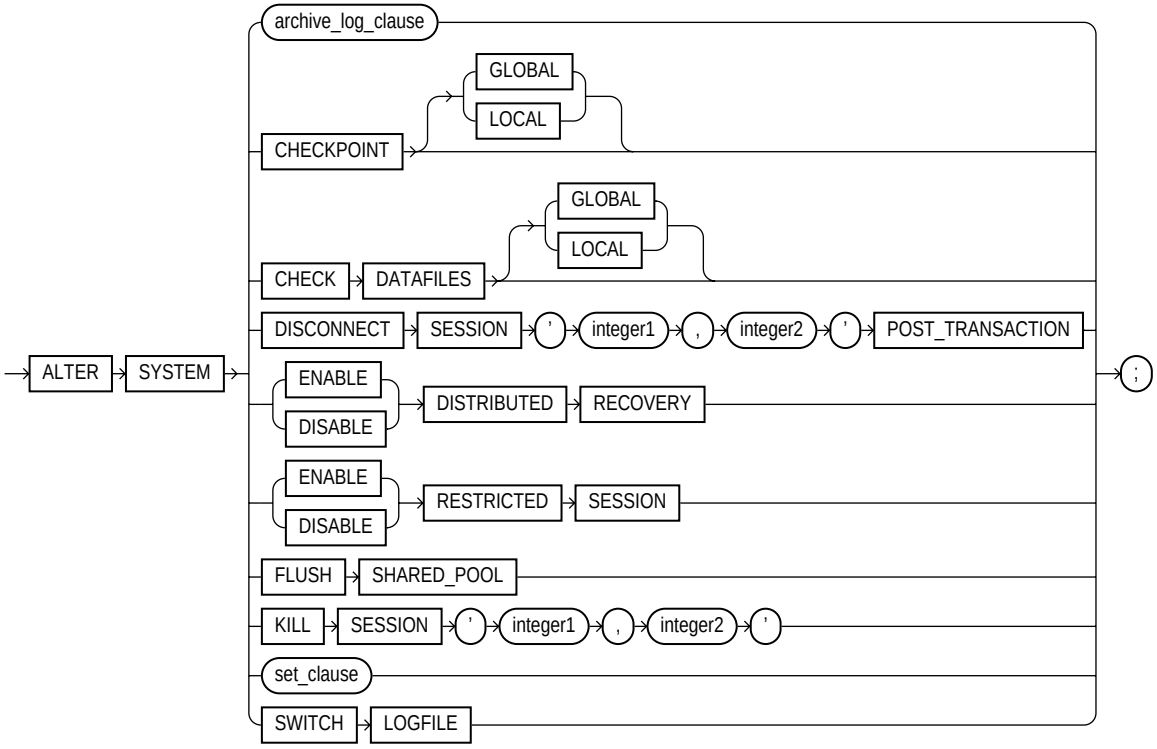
To dynamically alter your Oracle instance in one of the following ways:

- to explicitly perform a checkpoint
- to verify access to datafiles
- to terminate a session
- to enable or disable resource limits
- to enable distributed recovery in a single-process environment
- to disable distributed recovery
- to restrict logons to Oracle to only those users with RESTRICTED SESSION system privilege
- to clear all data from the shared pool in the system global area (SGA)
- to enable or disable global name resolution
- to dynamically change or disable limits or thresholds for concurrent usage licensing and named user licensing
- to manually archive redo log file groups or to enable or disable automatic archiving
- to manage shared server processes or dispatcher processes for the multithreaded server architecture
- to explicitly switch redo log file groups

Prerequisites

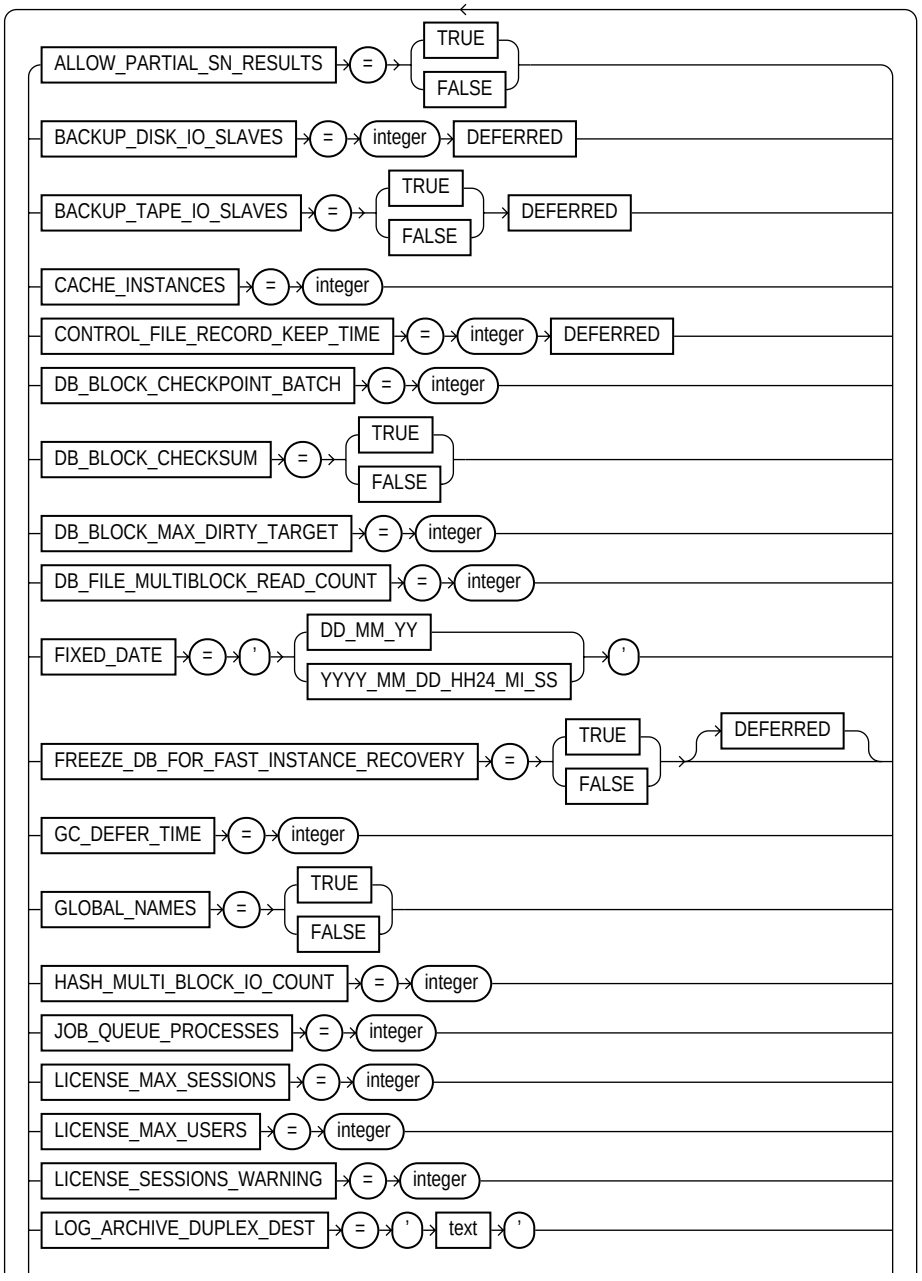
You must have ALTER SYSTEM system privilege.

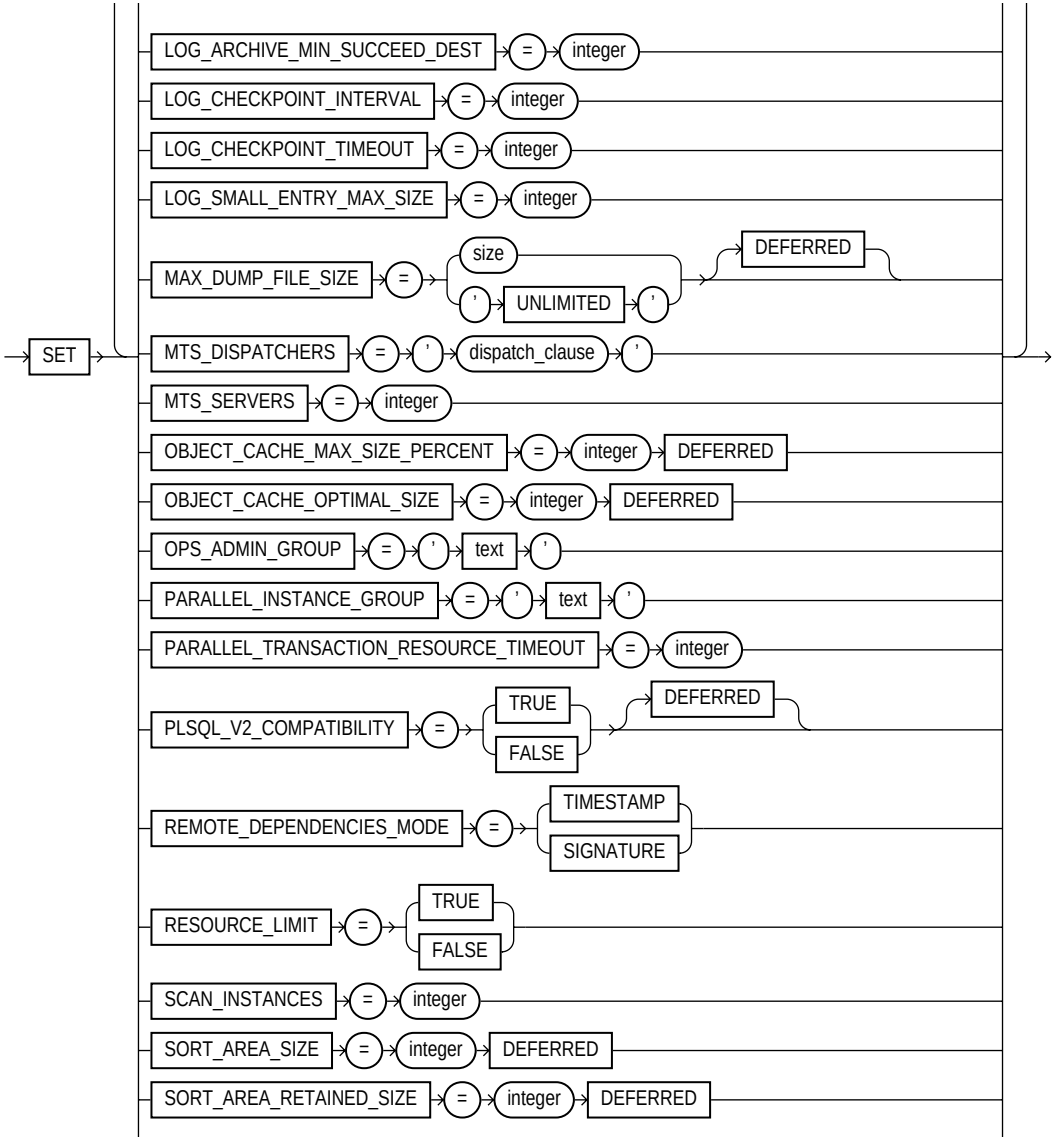
Syntax

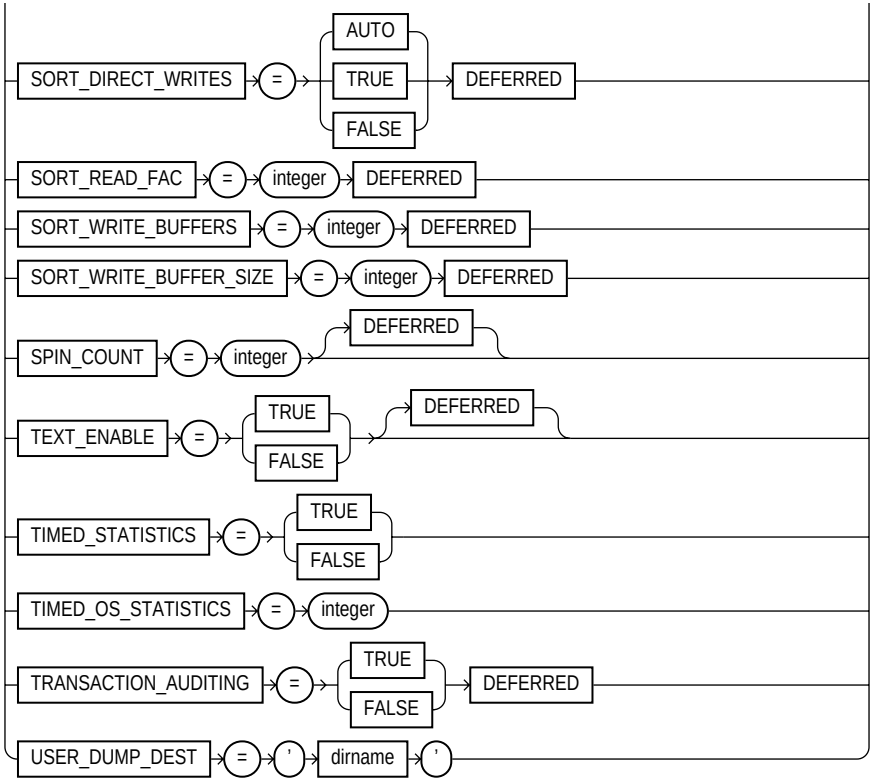


archive_log_clause: See the ARCHIVE LOG clause on page 4-167.

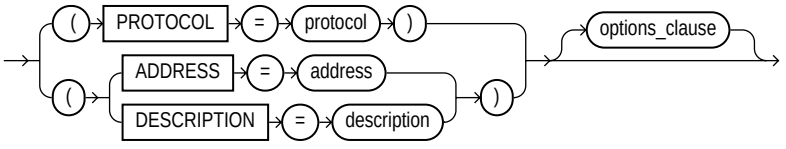
set_clause::=


ALLOW_PARTIAL_SN_RESULTS
=
TRUE
FALSE
BACKUP_DISK_IO_SLAVES
=
integer
DEFERRED
BACKUP_TAPE_IO_SLAVES
=
TRUE
FALSE
DEFERRED
CACHE_INSTANCES
=
integer
CONTROL_FILE_RECORD_KEEP_TIME
=
integer
DEFERRED
DB_BLOCK_CHECKPOINT_BATCH
=
integer
DB_BLOCK_CHECKSUM
=
TRUE
FALSE
DB_BLOCK_MAX_DIRTY_TARGET
=
integer
DB_FILE_MULTIBLOCK_READ_COUNT
=
integer
FIXED_DATE
=
'
DD_MM_YY
YYYY_MM_DD_HH24_MI_SS
'
FREEZE_DB_FOR_FAST_INSTANCE_RECOVERY
=
TRUE
FALSE
DEFERRED
GC_DEFER_TIME
=
integer
GLOBAL_NAMES
=
TRUE
FALSE
HASH_MULTI_BLOCK_IO_COUNT
=
integer
JOB_QUEUE_PROCESSES
=
integer
LICENSE_MAX_SESSIONS
=
integer
LICENSE_MAX_USERS
=
integer
LICENSE_SESSIONS_WARNING
=
integer
LOG_ARCHIVE_DUPLEX_DEST
=
'
text
'

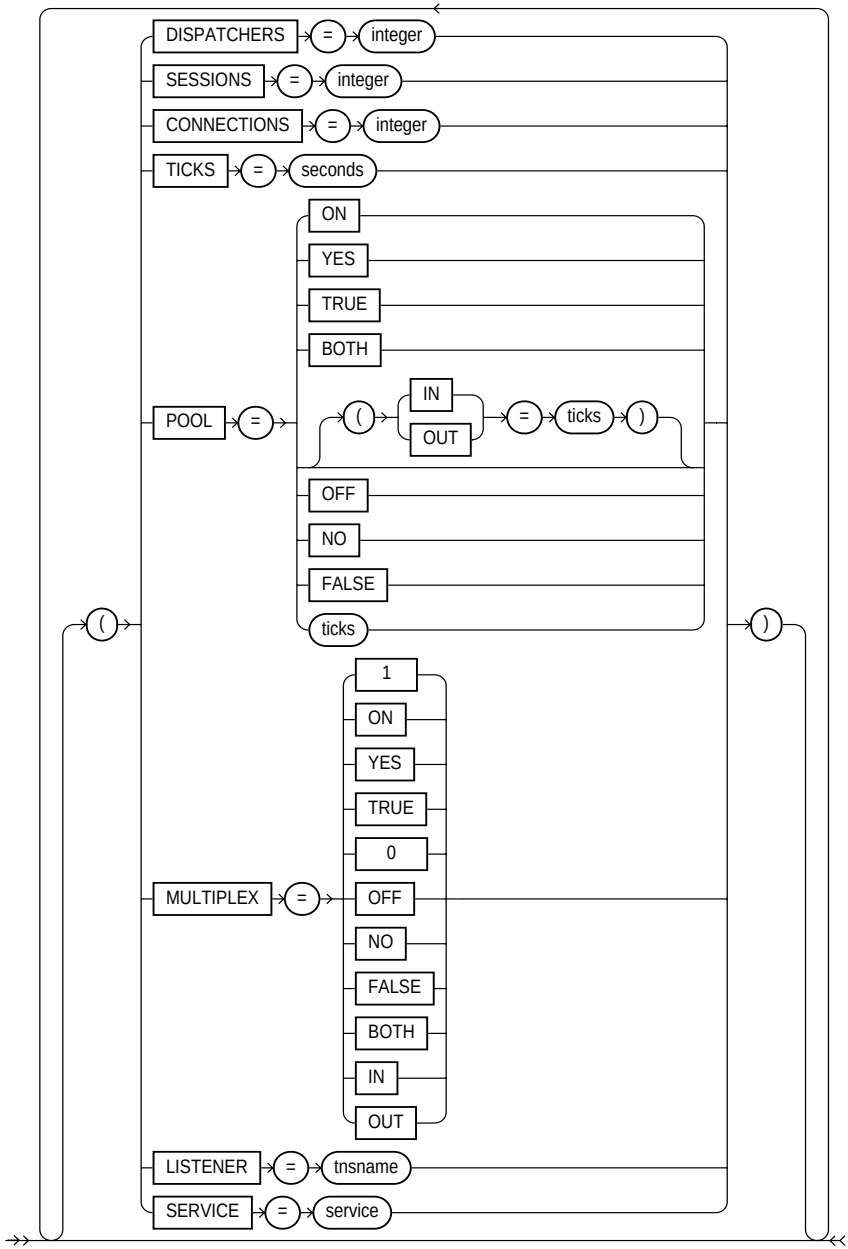

SET
LOG_ARCHIVE_MIN_SUCCEED_DEST
=
integer
LOG_CHECKPOINT_INTERVAL
=
integer
LOG_CHECKPOINT_TIMEOUT
=
integer
LOG_SMALL_ENTRY_MAX_SIZE
=
integer
MAX_DUMP_FILE_SIZE
=
size
'
UNLIMITED
'
DEFERRED
MTS_DISPATCHERS
=
'
dispatch_clause
'
MTS_SERVERS
=
integer
OBJECT_CACHE_MAX_SIZE_PERCENT
=
integer
DEFERRED
OBJECT_CACHE_OPTIMAL_SIZE
=
integer
DEFERRED
OPS_ADMIN_GROUP
=
'
text
'
PARALLEL_INSTANCE_GROUP
=
'
text
'
PARALLEL_TRANSACTION_RESOURCE_TIMEOUT
=
integer
PLSQL_V2_COMPATIBILITY
=
TRUE
FALSE
DEFERRED
REMOTE_DEPENDENCIES_MODE
=
TIMESTAMP
SIGNATURE
RESOURCE_LIMIT
=
TRUE
FALSE
SCAN_INSTANCES
=
integer
SORT_AREA_SIZE
=
integer
DEFERRED
SORT_AREA_RETAINED_SIZE
=
integer
DEFERRED



dispatch_clause::=



options_clause::=



Keywords and Parameters

You can use the following options regardless of whether your instance has the database dismounted or mounted, open or closed:

RESTRICTED SESSION specifies whether logon to Oracle is restricted

- **ENABLE** allows only users with RESTRICTED SESSION system privilege to logon to Oracle.
- **DISABLE** reverses the effect of the ENABLE RESTRICTED SESSION option, allowing all users with CREATE SESSION system privilege to log on to Oracle.

For more information, see "Restricting Logons" on page 4-97.

FLUSH SHARED_POOL clears all data from the shared pool in the system global area (SGA). For more information, see "Clearing the Shared Pool" on page 4-97.

You can use the following options when your instance has the database mounted, open or closed:

CHECKPOINT performs a checkpoint.

- **GLOBAL** performs a checkpoint for all instances that have opened the database.
- **LOCAL** performs a checkpoint only for the thread of redo log file groups for your instance. You can use this option only when your instance has the database open.

If you omit both the GLOBAL and LOCAL options, Oracle performs a global checkpoint. For more information, see "Performing a Checkpoint" on page 4-97.

CHECK DATAFILES

- **GLOBAL** verifies that all instances that have opened the database can access all online datafiles.
- **LOCAL** verifies that your instance can access all online datafiles.

If you omit both the GLOBAL and LOCAL options, Oracle uses GLOBAL by default. For more information, see "Checking Datafiles" on page 4-98.

You can use the following parameters and options only when your instance has the database open:

RESOURCE_LIMIT controls resource limits. TRUE enables resource limits; FALSE disables resource limits. See also "Using Resource Limits" on page 4-98.

GLOBAL_NAMES controls the enforcement of global name resolution for your session. TRUE enables the enforcement of global names; FALSE disables the enforcement of global names. For more information, see "Global Name Resolution" on page 4-99.

SCAN_INSTANCES in a parallel server, specifies the number of instances to participate in parallelized operations. This syntax will be obsolete in the next major release.

CACHE_INSTANCES — in a parallel server, specifies the number of instances that will cache a table. This syntax will be obsolete in the next major release.

For more information on parallel operations, see *Oracle8 Tuning*.

For more information on the following multithreaded server parameters, see "Managing Processes for the Multithreaded Server" on page 4-99.

MTS_SERVERS — specifies a new minimum number of shared server processes.

MTS_DISPATCHERS — specifies a new number of dispatcher processes:

- *protocol* — is the network protocol of the dispatcher processes.
- *integer* — is the new number of dispatcher processes of the specified protocol.

You can specify multiple MTS_DISPATCHERS parameters in a single command for multiple network protocols.

For more information on the following licensing parameters, see "Using Licensing Limits" on page 4-101.

JOB_QUEUE_PROCESSES — specifies the number of job queue processes per instance (SNPn, where n is 0 to 9 followed by A to Z). Set this parameter to 1 or higher if you wish to have your snapshots updated automatically. One job queue process is usually sufficient unless you have many snapshots that refresh simultaneously.

Oracle also uses job queue processes to process requests created by the DBMS_JOB package. For more information on managing table snapshots, see *Oracle8 Replication*.

LICENSE_MAX_SESSIONS — limits the number OS sessions on your instance. A value of 0 disables the limit.

LICENSE_SESSIONS_WARNING — establishes a threshold of sessions over which Oracle writes warning messages to the ALERT file for subsequent sessions. A value of 0 disables the warning threshold.

LICENSE_MAX_USERS — limits number of concurrent users on your database. A value of 0 disables the limit.

REMOTE_DEPENDENCIES_MODE — specifies how dependencies of remote stored procedures are handled by the server. For more information, refer to *Oracle8 Application Developer's Guide*.

SWITCH LOGFILE — switches redo log file groups. For more information, see "Switching Redo Log File Groups" on page 4-102.

DISTRIBUTED RECOVERY — specifies whether or not distributed recovery is enabled.

- ENABLE — enables distributed recovery. In a single-process environment, you must use this option to initiate distributed recovery.

	DISABLE	switches redo log files.
	For more information, see "Enabling and Disabling Distributed Recovery" on page 4-103.	
ARCHIVE LOG	manually archives redo log files or enables or disables automatic archiving. See the ARCHIVE LOG clause on page 4-167.	
KILL SESSION	terminates a session and any ongoing transactions. You must identify the session with both of the following values from the V$SESSION view:	
	integer1	is the value of the SID column.
	integer2	is the value of the SERIAL# column.
	For more information, see "Terminating a Session" on page 4-103.	
DISCONNECT SESSION	disconnects the current session by destroying the dedicated server process (or virtual circuit if the connection was made via MTS). If configured, application failover will take effect. For more information about application failover see *Oracle8 Tuning* and *Oracle8 Parallel Server Concepts and Administration*. You must identify the session with both of the following values from the V$SESSION view:	
	integer1	is the value of the SID column.
	integer2	is the value of the SERIAL# column.
	POST_ TRANSACTION	allows ongoing transactions to complete before the session is disconnected. This keyword is required when DISCONNECT SESSION is specified. For more information, see "Disconnecting a Session" on page 4-104.
PLSQL_V2_ COMPATIBILITY	modifies the compile-time behavior of PL/SQL programs to allow language constructs that are illegal in Oracle8 (PL/SQL V3), but were legal in Oracle7 (PL/SQL V2). See the *PL/SQL User's Guide and Reference* and *Oracle8 Reference* for more information about this system parameter.	
	TRUE	enables Oracle8 PL/SQL V3 programs to execute Oracle7 PL/SQL V2 constructs.
	FALSE	disallows illegal Oracle7 PL/SQL V2 constructs. This is the default.
MAX_DUMP_ FILE_SIZE	specifies the trace dump file size upper limit for all user sessions. Specify the maximum *size* as either a nonnegative integer that represents the number of blocks, or as 'UNLIMITED'. If you specify 'UNLIMITED', no upper limit is imposed.	
	DEFERRED	modifies the trace dump file size upper limit for future user sessions only.

Restricting Logons

By default, any user granted CREATE SESSION system privilege can log on to Oracle. The ENABLE RESTRICTED SESSION option of the ALTER SYSTEM command prevents logons by all users except those having RESTRICTED SESSION system privilege. Existing sessions are not terminated.

You may want to restrict logons if you are performing application maintenance and you want only application developers with RESTRICTED SESSION system privilege to log on. To restrict logons, 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 command.

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;
```

Clearing the Shared Pool

The FLUSH SHARED_POOL option of the ALTER SYSTEM command clears all information from the shared pool in the system global area (SGA). The shared pool stores this information:

- cached data dictionary information
- shared SQL and PL/SQL areas for SQL statements, stored procedures, functions, packages, and triggers

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;
```

The above statement does not clear shared SQL and PL/SQL areas for SQL statements, stored procedures, functions, packages, or triggers that are currently being executed, or for SQL SELECT statements for which all rows have not yet been fetched.

Performing a Checkpoint

The CHECKPOINT clause of the ALTER SYSTEM command explicitly forces Oracle to perform a checkpoint. You can force a checkpoint if you want to ensure

that all changes made by committed transactions are written to the data files on disk. For more information on checkpoints, see the "Recovery Structures" chapter of *Oracle8 Concepts*.

If you are using Oracle with the Parallel Server option in parallel mode, you can specify either the GLOBAL option to perform a checkpoint on all instances that have opened the database or the LOCAL option to perform a checkpoint on only your instance.

The following statement forces a checkpoint:

```
ALTER SYSTEM
CHECKPOINT;
```

Oracle does not return control to you until the checkpoint is complete.

Checking Datafiles

The CHECK DATAFILES clause of the ALTER SYSTEM command verifies access to all online datafiles. If any datafile is not accessible, Oracle writes a message to an ALERT file. You may want to perform this operation after fixing a hardware problem that prevented an instance from accessing a datafile. For more information on using this clause, see *Oracle8 Parallel Server Concepts and Administration*.

The following statement verifies that all instances that have opened the database can access all online datafiles:

```
ALTER SYSTEM
CHECK DATAFILES GLOBAL;
```

Using Resource Limits

When you start an instance, Oracle enables or disables resource limits based on the value of the initialization parameter RESOURCE_LIMIT. You can issue an ALTER SYSTEM statement with the RESOURCE_LIMIT option to enable or disable resource limits for subsequent sessions.

Enabling resource limits only causes Oracle to enforce the resource limits already assigned to users. To choose resource limit values for a user, you must create a *profile*, or a set of limits, and assign that profile to the user. For more information on this process, see CREATE PROFILE on page 4-265 and CREATE USER on page 4-357.

This ALTER SYSTEM statement dynamically enables resource limits:

```
ALTER SYSTEM
SET RESOURCE_LIMIT = TRUE;
```

Global Name Resolution

When you start an instance, Oracle determines whether to enforce global name resolution for remote objects accessed in SQL statements based on the value of the initialization parameter GLOBAL_NAMES. You can subsequently enable or disable global name resolution while your instance is running with the GLOBAL_NAMES parameter of the ALTER SYSTEM command. You can also enable or disable global name resolution for your session with the GLOBAL_NAMES parameter of the ALTER SESSION command discussed earlier in this chapter.

Oracle recommends that you enable global name resolution if you use or plan to use distributed processing. For more information on global name resolution and how Oracle enforces it, see "Referring to Objects in Remote Databases" on page 2-54 and *Oracle8 Distributed Database Systems*.

Managing Processes for the Multithreaded Server

When you start your instance, Oracle creates shared server processes and dispatcher processes for the multithreaded server architecture based on the values of the following initialization parameters:

MTS_SERVERS	specifies the initial and minimum number of shared server processes. Oracle may automatically change the number of shared server processes if the load on the existing processes changes. While your instance is running, the number of shared server processes can vary between the values of the initialization parameters MTS_SERVERS and MTS_MAX_SERVERS.
MTS_DISPATCHERS	specifies one or more network protocols and the number of dispatcher processes for each protocol.

For more information on the multithreaded server architecture, see *Oracle8 Concepts*.

You can use the MTS_SERVERS and MTS_DISPATCHERS parameters of the ALTER SYSTEM command to perform one of the following operations while the instance is running:

- **To create additional shared server processes:** You can cause Oracle to create additional shared server processes by increasing the minimum number of shared server processes.
- **To terminate existing shared server processes:** Oracle terminates the shared server processes after it finishes processing their current calls, unless the load

on the server processes is so high that it cannot be managed by the remaining processes.

- **To create more dispatcher processes for a specific protocol:** You can create additional dispatcher processes up to a maximum across all protocols specified by the initialization parameter MTS_MAX_DISPATCHERS.

  You cannot use this command to create dispatcher processes for network protocols that are not specified by the initialization parameter MTS_ DISPATCHERS. To create dispatcher processes for a new protocol, you must change the value of the initialization parameter.

- **To terminate existing dispatcher processes for a specific protocol:** Oracle terminates the dispatcher processes only after their current user processes disconnect from the instance.

Example I. The following statement changes the minimum number of shared server processes to 25:

```
ALTER SYSTEM
SET MTS_SERVERS = 25;
```

If there are currently fewer than 25 shared server processes, Oracle creates more. If there are currently more than 25, Oracle terminates some of them when they are finished processing their current calls if the load could be managed by the remaining 25.

Example II. 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 DECNET protocol to 10:

```
ALTER SYSTEM
SET MTS_DISPATCHERS = 'TCP, 5'
MTS_DISPATCHERS = 'DECnet, 10';
```

If there are currently fewer than 5 dispatcher processes for TCP, Oracle creates new ones. If there are currently more than 5, Oracle terminates some of them after the connected users disconnect.

If there are currently fewer than 10 dispatcher processes for DECnet, Oracle creates new ones. If there are currently more than 10, Oracle terminates some of them after the connected users disconnect.

If there are currently existing dispatchers for another protocol, the above statement does not affect the number of dispatchers for that protocol.

Using Licensing Limits

Oracle enforces concurrent usage licensing and named user licensing limits specified by your Oracle license. When you start your instance, Oracle establishes the licensing limits based on the values of the following initialization parameters:

LICENSE_ MAX_ SESSIONS	establishes the concurrent usage licensing limit, or the limit for concurrent sessions. Once this limit is reached, only users with RESTRICTED SESSION system privilege can connect.
LICENSE_ SESSIONS_ WARNING	establishes a warning threshold for concurrent usage. Once this threshold is reached, Oracle writes warning messages to the database ALERT file for each subsequent session. Also, users with RESTRICTED SESSION system privilege receive warning messages when they begin subsequent sessions.
LICENSE_ MAX_USERS	establishes the limit for users connected to your database. Once this limit is reached, more users cannot connect.

You can dynamically change or disable limits or thresholds while your instance is running using the LICENSE_MAX_SESSIONS, LICENSE_SESSIONS_WARNING, and LICENSE_MAX_USERS parameters of the ALTER SYSTEM command. Do not disable or raise session or user limits unless you have appropriately upgraded your Oracle license. For information on upgrading your license, contact your Oracle sales representative.

New limits apply only to future sessions and users:

- If you reduce the limit on sessions below the current number of sessions, Oracle does not end existing sessions to enforce the new limit. Users without RESTRICTED SESSION system privilege can begin new sessions only when the number of sessions falls below the new limit.
- If you reduce the warning threshold for sessions below the current number of sessions, Oracle writes a message to the ALERT file for all subsequent sessions.
- You cannot reduce the limit on users below the current number of users created for the database.

Example I. 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, Oracle 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, only users with RESTRICTED SESSION system privilege can begin new sessions until the number of sessions falls below 64 again.

Example II. The following statement dynamically disables the limit for sessions on your instance:

```
ALTER SYSTEM
SET LICENSE_MAX_SESSIONS = 0;
```

After you issue the above statement, Oracle no longer limits the number of sessions on your instance.

Example III. The following statement dynamically changes the limit on the number of users in the database to 200:

```
ALTER SYSTEM
SET LICENSE_MAX_USERS = 200;
```

After you issue the above statement, Oracle prevents the number of users in the database from exceeding 200.

Switching Redo Log File Groups

The SWITCH LOGFILE option of the ALTER SYSTEM command explicitly forces Oracle to begin writing to a new redo log file group, regardless of whether the files in the current redo log file group are full. You may 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 is writing to it. The forced log switch affects only your instance's redo log thread. Note that when you force a log switch, Oracle begins to perform a checkpoint. Oracle returns control to you immediately rather than when the associated checkpoint is complete.

The following statement forces a log switch:

```
ALTER SYSTEM
SWITCH LOGFILE;
```

Enabling and Disabling Distributed Recovery

Oracle allows you to perform distributed transactions, or transactions that modify data on multiple databases. If a network or machine failure occurs during the commit process for a distributed transaction, the state of the transaction may be unknown, or *in doubt*. Once the failure has been corrected and the network and its nodes are back online, Oracle recovers the transaction.

If you are using Oracle in multiple-process mode, this distributed recovery is performed automatically. If you are using Oracle in single-process (single user) mode, such as on the MS-DOS operating system, you must explicitly initiate distributed recovery with the following statement.

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

You may need to issue the above statement more than once to recover an in-doubt transaction, especially if the remote node involved in the transaction is not accessible. In-doubt transactions appear in the data dictionary view DBA_2PC_PENDING. You can tell that the transaction is recovered when it no longer appears in DBA_2PC_PENDING. For more information about distributed transactions and distributed recovery, see *Oracle8 Distributed Database Systems.*

You can disable distributed recovery in both single-process and multiprocess mode with the following statement:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

You may want to disable distributed recovery for demonstration purposes. You can then enable distributed recovery again by issuing an ALTER SYSTEM statement with the ENABLE DISTRIBUTED RECOVERY clause.

Terminating a Session

The KILL SESSION clause of the ALTER SYSTEM command terminates a session, immediately performing the following tasks:

- rolling back its current transactions
- releasing all of its locks
- freeing all of its resources

You may want to kill 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 killed; that user can no longer make calls to the database without beginning a new session. You can kill a session only on the same instance as your current session.

If you try to kill a session that is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, Oracle waits for this activity to complete, kills the session, and then returns control to you. If the waiting lasts as long as a minute, Oracle marks the session to be killed and returns control to you with a message indicating that the session is marked to be killed. Oracle then kills the session when the activity is complete.

Example. Consider this data from the V$SESSION dynamic performance table:

```
SELECT sid, serial, username
FROM v$session

  SID    SERIAL USERNAME
----- --------- ----------------
    1         1
    2         1
    3         1
    4         1
    5         1
    7         1
    8        28 OPS$BQUIGLEY
   10       211 OPS$SWIFT
   11        39 OPS$OBRIEN
   12        13 SYSTEM
   13         8 SCOTT
```

The following statement kills the session of the user SCOTT using the SID and SERIAL# values from V$SESSION:

```
ALTER SYSTEM
KILL SESSION '13, 8';
```

Disconnecting a Session

The DISCONNECT SESSION clause is similar to the KILL SESSION clause, but with two distinct differences.

First, the ALTER SYSTEM DISCONNECT SESSION 'X, Y' POST_TRANSACTION command waits until any current transaction that the session is working on completes before taking effect.

Second, the session is disconnected rather than killed, which means that the dedicated server process (or virtual circuit if the connection was made through MTS) is destroyed by this command. Termination of a session's connection causes

application failover to take effect if the appropriate system parameters are configured.

Disconnecting a session essentially allows you to perform a manual application failover. Using this command in a parallel server environment allows you to disconnect sessions on an overloaded instance and shift them to another instance.

The POST_TRANSACTION keyword is required.

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;
```

For more information about application failover, see *Oracle8 Parallel Server Concepts and Administration* and *Oracle8 Tuning.*

Related Topics

ALTER SESSION on page 4-58
ALTER SESSION on page 4-58
CREATE USER on page 4-357
ARCHIVE LOG clause on page 4-167

ALTER TABLE

Purpose

To alter the definition of a table in one of the following ways:

- add a column
- add an integrity constraint
- redefine a column (datatype, size, default value)
- modify storage characteristics or other parameters
- modify the real storage attributes of a nonpartitioned table or the default attributes of a partitioned table
- enable, disable, or drop an integrity constraint or trigger
- explicitly allocate an extent
- explicitly deallocate the unused space of a table
- allow or disallow writing to a table
- modify the degree of parallelism for a table
- modify the logging attributes of a non-partitioned table, partitioned table or table partition(s)
- modify the CACHE / NOCACHE attributes
- add, modify, split, move, drop, or truncate table partitions
- rename a table or a table partition
- add or modify index-organized table characteristics
- add or modify LOB columns
- OBJ add or modify object type, nested table type, or VARRAY type column
- OBJ add integrity constraints to object type columns

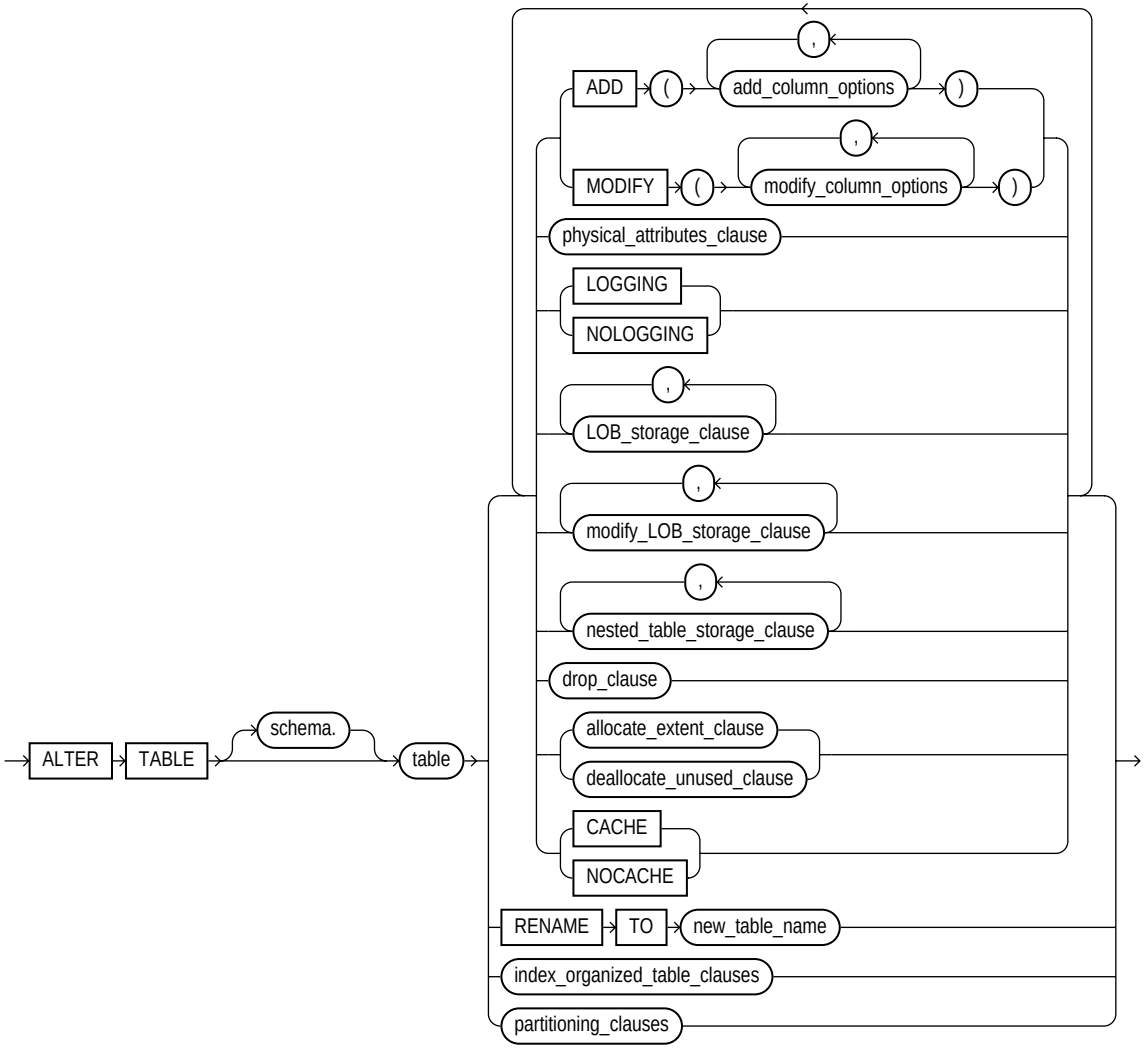
Note: Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

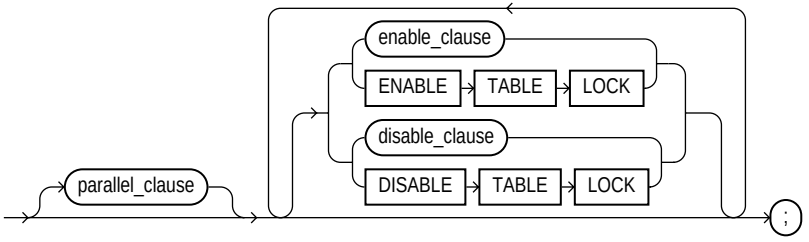
Prerequisites

The table must be in your own schema, or you must have ALTER privilege on the table, or you must have ALTER ANY TABLE system privilege. For some operations you may also need the CREATE ANY INDEX privilege.

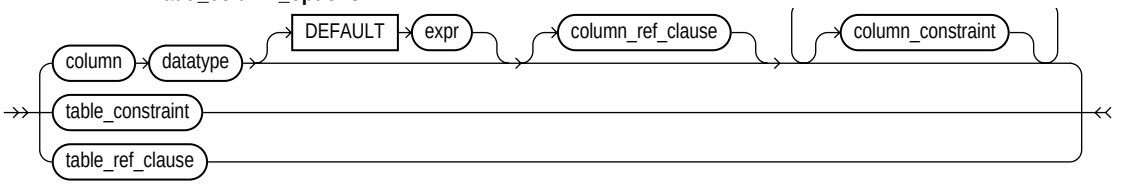
OBJ To use an object type in a column definition when modifying a table, either that object must belong to the same schema as the table being altered, or you must have either the EXECUTE ANY TYPE system privilege or the EXECUTE schema object privilege for the object type.

Syntax


ALTER
TABLE
schema.
table
,
ADD
(
add_column_options
)
,
MODIFY
(
modify_column_options
)
physical_attributes_clause
LOGGING
NOLOGGING
,
LOB_storage_clause
,
modify_LOB_storage_clause
,
nested_table_storage_clause
drop_clause
allocate_extent_clause
deallocate_unused_clause
CACHE
NOCACHE
RENAME
TO
new_table_name
index_organized_table_clauses
partitioning_clauses

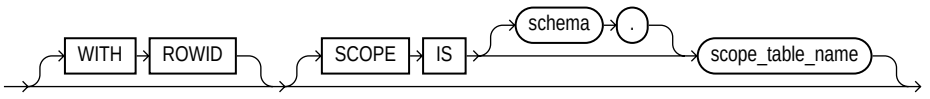


add_column_options::=

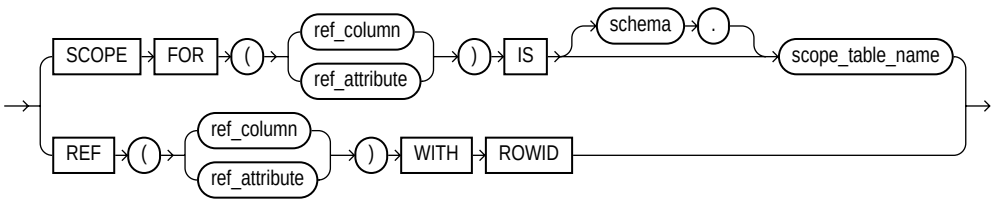


column_constraint, table_constraint: See the CONSTRAINT clause on page 4-188

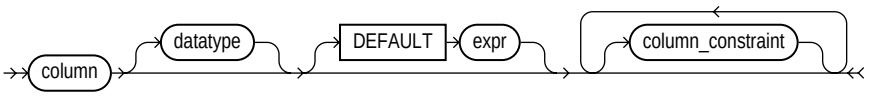
column_ref_clause::=



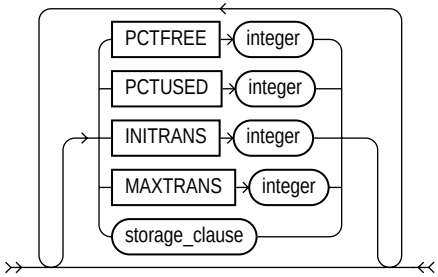
table_ref_clause::=



modify_column_options::=

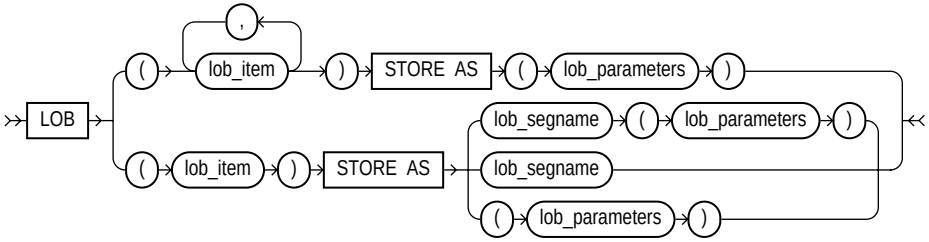


physical_attributes_clause::=

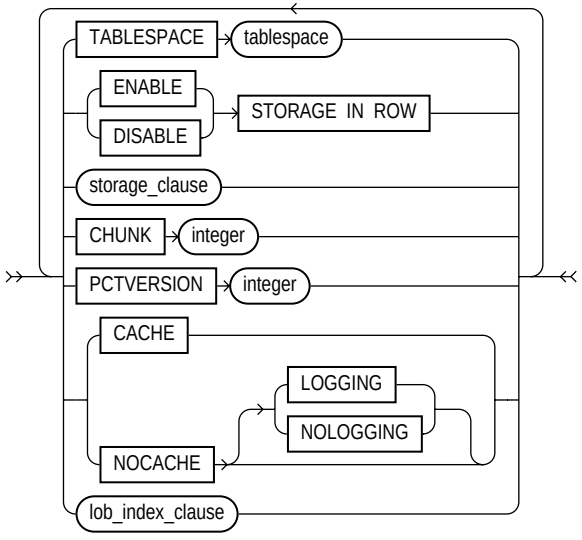


storage_clause: See STORAGE clause on page 4-523.

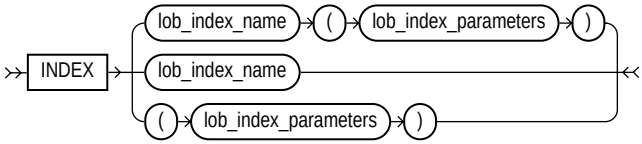
LOB_storage_clause::=



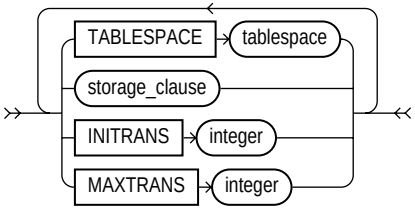
LOB_parameters::=



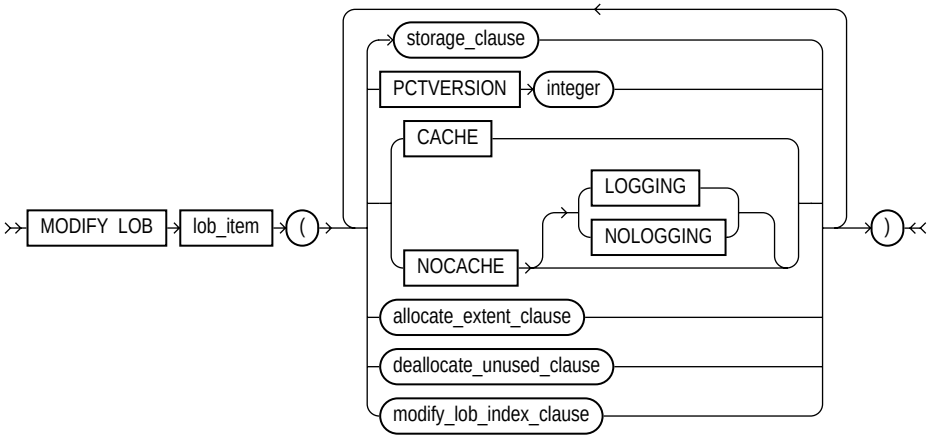
LOB_index_clause::=



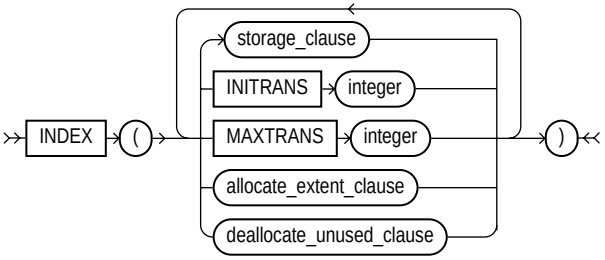
LOB_index_parameters::=



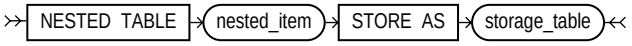
modify_LOB_storage_clause::=



modify_LOB_index_clause::=

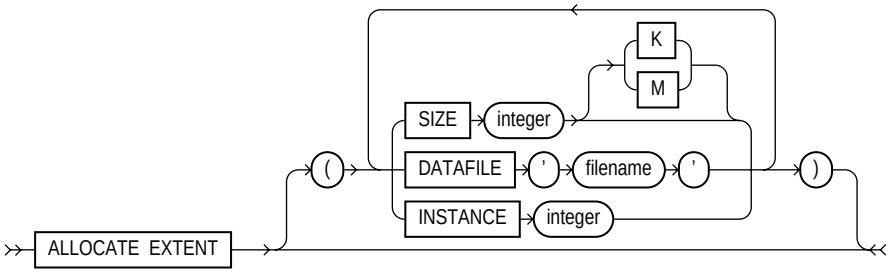


nested_table_storage_clause::=



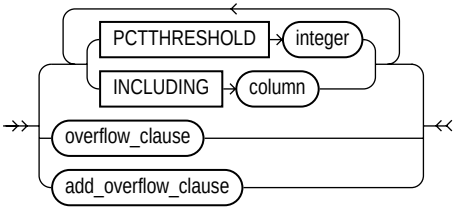
drop_clause: See the DROP clause on page 4-384.

allocate_extent_clause::=

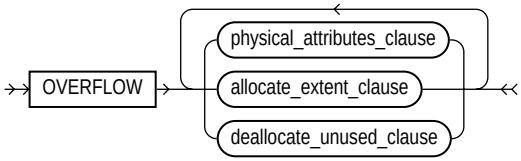


deallocate_unused_clause: See the DEALLOCATE UNUSED clause on page 4-372.

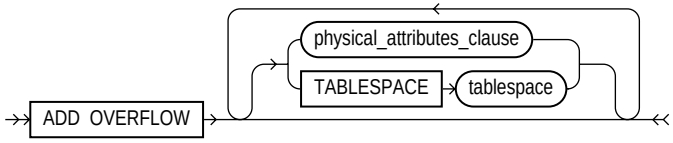
index_organized_table_clauses::=



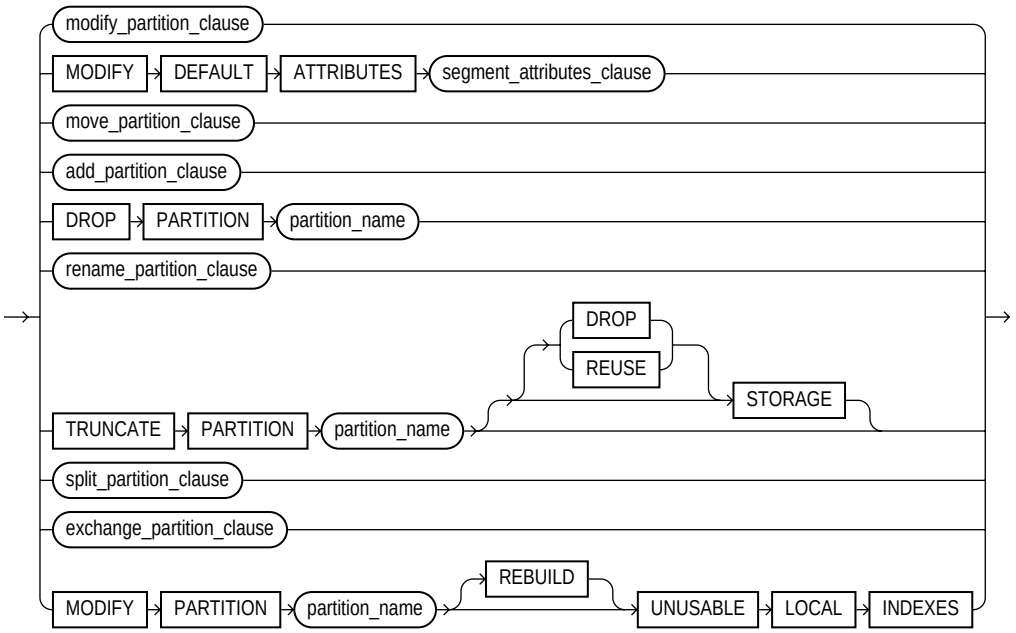
overflow_clause ::=



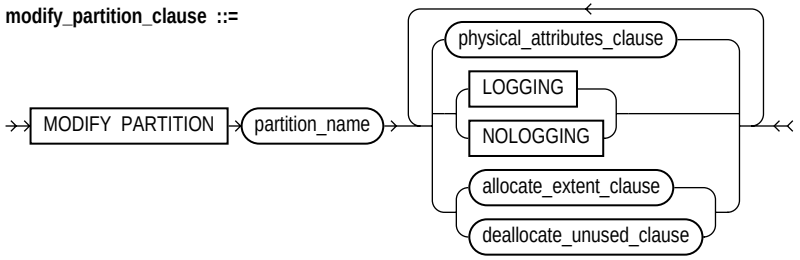
add_overflow_clause ::=



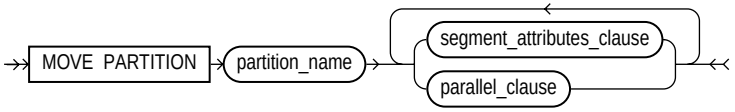
partitioning_clauses::=



modify_partition_clause ::=

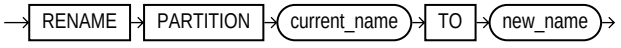


move_partition_clause ::=

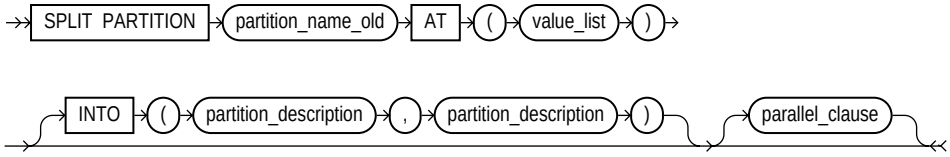


add_partition_clause ::=

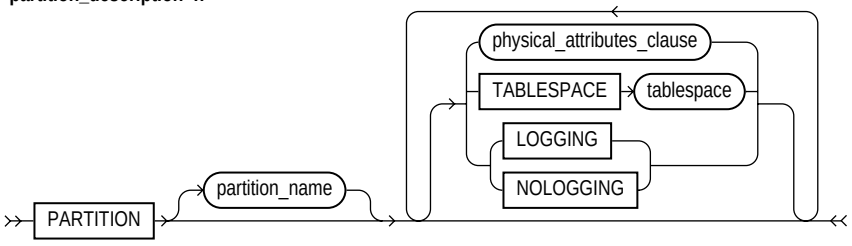
rename_partition_clause::=



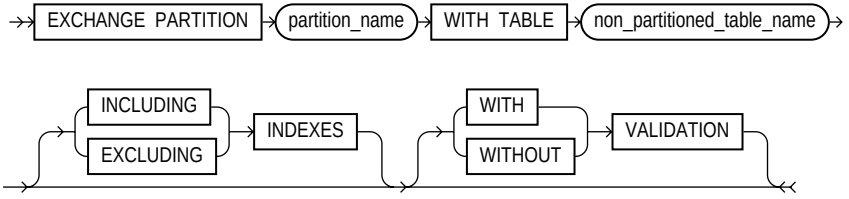
split_partition_clause ::=



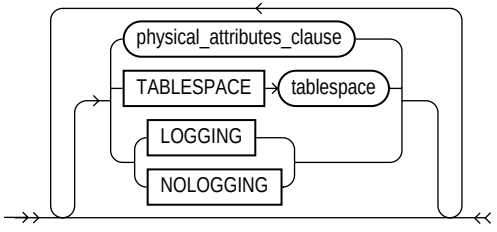
partition_description ::=



exchange_partition_clause ::=



segment_attributes_clause ::=



parallel_clause: See the PARALLEL clause on page 4-465.

Keywords and Parameters

schema is the schema containing the table. If you omit *schema*, Oracle assumes the table is in your own schema.

table is the name of the table to be altered. You can alter the definition of an index-organized *table*.

ADD adds a column or integrity constraint. You cannot ADD columns to an index-organized table. See also "Adding Columns" on page 4-122.

MODIFY modifies the definition of an existing column. If you omit any of the optional parts of the column definition (datatype, default value, or column constraint), these parts remain unchanged.You cannot MODIFY column definitions of index-organized tables. See also "Modifying Column Definitions" on page 4-123.

- *column* is the name of the column to be added or modified.
- *datatype* specifies a datatype for a new column or a new datatype for an existing column.

  You can omit the datatype only if the statement also designates the column as part of the foreign key of a referential integrity constraint. Oracle automatically assigns the column the same datatype as the corresponding column of the referenced key of the referential integrity constraint.
- DEFAULT specifies a default value for a new column or a new default for an existing column. Oracle assigns this value to the column if a subsequent INSERT statement omits a value for the column. If you are adding a new column to the table and specify the default value, Oracle inserts the default column value into all rows of the table.

  The datatype of the default value must match the datatype specified for the column. The column must also be long enough to hold the default value. A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.
- *column_constraint* adds or removes a NOT NULL constraint to or from an existing column. See the syntax of *column_constraint* in the CONSTRAINT clause on page 4-188.
- *table_constraint* adds an integrity constraint to the table. See the syntax of *table_constraint* in the CONSTRAINT clause on page 4-188.

  See also "REFs" on page 4-127.

modify_default_attributes_clause is a valid option only for partitioned tables. Use this option to specify new values for the default attributes of a partitioned table.

physical_ attributes_clause changes the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters for the table, partition, the overflow data segment, or the default characteristics of a partitioned table. See the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters of CREATE TABLE on page 4-306.

- *storage_clause* changes the storage characteristics of the table, partition, overflow data segment, or the default characteristics of a partitioned table. See the STORAGE clause on page 4-523.

PCTTHRESHOLD specifies the percentage of space reserved in the index block for an index-organized table row. Any portion of the row that exceeds the specified threshold is stored in the overflow area. If OVERFLOW is not specified, then rows exceeding the THRESHOLD limit are rejected. PCTTHRESHOLD must be a value from 0 to 50.

- INCLUDING *column_name* specifies a column at which to divide an index-organized table row into index and overflow portions. All columns that follow *column_ name* are stored in the overflow data segment. A *column_name* is either the name of the last primary key column or any non-primary-key column.
- OVERFLOW specifies the overflow data segment physical storage attributes to be modified for the index-organized table. Parameters specified in this clause are only applicable to the overflow data segment. See CREATE TABLE on page 4-306.
- ADD OVERFLOW adds an overflow data segment to the specified index-organized table.

See also “Index-Organized Tables” on page 4-125.

LOB specifies the LOB storage characteristics for the newly added LOB column. You cannot use this clause to modify an existing LOB column.

lob_item is 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.

STORE AS

- *lob_segname* specifies the name of the LOB data segment. You cannot use *lob_ segname* if more than one *lob_item* is specified.
- ENABLE STORAGE IN ROW specifies that 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.
- DISABLE STORAGE IN ROW specifies that the LOB value is stored outside of the row regardless of the length of the LOB value.

Note that the LOB locator is always stored in the row regardless of where the LOB value is stored. You cannot change the STORAGE IN ROW once it is set.

CHUNK *integer* — specifies the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and integer is 2050, Oracle allocates 4096 bytes (2 blocks).The maximum value is 32768 (32 K), which is the largest Oracle block size allowed.

Note: The value of CHUNK must be less than or equal to the values of both INITIAL and NEXT (either the default values or those specified in the storage clause). If CHUNK exceeds the value of either INITIAL or NEXT, Oracle returns an error.

PCTVERSION *integer* — is the maximum percentage of overall LOB storage space used for creating new versions of the LOB. The default value is 10, meaning that older versions of the LOB data rae not overwritten until 10% of the overall LOB storage space is used.

INDEX *lob_index_name* — is the name of the LOB index segment. You cannot use *lob_index_name* if more than one *lob_item* is specified.

MODIFY LOB (*lob_item*) — modifies the physical attributes of the LOB attribute *lob_item* or LOB object attribute. You can only specify one LOB column for each MODIFY LOB clause.

See also "LOB Columns" on page 4-125.

OBJ NESTED TABLE *nested_item* STORE AS *storage_table*

specifies *storage_table* as the name of the storage table in which the rows of all *nested_item* values reside. You must include this clause when modifying a table with columns or column attributes whose type is a nested table.

The *nested_item* is the name of a column or a column-qualified attribute whose type is a nested table.

The *storage_table* is the name of the storage table. The storage table is modified in the same schema and the same tablespace as the parent table.

See also "Nested Table Columns" on page 4-126.

drop_clause — drops an integrity constraint. See the DROP clause on page 4-384.

ALLOCATE EXTENT — explicitly allocates a new extent for the table, the partition, the overflow data segment, the LOB data segment, or the LOB index.

SIZE — specifies the size of the extent in bytes. You can use K or M to specify the extent size in kilobytes or megabytes. If you omit this parameter, Oracle determines the size based on the values of the table's overflow data segment's, or LOB index's STORAGE parameters.

DATAFILE — specifies one of the datafiles in the tablespace of the table, overflow data segment, LOB data tablespace, or LOB index to contain the new extent. If you omit this parameter, Oracle chooses the datafile.

INSTANCE — 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, the former is divided by the latter, and the remainder is used 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, the space is allocated to the table, but is not drawn from any particular freelist group. Rather, the master freelist is used, and space is allocated as needed. For more information, see *Oracle8 Concepts*. Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

Explicitly allocating an extent with this clause does affect the size for the next extent to be allocated as specified by the NEXT and PCTINCREASE storage parameters.

DEALLOCATE UNUSED explicitly deallocates unused space at the end of the table, partition, overflow data segment, LOB data segment, or LOB index and makes the space available for other segments. You can free only unused space above the high-water mark. If KEEP is omitted, all unused space is freed. For more information, see DEALLOCATE UNUSED clause on page 4-372.

KEEP — specifies the number of bytes above the high-water mark that the table, overflow data segment, LOB data segment, or LOB index will have after deallocation. If the number of remaining extents are 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.

enable_clause enables a single integrity constraint or all triggers associated with the table. See the ENABLE clause on page 4-417.

CACHE for data that is accessed frequently, 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 option is useful for small lookup tables.

CACHE is not a valid option for index-organized tables.

NOCACHE for data that is not accessed frequently, specifies 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.

For LOBs, the LOB value is either not brought into the buffer cache or brought into the buffer cache and placed at the least recently used end of the LRU list. (The latter is the default behavior.)

	NOCACHE is not a valid option for index-organized tables.
LOGGING / NOLOGGING	LOGGING / NOLOGGING specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against a nonpartitioned table, table partition, or all partitions of a partitioned table will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. When used with the *modify_default_attributes_clause*, this option affects the logging attribute of a partitioned table. LOGGING / NOLOGGING also specifies whether ALTER TABLE...MOVE and ALTER TABLE...SPLIT operations will be logged or not logged. 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 logged. Therefore, if you cannot afford to lose this table, it is important to take a backup after the NOLOGGING operation. If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation will restore the table. However, media recovery from a backup taken before the NOLOGGING operation will not restore the table. The logging attribute of the base table is independent of that of its indexes. For more information about the LOGGING option and Parallel DML, see *Oracle8 Parallel Server Concepts and Administration*. NOLOGGING is not a valid keyword for altering index-organized tables.
RENAME TO	renames *table* to *new_table_name*.
partitioning_ clauses	See also "Modifying Table Partitions" on page 4-128.
MODIFY PARTITION [table partitions]	modifies the real physical attributes of a table partition *partition_name*. You can specify any of the following as new physical attributes for the partition: the logging attribute; PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameter; or storage parameters.
MODIFY PARTITION [index partitions]	modifies the attributes of an index partition partition_name. Note that you cannot specify the following options with clauses of the MODIFY PARTITION [table partitions] option.

UNUSABLE LOCAL INDEXES	marks all the local index partitions associated with *partition_name* as unusable.
REBUILD UNUSABLE LOCAL INDEXES	rebuilds the unusable local index partitions associated with *partition_ name*.

RENAME PARTITION	renames table partition *current_name* to *new_name*.
MOVE PARTITION	moves table partition *partition_name* to another segment. You can move partition data to another tablespace, recluster data to reduce fragmentation, or change a create-time physical attribute.
ADD PARTITION	adds a new partition *new_partition_name* to the "high" end of a partitioned table. You can specify any of the following as new physical attributes for the partition: the logging attribute; the PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameter; or storage parameters.
VALUES LESS THAN (*value_list*)	specifies the upper bound for the new partition. The *value_list* is a comma-separated, ordered list of literal values corresponding to *column_list*. The *value_list* must collate greater than the partition bound for the highest existing partition in the table.
DROP PARTITION	removes partition *partition_name*, and the data in that partition, from a partitioned table.
TRUNCATE PARTITION	removes all rows from the partition *partition_name* in a table.
DROP STORAGE	specifies that space from the deleted rows be deallocated and made available for use by other schema objects in the tablespace.
REUSE STORAGE	specifies that space from the deleted rows remains allocated to the partition. The space is subsequently available only for inserts and updates to the same partition.
SPLIT PARTITION	from an original partition *partition_name_old*, creates two new partitions, each with a new segment and new physical attributes, and new initial extents. The segment associated with *partition_name_old* is discarded.
AT (*value_list*)	specifies the new noninclusive upper bound for *split_partition_1*. The *value_list* must compare less than the pre-split partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one).
INTO	describes the two partitions resulting from the split.
partition_description, partition_description	specifies optimal names and physical attributes of the two partitions resulting from the split.
EXCHANGE PARTITION	converts partition *partition_name* into a nonpartitioned table, and a nonpartitioned table into a partition of a partitioned table by exchanging their data (and index) segments. The default behavior is EXCLUDING INDEXES WITH VALIDATION.

	WITH TABLE *table*	specifies the table with which the partition will be exchanged.
	INCLUDING INDEXES	specifies that the local index partitions be exchanged with the corresponding regular indexes.
	EXCLUDING INDEXES	specifies that all the local index partitions corresponding to the partition and all the regular indexes on the exchanged table are marked as unusable.
	WITH VALIDATION	specifies that any rows in the exchanged table that do not collate properly return an error.
	WITHOUT VALIDATION	specifies that the proper collation of rows in the exchanged table is not checked.
parallel_clause	specifies the degree of parallelism for the table. PARALLEL is not a valid option for index-organized tables. See the PARALLEL clause on page 4-465.	
	ENABLE TABLE LOCK	enables DML and DDL locks on a table in a parallel server environment. For more information, see *Oracle8 Parallel Server Concepts and Administration.*
	disable_clause	disables a single integrity constraint or all triggers associated with the tables. See the DISABLE clause on page 4-380. Integrity constraints specified in DISABLED clauses must be defined in the ALTER TABLE statements or in a previously issued statement. You can also enable and disable integrity constraints with the ENABLE and DISABLE keywords of the CONSTRAINT clause. If you define an integrity constraint but do not explicitly enable or disable it, Oracle enables it by default.
	DISABLE TABLE LOCK	disables DML and DDL locks on a table to improve performance in a parallel server environment. For more information, see *Oracle8 Parallel Server Concepts and Administration.*

Adding Columns

If you use the ADD clause to add a new column to the table, then the initial value of each row for the new column is null. You can add a column with a NOT NULL constraint only to a table that contains no rows.

If you create a view with a query that uses the asterisk (*) in the select list to select all columns from the base table and you subsequently add columns to the base table, Oracle will not automatically add the new column to the view. To add the new column to the view, you can re-create the view using the CREATE VIEW command with the OR REPLACE option.

Operations performed by the ALTER TABLE command can cause Oracle to invalidate procedures and stored functions that access the table. For information on how and when Oracle invalidates such objects, see *Oracle8 Concepts*.

Modifying Column Definitions

You can use the MODIFY clause to change any of the following parts of a column definition: datatype, size, default value, or NOT NULL column constraint.

The MODIFY clause need only specify the column name and the modified part of the definition, rather than the entire column definition.

Datatypes and Sizes

You can change a CHAR column to VARCHAR2 (or VARCHAR) and a VARCHAR2 (or VARCHAR) to CHAR only if the column contains nulls in all rows or if you do not attempt to change the column size. You can change any column's datatype or decrease any column's size if all rows for the column contain nulls. However, you can always increase the size of a character or raw column or the precision of a numeric column.

You cannot change a column's datatype to a LOB or REF datatype.

Default Values

A change to a column's default value only affects rows subsequently inserted into the table. Such a change does not change default values previously inserted.

To discontinue previously specified default values, so that they are no longer automatically inserted into newly added rows, replace the values with nulls, as shown in this example:

```
ALTER TABLE accounts
   MODIFY (bal DEFAULT NULL);
```

This statement has no effect on any existing values in existing rows.

Integrity Constraints

The only type of integrity constraint that you can add to an existing column using the MODIFY clause with the column constraint syntax is a NOT NULL constraint. However, you can define other types of integrity constraints (UNIQUE, PRIMARY KEY, referential integrity, and CHECK constraints) on existing columns using the ADD clause and the table constraint syntax.

You can define a NOT NULL constraint on an existing column only if the column contains no nulls.

Example I. The following statement adds a column named THRIFTPLAN of datatype NUMBER with a maximum of seven digits and two decimal places and a column named LOANCODE of datatype CHAR with a size of one and a NOT NULL integrity constraint:

```
ALTER TABLE emp
   ADD (thriftplan NUMBER(7,2),
        loancode CHAR(1) NOT NULL);
```

Example II. The following statement increases the size of the THRIFTPLAN column to nine digits:

```
ALTER TABLE emp
   MODIFY (thriftplan NUMBER(9,2));
```

Because the MODIFY clause contains only one column definition, the parentheses around the definition are optional.

Example III. The following statement changes the values of the PCTFREE and PCTUSED parameters for the EMP table to 30 and 60, respectively:

```
ALTER TABLE emp
    PCTFREE 30
    PCTUSED 60;
```

Example IV. The following statement allocates an extent of 5 kilobytes for the EMP table and makes it available to instance 4:

```
ALTER TABLE emp
  ALLOCATE EXTENT (SIZE 5K INSTANCE 4);
```

Because this command omits the DATAFILE parameter, Oracle allocates the extent in one of the datafiles belonging to the tablespace containing the table.

Example V. This example modifies the BAL column of the ACCOUNTS table so that it has a default value of 0:

```
ALTER TABLE accounts
  MODIFY (bal  DEFAULT 0);
```

If you subsequently add a new row to the ACCOUNTS table and do not specify a value for the BAL column, the value of the BAL column is automatically 0:

```
INSERT INTO accounts(accno, accname)
  VALUES (accseq.nextval, 'LEWIS')
  SELECT *
    FROM accounts
    WHERE accname = 'LEWIS';

ACCNO  ACCNAME BAL
------ ------- ---
815234  LEWIS     0
```

Index-Organized Tables

Index-organized tables are special kinds of tables that keep data sorted on the primary key and are therefore best suited for primary-key-based access and manipulation.

You cannot ADD columns to an index-organized table, but you can alter the definition of an index-organized table.

Example I. This example modifies the INITRANS parameter for the index segment of index-organized table DOCINDEX:

```
ALTER TABLE docindex INITRANS 4;
```

Example II. The following statement adds an overflow data segment to index-organized table DOCINDEX:

```
ALTER TABLE docindex ADD OVERFLOW;
```

Example III. This example modifies the INITRANS parameter for the overflow data segment of index-organized table DOCINDEX:

```
ALTER TABLE docindex OVERFLOW INITRANS 4;
```

LOB Columns

You can add a LOB column to a table, or modify the LOB data segment or index storage characteristics.

Example I. The following statement adds CLOB column RESUME to the EMPLOYEE table:

```
ALTER TABLE employee ADD (resume CLOB)
 LOB (resume) STORE AS resume_seg (TABLESPACE resume_ts);
```

Example II. To modify the LOB column RESUME to use caching, enter the following statement:

```
ALTER TABLE employee MODIFY LOB (resume) (CACHE);
```

OBJ Nested Table Columns

You can add a nested table type column to a table. Specify a nested table storage clause for each column added.

Example I. The following example adds the nested table column SKILLS to the EMPLOYEE table:

```
ALTER TABLE employee ADD (skills skill_table_type)
    NESTED TABLE skills STORE AS nested_skill_table;
```

You can also modify a nested table's storage characteristics. Use the name of the storage table specified in the nested table storage clause to make the modification. You *cannot* query or perform DML statements on the storage table; only use the storage table to modify the nested table column storage characteristics.

Example II. OBJ The following example creates table VETSERVICE with nested table column CLIENT and storage table CLIENT_TAB. Nested table VETSERVICE is modified to specify constraints and modify a column length by altering nested storage table CLIENT_TAB:

```
CREATE TABLE vetservice (vet_name VARCHAR2(30),
                         client   pet_table)
  NESTED TABLE client STORE AS client_tab;
ALTER TABLE client_tab ADD UNIQUE (ssn);
ALTER TABLE client_tab MODIFY (pet_name VARCHAR2(35));
```

Example IV. OBJ The following statement adds a UNIQUE constraint to nested table NESTED_SKILL_TABLE:

```
ALTER TABLE nested_skill_table ADD UNIQUE (a);
```

For more information about nested table storage see the CREATE TABLE on page 4-306. For more information about nested tables, see *Oracle8 Application Developer's Guide*.

Example V. The following example 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);
```

Note that in order to execute these ALTER TABLE statements successfully, the storage table DEPTEMPS must be empty. Also, note that because the nested table is defined as a table of scalars (REFs), Oracle implicitly provides the column name COLUMN_VALUE for the storage table.

OBJ REFs

A REF value is a reference to a row in an object table. A table can have top-level REF columns or it can have REF attributes embedded within an object column. In general, if a table has a REF column, each REF value in the column could reference a row in a different object table. A SCOPE clause restricts the scope of references to a single table.

Use the ALTER TABLE command to add new REF columns or to add REF clauses to existing REF columns. You can modify any table, including named inner nested tables (storage tables). If a REF column is created WITH ROWID or with a scope table, you cannot modify the column to drop these options. However, if a table is created without any REF clauses, you can add them later with an ALTER TABLE statement.

Note: You can add a scope clause to existing REF columns of a table only if the table is empty. The *scope_table_name* must be in your own schema or you must have SELECT privilege on the table, or the SELECT ANY TABLE system privilege. This privilege is needed only while altering the table with the REF column.

Example I. In the following example an object type DEPT_T has been previously defined. Now, create table EMP as follows:

```
CREATE TABLE emp
   (name VARCHAR(100),
    salary NUMBER,
    dept REF dept_t);
```

An object table DEPARTMENTS is created as:

```
CREATE TABLE departments OF dept_t;
```

If the DEPARTMENTS table contains all possible departments, the DEPT column in EMP can only refer to rows in the DEPARTMENTS table. This can be expressed as a scope clause on the DEPT column as follows:

```
ALTER TABLE emp
    ADD (SCOPE FOR (dept) IS departments);
```

Note that the above ALTER TABLE statement will succeed *only if* the EMP table is empty.

Example II. If you want the REF values in the DEPT column of EMP to also store the ROWIDs, issue the following statement:

```
ALTER TABLE emp
 ADD (REF(dept) WITH ROWID);
```

Modifying Table Partitions

You can modify a table or table partition in any of the following ways. You cannot combine partition operations with other partition operations or with operations on the base table in one ALTER TABLE statement.

ADD PARTITION

Use ALTER TABLE ADD PARTITION to add a partition to the high end of the table (after the last existing partition). If the first element of the partition bound of the high partition is MAXVALUE, you cannot add a partition to the table. You must split the high partition.

You can add a partition to a table even if one or more of the table indexes or index partitions are marked UNUSABLE.

You must use the SPLIT PARTITION clause to add a partition at the beginning or the middle of the table.

The following example adds partition JAN97 to tablespace TSX:

```
ALTER TABLE sales
  ADD PARTITION jan97 VALUES LESS THAN( '970201' )
  TABLESPACE tsx;
```

DROP PARTITION

ALTER TABLE DROP PARTITION drops a partition and its data. If you want to drop a partition but keep its data in the table, you must merge the partition into

one of the adjacent partitions. For information about merging two tables partitions, see the *Oracle8 Administrator's Guide*.

If you drop a partition and later insert a row that would have belonged to the dropped partition, the row will be stored in the next higher partition. However, if you drop the highest partition, the insert will fail because the range of values represented by the dropped partition is no longer valid for the table.

This statement also drops the corresponding partition in each local index defined on *table*. The index partitions are dropped even if they are marked as unusable.

If there are global indexes defined on *table*, and the partition you want to drop is *not* empty, dropping the partition marks all the global, nonpartitioned indexes and all the partitions of global partitioned indexes as unusable.

When a table contains only one partition, you cannot drop the partition. You must drop the table.

The following example drops partition DEC95:

```
ALTER TABLE sales DROP PARTITION dec95;
```

EXCHANGE PARTITION

This form of ALTER TABLE converts a partition to a nonpartitioned table and a NONPARTITIONED table to a partition by exchanging their data segments. You must have ALTER TABLE privileges on both tables to perform this operation.

The statistics of the table and partition—including table, column, index statistics and histograms—are exchanged. The aggregate statistics of the partitioned table are recalculated.

The logging attribute of the table and partition is exchanged.

The following example converts partition FEB97 to table SALES_FEB97 without exchanging local index partitions with corresponding indexes on SALES_FEB97 and without verifying that data in SALES_FEB97 falls within the bounds of partition FEB97:

```
ALTER TABLE sales
  EXCHANGE PARTITION feb97 WITH TABLE sales_feb97
   WITHOUT VALIDATION;
```

MODIFY PARTITION

Use the MODIFY PARTITION options of ALTER TABLE to

- mark local index partitions corresponding to a table partition as unusable

- rebuild all the unusable local index partitions corresponding to a table partition
- modify the physical attributes of a table partition

The following example marks all the local index partitions corresponding to the NOV96 partition of the SALES table UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION nov96
  UNUSABLE LOCAL INDEXES;
```

The following example rebuilds all the local index partitions that were marked UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION jan97
  REBUILD UNUSABLE LOCAL INDEXES;
```

The following example changes MAXEXTENTS and logging attribute for partition BRANCH_NY:

```
ALTER TABLE branch MODIFY PARTITION branch_ny
  STORAGE(MAXEXTENTS 75) LOGGING;
```

MOVE PARTITION

This ALTER TABLE option moves a table partition to another segment. MOVE PARTITION always drops the partition's old segment and creates a new segment, even if you do not specify a new tablespace.

If partition *partition_name* is not empty, MOVE PARTITION marks all corresponding local index partitions and all global nonpartitioned indexes, and all the partitions of global partitioned indexes as unusable.

ALTER TABLE MOVE PARTITION obtains its parallel attribute from the PARALLEL clause, if specified. If not specified, the default PARALLEL attributes of the table, if any, are used. If neither is specified, it performs the move without using parallelism.

The PARALLEL clause on MOVE PARTITION does not change the default PARALLEL attributes of *table*.

The following example moves partition DEPOT2 to tablespace TS094:

```
ALTER TABLE parts
  MOVE PARTITION depot2 TABLESPACE ts094 NOLOGGING;
```

RENAME

Use the RENAME option of ALTER TABLE to rename a table or to rename a partition.

The following example renames a table:

```
ALTER TABLE emp RENAME TO employee;
```

In the following example, partition EMP3 is renamed:

```
ALTER TABLE employee RENAME PARTITION emp3 TO employee3;
```

SPLIT PARTITION

The SPLIT PARTITION option divides a partition into two partitions. A new segment is allocated for each partition resulting from the split. The attributes of the new partitions are inherited from the partition that was split, except for attributes whose values you explicitly override in the SPLIT clause. The segment associated with the old partition is discarded.

This statement also performs a matching split on the corresponding partition in each local index defined on *table*. The index partitions are split even if they are marked unusable.

With the exception of the TABLESPACE attribute, the physical attributes of the LOCAL index partition being split are used for both new index partitions. If the parent LOCAL index lacks a default TABLESPACE attribute, new LOCAL index partitions will reside in the same tablespace as the corresponding newly created partitions of the underlying table.

If you do not specify physical attributes (PCTFREE, PCTUSED, INITRANS, MAXTRANS, STORAGE) for the new partitions, the current values of the partition being split are used as the default values for both partitions.

If *partition_name* is not empty, SPLIT PARTITION marks all affected index partitions as unusable. This includes all global index partitions as well as the local index partitions that result from the split.

The PARALLEL clause on SPLIT PARTITION does not change the default PARALLEL attributes of *table*.

The following example splits the old partition DEPOT4, creating two new partitions, naming one DEPOT9 and reusing the name of the old partition for the other:

```
ALTER TABLE parts
  SPLIT PARTITION depot4 AT ( '40-001' )
  INTO ( PARTITION depot4 TABLESPACE ts009 (MINEXTENTS 2),
         PARTITION depot9 TABLESPACE ts010 )
  PARALLEL ( DEGREE 10 );
```

TRUNCATE PARTITION

Use TRUNCATE PARTITION to remove all rows from a partition in a table. Freed space is deallocated or reused depending on whether DROP STORAGE or REUSE STORAGE is specified in the clause.

This statement truncates the corresponding partition in each local index defined on *table*. The local index partitions are truncated even if they are marked as unusable. The unusable local index partitions are marked valid, resetting the UNUSABLE indicator.

If any global indexes are defined on *table*, and the partition you want to truncate is *not* empty, truncating the partition marks all the global nonpartitioned indexes and all the partitions of global partitioned indexes as unusable.

If you want to truncate a partition that contains data, you must first disable any referential integrity constraints on the table. Alternatively, you can delete the rows and then truncate the partition.

The following example deletes all the data in the SYS_P017 partition and deallocates the freed space:

```
ALTER TABLE deliveries
  TRUNCATE PARTITION sys_p017 DROP STORAGE;
```

For examples of defining integrity constraints with the ALTER TABLE command, see the CONSTRAINT clause on page 4-188.

For examples of enabling, disabling, and dropping integrity constraints and triggers with the ALTER TABLE command, see the ENABLE clause on page 4-417, the DISABLE clause on page 4-380, and the DROP clause on page 4-384.

For examples of changing the value of a table's storage parameters, see the STORAGE clause on page 4-523.

Related Topics

CREATE TABLE on page 4-306
CONSTRAINT clause on page 4-188
DISABLE clause on page 4-380
DISABLE clause on page 4-380
ENABLE clause on page 4-417
STORAGE clause on page 4-523
CREATE VIEW on page 4-363

ALTER TABLESPACE

Purpose

To alter an existing tablespace in one of the following ways:

- add datafile(s)
- rename datafiles
- change default storage parameters
- take the tablespace online or offline
- begin or end a backup
- allow or disallow writing to a tablespace
- change the default logging attribute of the tablespace
- change the minimum tablespace extent length

See also "Using ALTER TABLESPACE" on page 4-138.

Prerequisites

If you have ALTER TABLESPACE system privilege, you can perform any of this command's operations. If you have MANAGE TABLESPACE system privilege, you can only perform the following operations:

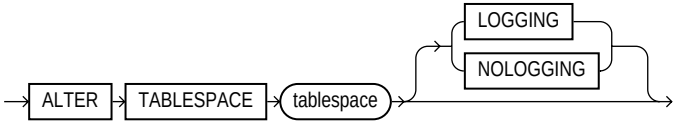
- take the tablespace online or offline
- begin or end a backup
- make the tablespace read-only or read-write

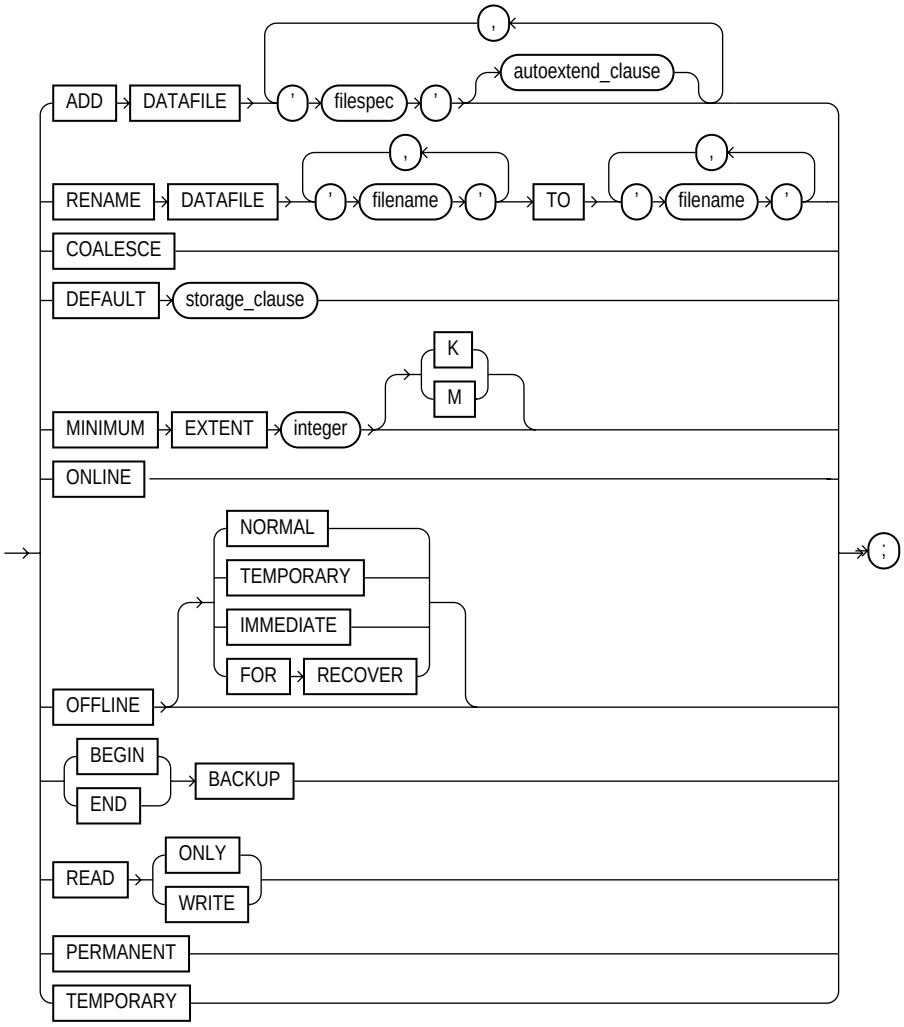
Before you can make a tablespace read-only, the following conditions must be met. Performing this function in restricted mode may help you meet these restrictions, since only users with RESTRICTED SESSION system privilege can be logged on.

- The tablespace must be online.
- There must not be any active transactions in the entire database. This is necessary to ensure that no undo information needs to be applied to the tablespace.
- 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.

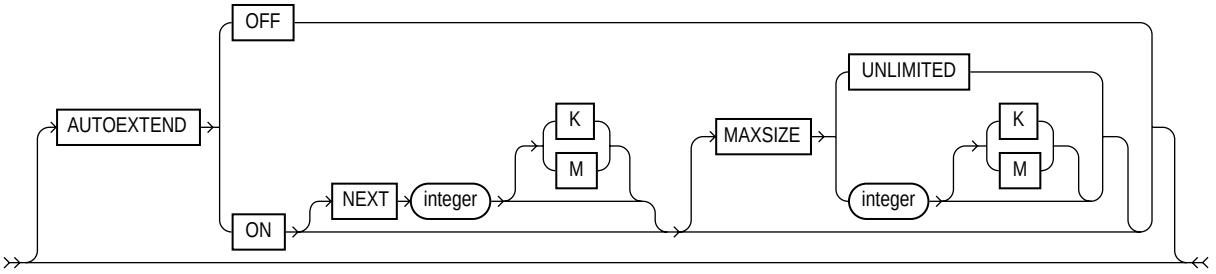
- The tablespace must not be involved in an open backup, because the end of a backup updates the header file of all datafiles in the tablespace.
- The COMPATIBLE initialization parameter must be set to 7.1.0 or greater.

Syntax




,
ADD
DATAFILE
'
filespec
'
autoextend_clause
,
RENAME
DATAFILE
'
filename
'
TO
'
filename
'
COALESCE
DEFAULT
storage_clause
K
M
MINIMUM
EXTENT
integer
ONLINE
NORMAL
TEMPORARY
IMMEDIATE
FOR
RECOVER
OFFLINE
BEGIN
END
BACKUP
READ
ONLY
WRITE
PERMANENT
TEMPORARY
;

autoextend_clause ::=



filespec: See "Filespec" on page 4-431.

storage_clause: See STORAGE clause on page 4-523.

Keywords and Parameters

tablespace is the name of the tablespace to be altered.

LOGGING / NOLOGGING specifies the default logging attribute 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 attributes of existing objects are not changed.

Only the following operations support NOLOGGING mode:

- DML: direct-load INSERT (serial or parallel); Direct Loader (SQL*Loader)
- DDL: CREATE TABLE... AS SELECT, CREATE INDEX, ALTER INDEX... REBUILD, ALTER INDEX... REBUILD PARTITION, ALTER INDEX... SPLIT PARTITION, ALTER TABLE... SPLIT PARTITION, ALTER TABLE... MOVE PARTITION.

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 logged. Therefore, if you cannot afford to lose the object, it is important to take a backup after the NOLOGGING operation.

ADD DATAFILE adds the datafile specified by *filespec* to the tablespace. (See the syntax description of Filespec on page 4-431*)*. You can add a datafile while the tablespace is online or offline. Be sure that the datafile is not already in use by another database.

AUTOEXTEND	enables or disables the autoextending of the size of the datafile in the tablespace.	
	OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND commands.
	ON	enables autoextend.
	NEXT	specifies the size in bytes of the next increment of disk space to be allocated automatically to the datafile when more extents are required. You can use K or M to specify this size in kilobytes or megabytes. The default is one data block.
	MAXSIZE	specifies maximum disk space allowed for automatic extension of the datafile.
	UNLIMITED	sets no limit on allocating disk space to the datafile.
RENAME DATAFILE	renames one or more of the tablespace's datafiles. Take the tablespace offline before renaming the datafile. Each '*filename*' must fully specify a datafile using the conventions for filenames on your operating system.	
	This clause only 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.	
COALESCE	for each datafile in the tablespace, coalesces all contiguous free extents into larger contiguous extents.	
	COALESCE cannot be specified with any other command option.	
DEFAULT *storage_clause*	specifies the new default storage parameters for objects subsequently created in the tablespace. See the STORAGE clause on page 4-523.	
MINIMUM EXTENT *integer*	controls free space fragmentation in the tablespace by ensuring that every used and / or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*. For more information about using MINIMUM EXTENT to control space fragmentation, see *Oracle8 Administrator's Guide*	
ONLINE	brings the tablespace online.	
OFFLINE	takes the tablespace offline and prevents further access to its segments.	
	NORMAL	performs a checkpoint for all datafiles in the tablespace. All of these datafiles must be online. This is the default. You need not perform media recovery on this tablespace before bringing it back online. You must use this option if the database is in NOARCHIVELOG mode.
	TEMPORARY	performs a checkpoint for all online datafiles in the tablespace but does not ensure that all files can be written. Any offline files may require media recovery before you bring the tablespace back online.

IMMEDIATE	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.
FOR RECOVER	takes the production database tablespaces in the recovery set offline. Use this option when one or more datafiles in the tablespace are unavailable.

Suggestion: Before taking a tablespace offline for a long time, you may want to alter any users who have been assigned the tablespace as either a default or temporary tablespace. When the tablespace is offline, these users cannot allocate space for objects or sort areas in the tablespace. You can reassign to such users new default and temporary tablespaces with the ALTER USER command.

BEGIN BACKUP	signifies that an open backup is to be performed on the datafiles that make up this tablespace. This option does not prevent users from accessing the tablespace. You must use this option before beginning an open backup. You cannot use this option on a read-only tablespace. **Note:** While the backup is in progress, you cannot: take the tablespace offline normally, shutdown the instance, or begin another backup of the tablespace.
END BACKUP	signifies that an open backup of the tablespace is complete. Use this option as soon as possible after completing an open backup. You cannot use this option on a read-only tablespace. If you forget to indicate the end of an online tablespace backup, and an instance failure or SHUTDOWN ABORT occurs, Oracle assumes that media recovery (possibly requiring archived redo log) is necessary at the next instance start up. To restart the database without media recovery, see *Oracle8 Administrator's Guide*.
READ ONLY	signifies that no further write operations are allowed on the tablespace. The tablespace becomes read only. Once a tablespace is read-only, you can copy its files to read-only media. You must then rename the datafiles in the control file to point to the new location by using the SQL command ALTER DATABASE RENAME.
READ WRITE	signifies that write operations are allowed on a previously read-only tablespace.
PERMANENT	specifies that the tablespace is to be converted from a temporary to a permanent one. A permanent tablespace is one wherein permanent database objects can be stored. This is the default when a tablespace is created.
TEMPORARY	specifies that the tablespace is to be converted from a permanent to a temporary one. A temporary tablespace is one in which no permanent database objects can be stored.

Using ALTER TABLESPACE

The following examples illustrate the use of the ALTER TABLESPACE COMMAND.

Example I. The following statement signals to the database that a backup is about to begin:

```
ALTER TABLESPACE accounting
    BEGIN BACKUP;
```

Example II. The following statement signals to the database that the backup is finished:

```
ALTER TABLESPACE accounting
    END BACKUP;
```

Example III. This example moves and renames a datafile associated with the ACCOUNTING tablespace from 'DISKA:PAY1.DAT' to 'DISKB:RECEIVE1.DAT':

1. Take the tablespace offline using an ALTER TABLESPACE statement with the OFFLINE option:

   ```
   ALTER TABLESPACE accounting OFFLINE NORMAL;
   ```

2. Copy the file from 'DISKA:PAY1.DAT' to 'DISKB:RECEIVE1.DAT' using your operating system's commands.

3. Rename the datafile using the ALTER TABLESPACE command with the RENAME DATAFILE clause:

   ```
   ALTER TABLESPACE accounting
     RENAME DATAFILE 'diska:pay1.dbf'
     TO     'diskb:receive1.dbf';
   ```

4. Bring the tablespace back online using an ALTER TABLESPACE statement with the ONLINE option:

   ```
   ALTER TABLESPACE accounting ONLINE;
   ```

Example IV. The following statement adds a datafile to the tablespace and changes the default logging attribute to NOLOGGING; when more space is needed new extents of size 10 kilobytes will be added up to a maximum of 100 kilobytes:

```
ALTER TABLESPACE accounting NOLOGGING
    ADD DATAFILE 'disk3:pay3.dbf'
    AUTOEXTEND ON
    NEXT 10 K
    MAXSIZE 100 K;
```

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.

Example V. The following statement changes the allocation of every extent of TABSPACE_ST to a multiple of 128K:

```
ALTER TABLESPACE tabspace_st MINIMUM EXTENT 128K;
```

Related Topics

CREATE TABLESPACE on page 4-328
CREATE DATABASE on page 4-219
DROP TABLESPACE on page 4-407
STORAGE clause on page 4-523
“Filespec” on page 4-431

ALTER TRIGGER

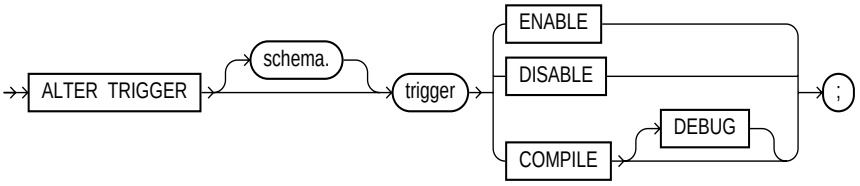
Purpose

To enable, disable, or compile a database trigger.

Prerequisites

The trigger must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.
trigger	is the name of the trigger to be altered. See also "Invalid Triggers" on page 4-141.
ENABLE	enables the trigger. See also "Enabling and Disabling Triggers" on page 4-142.
DISABLE	disables the trigger. See also "Enabling and Disabling Triggers" on page 4-142.
COMPILE	compiles the trigger.
DEBUG	instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. This option can be used for normal triggers and for instead-of triggers.

Invalid Triggers

You can use the ALTER TRIGGER command to explicitly recompile a trigger that is invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

When you issue an ALTER TRIGGER statement, Oracle recompiles the trigger regardless of whether it is valid or invalid.

When you recompile a trigger, Oracle first recompiles objects upon which the trigger depends, if any of these objects are invalid. If Oracle recompiles the trigger successfully, the trigger becomes valid. If recompiling the trigger results in compilation errors, then Oracle returns an error and the trigger remains invalid. You can then debug triggers using the predefined package DBMS_OUTPUT. For information on debugging procedures, see *Oracle8 Application Developer's Guide*. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8 Concepts*.

Note: This command does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, you must use the CREATE TRIGGER command with the OR REPLACE option.

Enabling and Disabling Triggers

A database trigger is always either enabled or disabled. If a trigger is enabled, Oracle fires the trigger when a triggering statement is issued. If the trigger is disabled, Oracle does not fire the trigger when a triggering statement is issued.

When you create a trigger, Oracle enables it automatically. You can use the ENABLE and DISABLE options of the ALTER TRIGGER command to enable and disable a trigger.

You can also use the ENABLE and DISABLE clauses of the ALTER TABLE command to enable and disable all triggers associated with a table.

Note: The ALTER TRIGGER command does not change the definition of an existing trigger. To redefine a trigger, you must use the CREATE TRIGGER command with the OR REPLACE option.

Example. Consider a trigger named REORDER created on the INVENTORY table. The trigger is fired whenever an UPDATE statement reduces the number of a particular part on hand below the part's reorder point. The trigger inserts into a table of pending orders a row that contains the part number, a reorder quantity, and the current date.

When this trigger is created, Oracle enables it automatically. You can subsequently disable the trigger with the following statement:

```
ALTER TRIGGER reorder
    DISABLE;
```

When the trigger is disabled, Oracle does not fire the trigger when an UPDATE statement causes the part's inventory to fall below its reorder point.

After disabling the trigger, you can subsequently enable it with the following statement:

```
ALTER TRIGGER reorder
    ENABLE;
```

After you reenable the trigger, Oracle fires the trigger whenever a part's inventory falls below its reorder point as a result of an UPDATE statement. Note that a part's inventory may have fallen below its reorder point while the trigger was disabled. When you reenable the trigger, Oracle does not automatically fire the trigger for this part until another transaction further reduces the inventory.

Related Topics

CREATE TRIGGER on page 4-333
DROP TRIGGER on page 4-409
DISABLE clause on page 4-380
ENABLE clause on page 4-417
ALTER TABLE on page 4-106

OBJ ALTER TYPE

Purpose

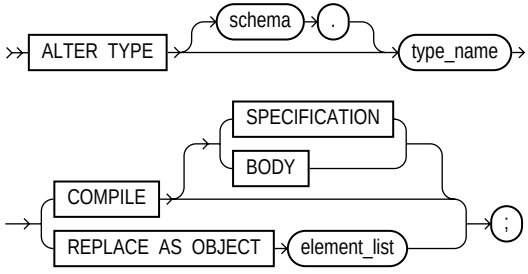
To recompile the specification and / or body, or to change the specification of an object type by adding new object member subprogram specifications.

> **Note:** This command is available only if the Oracle objects option is installed on your database server.

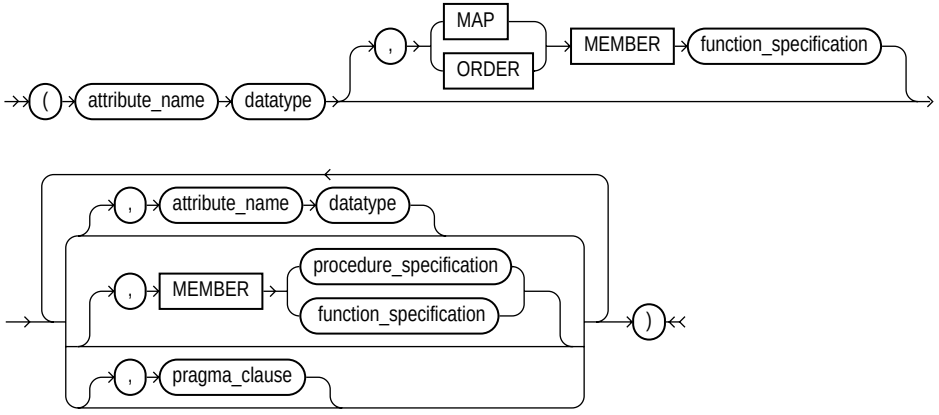
Prerequisites

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.

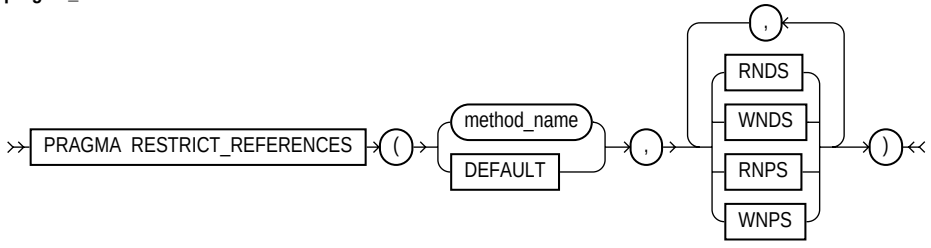
Syntax



element_list ::=



pragma_clause ::=



Keywords and Parameters

schema	is the schema that contains the type. If you omit *schema*, Oracle creates the type in your current schema.
type_name	is the name of an object type, a nested table type, or a VARRAY type.
COMPILE	compiles the object type specification and body. This is the default if no option is specified.
	SPECIFICATION compiles only the object type specification.
	BODY compiles only the object type body.
REPLACE AS OBJECT	adds new member subprogram specifications. This option is valid only for object types.
attribute_name	is an object attribute name. Attributes are data items with a name and a type specifier that form the structure of the object.

MAP / ORDER MEMBER *function_specification*

MAP — specifies a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined *scalar* type. PL / SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

A *scalar* value is always manipulated as a single unit. Scalars are mapped directly to the underlying hardware. An integer, for example, occupies 4 or 8 contiguous bytes of storage, in memory or on disk.

An object specification can contain only one map method, which must be a function. The result type must be a predefined SQL scalar type, and the map function can have no arguments other than the implicit SELF argument.

ORDER — specifies a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, zero, or positive indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

When instances of the same object type definition are compared in an ORDER BY clause, the order method *function_specification* is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type INTEGER.

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare object instances only for equality or inequality. Note that instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types. For more information about object value comparisons, "Object Values" on page 2-27.

MEMBER — specifies a function or procedure subprogram associated with the object type which is referenced as an attribute. For information about overloading subprogram names within a package, see the *PL/SQL User's Guide and Reference*. See also "Restriction" on page 4-147.

You must specify a corresponding method body in the object type body for each procedure or function specification. See CREATE TYPE BODY on page 4-353.

procedure_specification — is the specification of a procedure subprogram.

	function_specification	is the specification of a function subprogram.
PRAGMA RESTRICT_REFERENCES	is a complier directive that denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects. For more information, see the *PL/SQL User's Guide and Reference*.	
	method_name	is the name of the MEMBER function or procedure to which the pragma is being applied.
	WNDS	specifies constraint *writes no database state* (does not modify database tables).
	WNPS	specifies constraint *writes no package state* (does not modify packaged variables).
	RNDS	specifies constraint *reads no database state* (does not query database tables).
	RNPS	specifies constraint *reads no package state* (does not reference packages variables).

Restriction

You cannot change the existing properties (attributes, member subprograms, map or order functions) of an object type, but you can add new member subprogram specifications.

Example I. In the following example, member function QTR is added to the type definition of DATA_T:

```
CREATE TYPE data_t AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );

   CREATE TYPE BODY data_t IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
        BEGIN
           RETURN (year + invent);
        END;
      END;

   ALTER TYPE data_t REPLACE AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION  prod(invent NUMBER) RETURN NUMBER,
     MEMBER FUNCTION  qtr(der_qtr DATE) RETURN CHAR
```

```
   );

   CREATE OR REPLACE TYPE BODY data_t IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
        BEGIN
           RETURN (year + invent);
        END;
      MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
        BEGIN
           RETURN 'FIRST';
        END;
      END;
```

Example II. The following example recompiles type LOAN_T:

```
CREATE TYPE loan_t AS OBJECT
  ( loan_num        INTEGER,
    interest_rate   FLOAT,
    amount          FLOAT,
    start_date      DATE,
    end_date        DATE );

ALTER TYPE loan_t COMPILE;
```

Example III. The following example compiles the type body of LINK2:

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);

CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);

CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS t13 link1;
     BEGIN t13 := link1(13);
      dbms_output.put_line(t13.a);
        RETURN 5;
     END;
END;

CREATE TYPE link3 AS OBJECT (a link2);
CREATE TYPE link4 AS OBJECT (a link3);
CREATE TYPE link5 AS OBJECT (a link4);
ALTER TYPE link2 COMPILE BODY;
```

Example IV. The following example compiles the type specification of LINK2:

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);

CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);

CREATE TYPE BODY link2 AS
    MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS t14 link1;
      BEGIN t14 := link1(14);
      dbms_output.put_line(t14.a);
       RETURN 5;
      END;
END;

CREATE TYPE link3 AS OBJECT (a link2);
CREATE TYPE link4 AS OBJECT (a link3);
CREATE TYPE link5 AS OBJECT (a link4);
ALTER TYPE link2 COMPILE SPECIFICATION;
```

Related Topics

CREATE TYPE on page 4-345
CREATE TYPE BODY on page 4-353
PL/SQL User's Guide and Reference
Oracle8 Application Developer's Guide

ALTER USER

Purpose

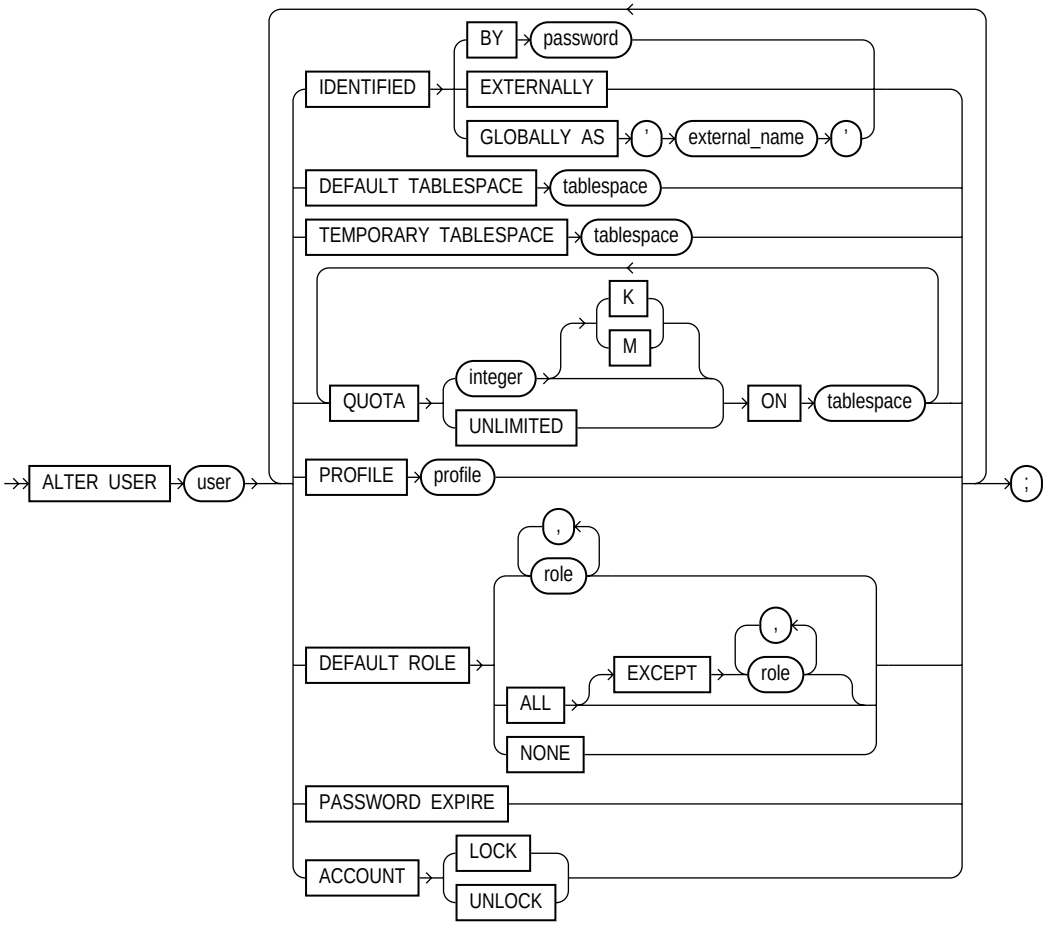
To change any of the following characteristics of a database user:

- authentication mechanism of the user
- password
- default tablespace for object creation
- tablespace for temporary segments created for the user
- tablespace access and tablespace quotas
- limits on database resources
- default roles

Prerequisites

You must have the ALTER USER system privilege. However, you can change your own password without this privilege.

Syntax



Keywords and Parameters

The keywords and parameters in the ALTER USER command all have the same meaning as in the CREATE USER command. For information on these keywords and parameters, see CREATE USER on page 4-357.

For more information on default roles, see “Establishing Default Roles” on page 4-152. For more information on security domains, see “Changing Authentication Methods” on page 4-152.

Establishing Default Roles

The DEFAULT ROLE clause can only contain roles that have been granted directly to the user with a GRANT statement. You cannot use the DEFAULT ROLE clause to enable:

- 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 Security Service Certification Authority

Note that Oracle enables default roles at logon without requiring the user to specify their passwords.

Example I. The following statement changes the user SCOTT's password to LION and default tablespace to the tablespace TSTEST:

```
ALTER USER scott
    IDENTIFIED BY lion
    DEFAULT TABLESPACE tstest;
```

Example II. The following statement assigns the CLERK profile to SCOTT:

```
ALTER USER scott
    PROFILE clerk;
```

In subsequent sessions, SCOTT is restricted by limits in the CLERK profile.

Example III. The following statement makes all roles granted directly to SCOTT default roles, except the AGENT role:

```
ALTER USER scott
    DEFAULT ROLE ALL EXCEPT agent;
```

At the beginning of SCOTT's next session, Oracle enables all roles granted directly to SCOTT except the AGENT role.

Changing Authentication Methods

You can change a user's access verification method to IDENTIFIED GLOBALLY AS '*external_name*' only if all external roles granted directly to the user are revoked.

You can change a user created as IDENTIFIED GLOBALLY AS '*external_name*' to IDENTIFIED BY *password* or IDENTIFIED EXTERNALLY.

Example I The following example changes user TOM's authentication mechanism:

```
ALTER USER tom IDENTIFIED GLOBALLY AS 'CN=tom';
```

Example II The following example causes user FRED's password to expire:

```
ALTER USER fred PASSWORD EXPIRE;
```

If you cause a database user's password to expire with PASSWORD EXPIRE, the user must change the password before attempting to log in to the database following the expiration. However, tools such as SQL*Plus allow you to change the password on the first attempted login following the expiration.

Related Topics

CREATE PROFILE on page 4-265
CREATE ROLE on page 4-272
CREATE USER on page 4-357
CREATE TABLESPACE on page 4-328

ALTER VIEW

Purpose

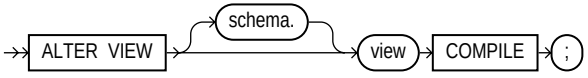
To recompile a view or an object view. See also "Recompiling Views" on page 4-154.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

The view must be in your own schema or you must have ALTER ANY TABLE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the view. If you omit *schema*, Oracle assumes the view is in your own schema.
view	is the name of the view to be recompiled.
COMPILE	causes Oracle to recompile the view. The COMPILE keyword is required.

Recompiling Views

You can use the ALTER VIEW command to explicitly recompile a view that is invalid. Explicit recompilation allows you to locate recompilation errors before run time. You may want to explicitly recompile a view after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it.

When you issue an ALTER VIEW statement, Oracle recompiles the view regardless of whether it is valid or invalid. Oracle also invalidates any local objects that

depend on the view. For more about dependencies among schema objects, see *Oracle8 Concepts*.

> **Note:** This command does not change the definition of an existing view. To redefine a view, you must use the CREATE VIEW command with the OR REPLACE option.

Example. To recompile the view CUSTOMER_VIEW, issue the following statement:

```
ALTER VIEW customer_view
    COMPILE;
```

If Oracle encounters no compilation errors while recompiling CUSTOMER_VIEW, CUSTOMER_VIEW becomes valid. If recompiling results in compilation errors, Oracle returns an error and CUSTOMER_VIEW remains invalid.

Oracle also invalidates all dependent objects. These objects include any procedures, functions, package bodies, and views that reference CUSTOMER_VIEW. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

Related Topics

CREATE VIEW on page 4-363

ANALYZE

Purpose

To perform one of the following functions on an index or index partition, table or table partition, index-organized table, or cluster:

- collect statistics about the schema object used by the optimizer and store them in the data dictionary
- delete statistics about the schema object from the data dictionary
- validate the structure of the schema object
- identify migrated and chained rows of the table or cluster
- OBJ collect statistics on scalar object attributes
- OBJ validate and update object references (REFs)

> **Note:** Descriptions of commands and clauses preceded by OBJ are only available if the Oracle objects option is installed on your database server.

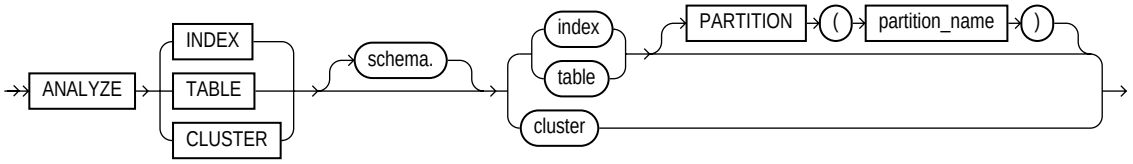
Prerequisites

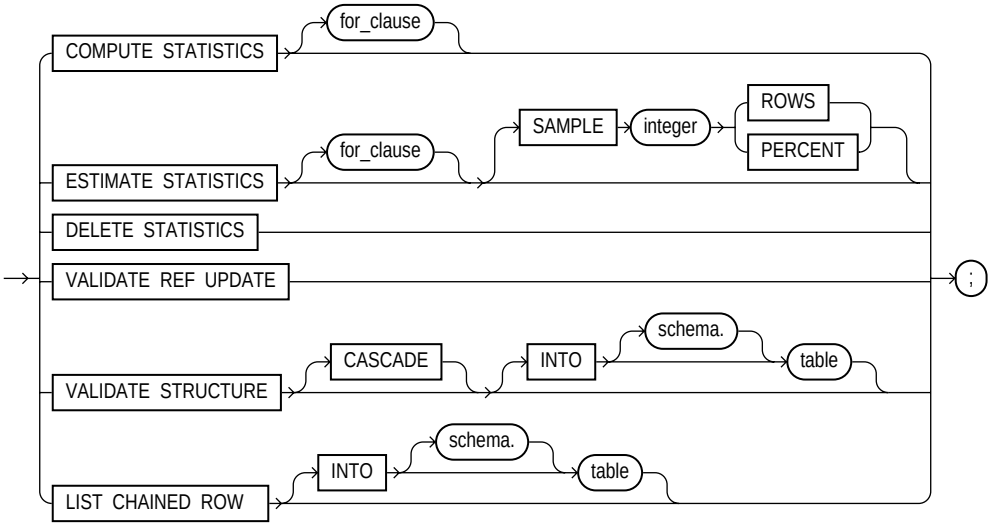
The schema object to be analyzed must be in your own schema or you must have the ANALYZE ANY system privilege.

If you want to list chained rows of a table or cluster into a list table, 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, you must have INSERT privilege on the table into which you list analyzed ROWIDS, or you must have INSERT ANY TABLE system privilege.

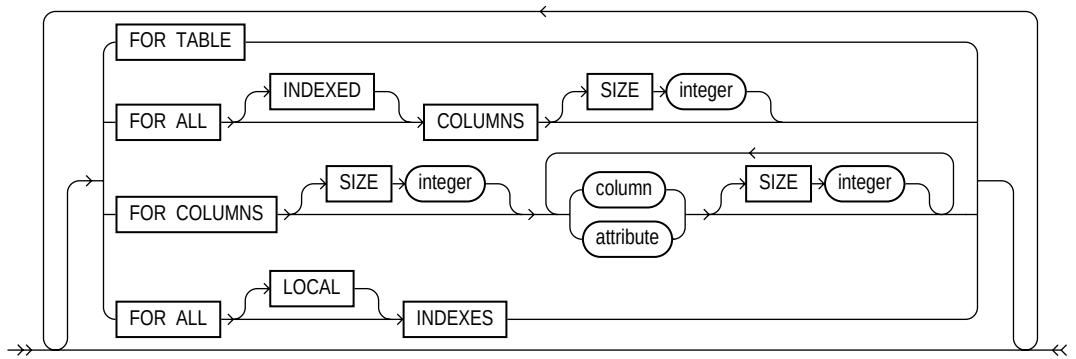
See also “Restrictions” on page 4-160.

Syntax





for_clause ::=



Keywords and Parameters

schema is the schema containing the index, table, or cluster. If you omit *schema*, Oracle assumes the index, table, or cluster is in your own schema.

index identifies an index to be analyzed (if no FOR clause is used).

table identifies a table to be analyzed. When you collect statistics for a table, Oracle also automatically collects the statistics for each of the table's indexes, provided that no FOR clauses are used.

PARTITION specifies that statistics will be gathered for *(partition_name)*. You cannot use this option when analyzing clusters.

cluster identifies a cluster to be analyzed. When you collect statistics for a cluster, Oracle also automatically collects the statistics for all the cluster's tables and all their indexes, including the cluster index. See also "Clusters" on page 4-163.

OBJ VALIDATE REF UPDATE validates the REFs in the specified table, checks the ROWID portion in each REF, compares it with the true ROWID, and corrects, if necessary. You can use this option only when analyzing a table.

COMPUTE STATISTICS computes exact statistics about the analyzed object and stores them in a data dictionary. See also "Collecting Statistics" on page 4-160.

ESTIMATE STATISTICS estimates statistics about the analyzed object and stores them in the data dictionary.

- SAMPLE specifies the amount of data from the analyzed object Oracle samples to estimate statistics. If you omit this parameter, Oracle samples 1064 rows. If you specify more than half of the data, Oracle reads all the data and computes the statistics.
- ROWS causes Oracle to sample *integer* rows of the table or cluster or *integer* entries from the index. The integer must be at least 1.
- PERCENT causes Oracle to sample *integer* percent of the rows from the table or cluster or *integer* percent of the index entries. The integer can range from 1 to 99.

for_clause specifies whether an entire table or index, or just particular columns, will be analyzed. The following clauses apply only to the ANALYZE TABLE version of this command:

- FOR TABLE collects table statistics for the table.
- FOR ALL COLUMNS collects column statistics for all columns and scalar object attributes.

  INDEX collects column statistics for all indexed columns in the table.

	FOR COLUMNS	collects column statistics for the specified columns and scalar object attributes.
	OBJ *attribute*	specifies the qualified column name of an item in an object.
	FOR ALL INDEXES	all indexes associated with the table will be analyzed.
	FOR ALL LOCAL INDEXES	specifies that all local index partitions are analyzed. You must specify the keyword LOCAL if the PARTITION (*partition_name*) clause and the index option are specified.
	SIZE	specifies the maximum number of partitions in the histogram. The default value is 75, minimum value is 1, and maximum value is 254.
	Histogram statistics are described in *Oracle8 Tuning*. See also "Columns" on page 4-161.	
DELETE STATISTICS	deletes any statistics about the analyzed object that are currently stored in the data dictionary. See also "Deleting Statistics" on page 4-163.	
VALIDATE STRUCTURE	validates the structure of the analyzed object. If you use this option when analyzing a cluster, Oracle automatically validates the structure of the cluster's tables. If you use this option when analyzing a partitioned table, Oracle also verifies that the row belongs to the correct partition. See also "Validating Structures" on page 4-164.	
	INTO	specifies a table into which Oracle lists the ROWIDs of the partitions whose rows do not collate correctly. If you omit *schema*, Oracle assumes the list is in your own schema. If you omit this clause all together, Oracle assumes that the table is named INVALID_ROWS. The SQL script used to create this table is UTLVALID.SQL.
	CASCADE	validates the structure of the indexes associated with the table or cluster. If you use this option when validating a table, Oracle also validates the table's indexes. If you use this option when validating a cluster, Oracle also validates all the clustered tables' indexes, including the cluster index.
LIST CHAINED ROWS	identifies migrated and chained rows of the analyzed table or cluster. You cannot use this option when analyzing an index.	
	INTO	specifies a table into which Oracle lists the migrated and chained rows. If you omit schema, Oracle assumes the list table is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named CHAINED_ROWS. The script used to create this table is UTLCHAIN.SQL. The list table must be on your local database.

To analyze index-organized tables, you must create a separate chained-rows table for each index-organized table created to accommodate the primary key storage of index-organized tables. Use the SQL scripts DBMSIOTC.SQL and PRVTIOTC.PLB to define the BUILD_CHAIN_ROWS_TABLE package, and then execute this procedure to create an IOT_CHAINED_ROWS table for an index-organized table.

See also "Listing Chained Rows" on page 4-166.

Restrictions

Do not use ANALYZE to collect statistics on data dictionary tables.

You cannot compute or estimate statistics for the following column types:

- REFs
- VARRAYs
- nested tables
- LOBs (LOBs are not analyzed, they are skipped)
- LONGs
- object types

Collecting Statistics

You can collect statistics about the physical storage characteristics and data distribution of an index, table, column, or cluster and store them in the data dictionary. For computing or estimating statistics:

- Computation always provides exact values, but can take longer than estimation.
- Estimation is often much faster than computation and the results are usually nearly exact.

Use estimation, rather than computation, unless you feel you need exact values. Some statistics are always computed exactly, regardless of whether you specify computation or estimation. If you choose estimation and the time saved by estimating a statistic is negligible, Oracle computes the statistic exactly.

If the data dictionary already contains statistics for the analyzed object, Oracle updates the existing statistics with the new ones.

Example I. OBJ The following statement calculates statistics for a scalar object attribute:

```
ANALYZE TABLE emp COMPUTE STATISTICS FOR COLUMNS addr.street;
```

The statistics are used by the Oracle optimizer to choose the execution plan for SQL statements that access analyzed objects. These statistics may also be useful to application developers who write such statements. For information on how these statistics are used, see *Oracle8 Tuning*.

The following sections list the statistics for that are collected for indexes, tables, columns, and clusters. The statistics marked with asterisks (*) are always computed exactly.

Indexes

For an index, Oracle collects the following statistics:

- depth of the index from its root block to its leaf blocks*
- number of leaf blocks
- number of distinct index values
- average number of leaf blocks per index value
- average number of data blocks per index value (for an index on a table)
- clustering factor (how well ordered the rows are about the indexed values)

Index statistics appear in the data dictionary views USER_INDEXES, ALL_ INDEXES, and DBA_INDEXES.

Tables

For a table, Oracle collects the following statistics:

- number of rows
- number of data blocks currently containing data *
- number of data blocks allocated to the table that have never been used *
- average available free space in each data block in bytes
- number of chained rows
- average row length, including the row's overhead, in bytes

Table statistics appear in the data dictionary views USER_TABLES, ALL_TABLES, and DBA_TABLES.

Columns

Column statistics can be based on the entire column or can use a histogram. A histogram partitions the values in the column into bands, so that all column values in a band fall within the same range. In some cases, it is useful to see how many

values fall in various ranges. Oracle's histograms are height balanced as opposed to width balanced. This means that the column values are divided into bands so that each band contains approximately the same number of values. The useful information the histogram provides, then, is where in the range of values the endpoints fall. Width-balanced histograms, in contrast, divide the data into a number of ranges, all of which are the same size, and then count the number of values falling into each range.

Oracle collects the following column statistics:

- number of distinct values in the column as a whole
- maximum and minimum values in each band

When to Use Histograms

For uniformly distributed data, the cost-based approach makes fairly accurate guesses at the cost of executing a particular statement. For non-uniformly distributed data, Oracle allows you to store histograms describing the data distribution of a particular column. These histograms are stored in the dictionary and can be used by the cost-based optimizer.

Histograms are persistent objects, so there is a maintenance and space cost for using them. You should compute histograms only for columns that you know have highly skewed data distribution. Also, be aware that histograms, as well as all optimizer statistics, are static. If the data distribution of a column changes frequently, you must reissue the ANALYZE command to recompute the histogram for that column.

Histograms are not useful for columns with the following characteristics:

- all predicates on the column use bind variables
- the column data is uniformly distributed
- the column is not used in WHERE clauses of queries
- the column is unique and is used only with equality predicates

Create histograms on columns that are frequently used in WHERE clauses of queries and have a highly skewed data distribution. You create a histogram by using the ANALYZE TABLE command. For example, if you want to create a 10-band histogram on the SAL column of the EMP table, issue the following statement:

```
ANALYZE TABLE emp
   COMPUTE STATISTICS FOR COLUMNS sal SIZE 10;
```

You can also collect histograms for a single partition of a table. The following statement analyzes the EMP table partition P1:

```
ANALYZE TABLE emp PARTITION (p1) COMPUTE STATISTICS;
```

Column statistics appear in the data dictionary views: USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS.

Histograms appear in the data dictionary views USER_HISTOGRAMS, DBA_ HISTOGRAMS, and ALL_HISTOGRAMS.

Clusters

For an indexed cluster, Oracle collects the average number of data blocks taken up by a single cluster key value and all of its rows. For a hash clusters, Oracle collects the average number of data blocks taken up by a single hash key value and all of its rows. These statistics appear in the data dictionary views USER_CLUSTERS and DBA_CLUSTERS.

Example II. The following statement estimates statistics for the CUST_HISTORY table and all of its indexes:

```
ANALYZE TABLE cust_history
   ESTIMATE STATISTICS;
```

Deleting Statistics

With the DELETE STATISTICS option of the ANALYZE command, you can remove existing statistics about an object from the data dictionary. You may want to remove statistics if you no longer want the Oracle optimizer to use them.

When you use the DELETE STATISTICS option on a table, Oracle also automatically removes statistics for all the table's indexes. When you use the DELETE STATISTICS option on a cluster, Oracle also automatically removes statistics for all the cluster's tables and all their indexes, including the cluster index.

Example. The following statement deletes statistics about the CUST_HISTORY table and all its indexes from the data dictionary:

```
ANALYZE TABLE cust_history
   DELETE STATISTICS;
```

Validating Structures

With the VALIDATE STRUCTURE option of the ANALYZE command, you can verify the integrity of the structure of an index, table, or cluster. If Oracle successfully validates the structure, a message confirming its validation is returned to you. If Oracle encounters corruption in the structure of the object, an error message is returned to you. In this case, drop and re-create the object.

Validating the structure of a object prevents SELECT, INSERT, UPDATE, and DELETE statements from concurrently accessing the object. Therefore, do not use this option on the tables, clusters, and indexes of your production applications during periods of high database activity.

Indexes

For an index, the VALIDATE STRUCTURE option verifies the integrity of each data block in the index and checks for block corruption. Note that this option 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 option.

When you use the VALIDATE STRUCTURE option on an index, Oracle also collects statistics about the index and stores them in the data dictionary view INDEX_STATS. Oracle overwrites any existing statistics about previously validated indexes. At any time, INDEX_STATS can contain only one row describing only one index. The INDEX_STATS view is described in the *Oracle8 Reference*.

The statistics collected by this option are not used by the Oracle optimizer. Do not confuse these statistics with the statistics collected by the COMPUTE STATISTICS and ESTIMATE STATISTICS options.

Example I. The following statement validates the structure of the index PARTS_INDEX:

```
ANALYZE INDEX parts_index
  VALIDATE STRUCTURE;
```

Tables

For a table, the VALIDATE STRUCTURE option verifies the integrity of each of the table's data blocks and rows. You can use the CASCADE option to also validate the structure of all indexes on the table as well and to perform cross-referencing between the table and each of its indexes. For each index, the cross-referencing involves the following validations:

- Each value of the table's indexed column must match the indexed column value of an index entry. The matching index entry must also identify the row in the table by the correct ROWID.
- Each entry in the index must identify a row in the table. The indexed column value in the index entry must match that of the identified row.

Example II. The following statement analyzes the EMP table and all of its indexes:

```
ANALYZE TABLE emp
   VALIDATE STRUCTURE CASCADE;
```

OBJ For a table, the VALIDATE REF UPDATE option verifies the REFs 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, the REF is updated so that the ROWID portion is correct.

Example III. OBJ The following statement validates the REFs in the EMP table:

```
ANALYZE TABLE emp VALIDATE REF UPDATE;
```

Clusters

For a cluster, the VALIDATE STRUCTURE option verifies the integrity of each row in the cluster and automatically validates the structure of each of the cluster's tables. You can use the CASCADE option to also validate the structure of all indexes on the cluster's tables as well, including the cluster index.

Example IV. The following statement analyzes the ORDER_CUSTS cluster, all of its tables, and all of their indexes, including the cluster index:

```
ANALYZE CLUSTER order_custs
    VALIDATE STRUCTURE CASCADE;
```

Partitioned Tables

There is no rule-based optimizer for partitioned tables, so it is important to analyze partitioned tables and indexes regularly.

For a partitioned table, the VALIDATE STRUCTURE option verifies each row in the partition to verify whether the column values of the partitioning columns collate less than the partition bound of that partition and greater than the partition bound of the previous partition (except the first partition). If the row does not collate correctly, the ROWID is inserted into the INVALID_ROWS table.

Listing Chained Rows

With the LIST option of the ANALYZE command, you can collect information about the migrated and chained rows in a table or cluster. A *migrated row* is one that has been moved from one data block to another. For example, Oracle migrates a row in a cluster if its cluster key value is updated. A *chained row* is one that is contained in more than one data block. For example, Oracle chains a row of a table or cluster if the row is too long to fit in a single data block. Migrated and chained rows may cause excessive I/O. You may want to identify such rows to eliminate them. For information on eliminating migrated and chained rows, see *Oracle8 Tuning*.

You can use the INTO clause to specify an output table into which Oracle places this information. The definition of a sample output table CHAINED_ROWS is provided in a SQL script available on your distribution media. Your list table must have the same column names, types, and sizes as the CHAINED_ROWS table. On many operating systems, the name of this script is UTLCHAIN.SQL. The actual name and location of this script depends on your operating system.

Example. The following statement collects information about all the chained rows of the table ORDER_HIST:

```
ANALYZE TABLE order_hist
    LIST CHAINED ROWS INTO cr;
```

The preceding statement places the information into the table CR. You can then examine the rows with this query:

```
SELECT *
    FROM cr
OWNER_NAME  TABLE_NAME  CLUSTER_NAME  HEAD_ROWID         TIMESTAMP
----------  ----------  ------------  ------------------ ---------
SCOTT       ORDER_HIST                AAAAZzAABAAABrXAAA 15-MAR-96
```

Related Topics

Oracle8 Tuning

ARCHIVE LOG clause

Purpose

To manually archive redo log file groups or to enable or disable automatic archiving. See also “Restrictions” on page 4-169.

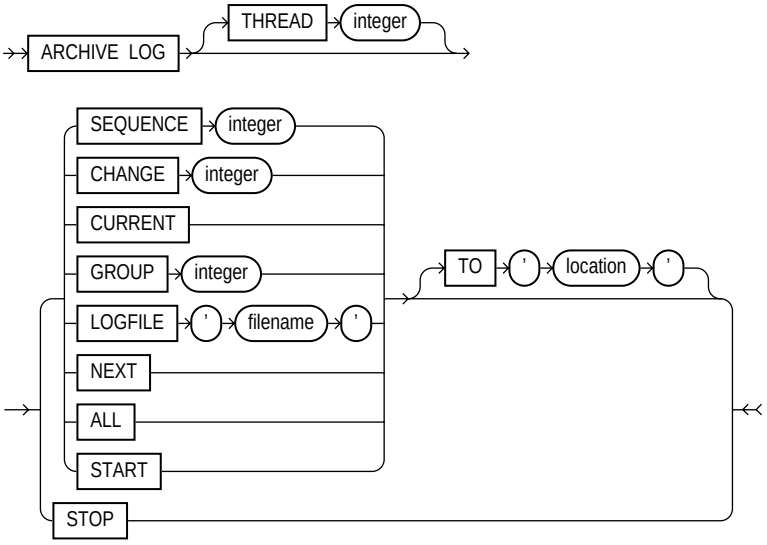
Prerequisites

The ARCHIVE LOG clause must appear in an ALTER SYSTEM command. You must have the privileges necessary to issue this statement. For information on these privileges, see ALTER SYSTEM on page 4-88.

You must also have the OSDBA or OSOPER role enabled.

You can use most of the options of this clause when your instance has the database mounted, open or closed. Options that require your instance to have the database open are noted.

Syntax



Keywords and Parameters

THREAD	specifies the thread containing the redo log file group to be archived. You need to specify this parameter only if you are using Oracle with the Parallel Server option in parallel mode.
SEQUENCE	manually archives the online redo log file group identified by the log sequence number *integer* in the specified thread. If you omit the THREAD parameter, Oracle archives the specified group from the thread assigned to your instance.
CHANGE	manually archives 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, Oracle performs a log switch. If you omit the THREAD parameter, Oracle archives the groups containing this SCN from all enabled threads. You can use this option only when your instance has the database open.
CURRENT	manually archives the current redo log file group of the specified thread, forcing a log switch. If you omit the THREAD parameter, Oracle archives all redo log file groups from all enabled threads, including logs previous to current logs. You can use this option only when your instance has the database open.
GROUP	manually archives 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 examining the data dictionary view DBA_LOG_FILES. If you specify both the THREAD and GROUP parameters, the specified redo log file group must be in the specified thread.
LOGFILE	manually archives the online redo log file group containing the redo log file member identified by '*filename*'. If you specify both the THREAD and LOGFILE parameters, the specified redo log file group must be in the specified thread.
NEXT	manually archives 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, Oracle archives the earliest unarchived redo log file group from any enabled thread.
ALL	manually archives all online redo log file groups from the specified thread that are full but have not been archived. If you omit the THREAD parameter, Oracle archives all full unarchived redo log file groups from all enabled threads.
START	enables automatic archiving of redo log file groups. You can enable automatic archiving only for the thread assigned to your instance.
TO	specifies the location to which the redo log file group is 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, Oracle archives the redo log file group to the location specified by the initialization parameter LOG_ARCHIVE_DEST.
STOP	disables automatic archiving of redo log file groups. You can disable automatic archiving only for the thread assigned to your instance.

Restrictions

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, Oracle returns an error. If you specify a redo log file group for archiving with the CHANGE parameter or CURRENT option, and earlier redo log file groups are not yet archived, Oracle archives all unarchived groups up to and including the specified group.

You can also manually archive redo log file groups with the ARCHIVE LOG Server Manager command. For information on this command, see the *Oracle Server Manager User's Guide*.

You can also choose to have Oracle archive redo log files groups automatically. For information on automatic archiving, see *Oracle8 Administrator's Guide.* Note that you can always manually archive redo log file groups regardless of whether automatic archiving is enabled.

Example I. The following statement manually archives the redo log file group with the log sequence number 4 in thread number 3:

```
ALTER SYSTEM ARCHIVE LOG THREAD 3 SEQUENCE 4;
```

Example II. 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;
```

Example III. 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$]';
```

Related Topics

ALTER SYSTEM on page 4-88

AUDIT (SQL Statements)

Purpose

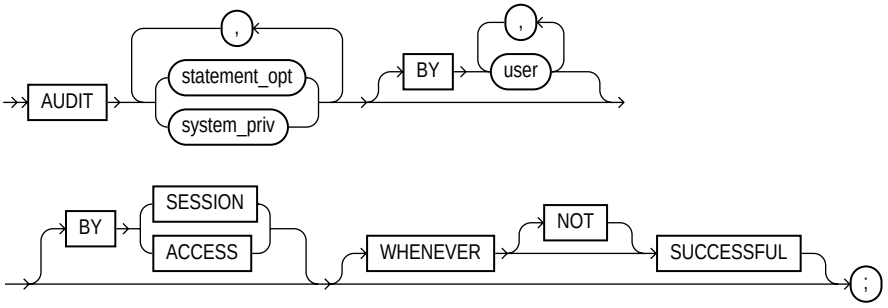
To choose specific SQL statements for auditing in subsequent user sessions. To choose particular schema objects for auditing, see AUDIT (Schema Objects) on page 4-178. See also "Auditing" on page 4-171.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

You must have AUDIT SYSTEM system privilege.

Syntax



Keywords and Parameters

statement_opt chooses specific SQL statements for auditing. For a list of these statement options and the SQL statements they audit, see Table 4–6 and Table 4–7. See also "Statement Options for Database Objects" on page 4-172 and "Statement Options for Commands" on page 4-174.

system_priv chooses SQL statements that are authorized by the specified system privilege for auditing. For a list of all system privileges and the SQL statements that they authorize, see Table 4–19. See also "Shortcuts for System Privileges and Statement Options" on page 4-176.

BY *user*	chooses only SQL statements issued by specified users for auditing. If you omit this clause, Oracle audits all users' statements.
BY SESSION	causes Oracle to write a single record for all SQL statements of the same type issued in the same session.
BY ACCESS	causes Oracle to write one record for each audited statement. If you specify statement options or system privileges that audit data definition language (DDL) statements, Oracle automatically audits by access regardless of whether you specify the BY SESSION or BY ACCESS option. For statement options and system privileges that audit other types of SQL statements other than DDL, you can specify either the BY SESSION or BY ACCESS option. BY SESSION is the default.
WHENEVER SUCCESSFUL	chooses auditing only for statements that succeed. NOT chooses auditing only for statements that fail or result in errors. If you omit the WHENEVER clause, Oracle audits SQL statements regardless of success or failure.

Auditing

Auditing keeps track of operations performed by database users. For each audited operation, Oracle produces an audit record containing this information:

- user performing the operation
- type of operation
- object involved in the operation
- date and time of the operation

Oracle writes audit records to the audit trail. The audit trail is a database table that contains audit records. You can review database activity by examining the audit trail through data dictionary views. For information on these views, see the *Oracle8 Reference*.

To generate audit records, you must perform the following steps:

Enable auditing

You must enable auditing by setting the initialization parameter AUDIT_TRAIL = DB.

Specify auditing options

To specify auditing options, you must use the AUDIT command. Auditing options choose which SQL commands, operations, database objects, and users Oracle audits. After you specify auditing options, they appear in the data dictionary. For more information on data dictionary views containing auditing options see the *Oracle8 Reference*.

You can specify auditing options regardless of whether auditing is enabled. However, Oracle does not generate audit records until you enable auditing.

Auditing options specified by the AUDIT command (SQL Statements) apply only to subsequent sessions, rather than to current sessions.

Statement Options for Database Objects

Table 4–6 lists the statement options relating to database objects and the statements that they audit.

Table 4–6 Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
CLUSTER	CREATE CLUSTER AUDIT CLUSTER DROP CLUSTER TRUNCATE CLUSTER
DATABASE LINK	CREATE DATABASE LINK DROP DATABASE LINK
DIRECTORY	CREATE DIRECTORY DROP DIRECTORY
INDEX	CREATE INDEX ALTER INDEX DROP INDEX
NOT EXISTS	All SQL statements that fail because a specified object does not exist.

Table 4–6 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
PROCEDURE	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 DROP PUBLIC DATABASE LINK
PUBLIC SYNONYM	CREATE PUBLIC SYNONYM DROP PUBLIC SYNONYM
ROLE	CREATE ROLE ALTER ROLE DROP ROLE SET ROLE
ROLLBACK STATEMENT	CREATE ROLLBACK SEGMENT ALTER ROLLBACK SEGMENT DROP ROLLBACK SEGMENT
SEQUENCE	CREATE SEQUENCE DROP SEQUENCE
SESSION	Logons
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 4–6 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
TABLE	CREATE TABLE DROP TABLE TRUNCATE TABLE
TABLESPACE	CREATE TABLESPACE ALTER TABLESPACE DROP TABLESPACE
TRIGGER	CREATE TRIGGER ALTER TRIGGER with ENABLE and DISABLE options DROP TRIGGER ALTER TABLE with ENABLE ALL TRIGGERS and DISABLE ALL TRIGGERS clauses
OBJ TYPE	CREATE TYPE CREATE TYPE BODY ALTER TYPE DROP TYPE DROP TYPE BODY
USER	CREATE USER ALTER USER DROP USER
VIEW	CREATE VIEW DROP VIEW

Statement Options for Commands

Table 4–7 lists additional statement options related to commands and the SQL statements and operations that they audit.

Table 4–7 Statement Auditing Options for Commands

Statement Option	SQL Statements and Operations
ALTER SEQUENCE	ALTER SEQUENCE
ALTER TABLE	ALTER TABLE
COMMENT TABLE	COMMENT ON TABLE *table, view, snapshot* COMMENT ON COLUMN *table.column, view.column, snapshot.column*

Table 4–7 (Cont.) Statement Auditing Options for Commands

Statement Option	SQL Statements and Operations
DELETE TABLE	DELETE FROM *table, view*
EXECUTE PROCEDURE	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, snapshot.* REVOKE *privilege* ON t*able, view, snapshot*
OBJ GRANT TYPE	GRANT *privilege* ON TYPE REVOKE *privilege* ON TYPE
INSERT TABLE	INSERT INTO *table, view*
LOCK TABLE	LOCK TABLE *table, view*
SELECT SEQUENCE	Any statement containing sequence.CURRVAL or sequence.NEXTVAL
SELECT TABLE	SELECT FROM *table, view, snapshot*
UPDATE TABLE	UPDATE *table, view*

Example I. 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 error, issue the following statement:

```
AUDIT ROLE
    WHENEVER NOT SUCCESSFUL;
```

Example II. 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 SCOTT and BLAKE that query or update a table or view, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE
    BY scott, blake;
```

Example III. To choose auditing for statements issued using the DELETE ANY TABLE system privilege, issue the following statement:

```
AUDIT DELETE ANY TABLE;
```

Example IV. To choose auditing for statements issued using the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT CREATE ANY DIRECTORY;
```

Example V. 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;
```

Shortcuts for System Privileges and Statement Options

Oracle provides shortcuts for specifying groups of system privileges and statement options at once. However, Oracle encourages you to choose individual system privileges and statement options for auditing, because these shortcuts may not be supported in future versions of Oracle. The shortcuts are follows:

CONNECT	is equivalent to specifying the CREATE SESSION system privilege.
RESOURCE	is equivalent to specifying the following system privileges:

- ALTER SESSION
- CREATE CLUSTER
- CREATE DATABASE LINK
- CREATE PROCEDURE
- CREATE ROLLBACK SEGMENT
- CREATE SEQUENCE
- CREATE SYNONYM
- CREATE TABLE
- CREATE TABLESPACE
- CREATE VIEW

DBA — is equivalent to the SYSTEM GRANT statement option and the following system privileges:

- AUDIT SYSTEM
- CREATE PUBLIC DATABASE LINK
- CREATE PUBLIC SYNONYM
- CREATE ROLE
- CREATE USER

ALL — equivalent to specifying all statement options shown in Table 4–6, but not the additional statement options shown in Table 4–7.

ALL PRIVILEGES — is equivalent to specifying all system privileges.

Related Topics

AUDIT (Schema Objects) on page 4-178
NOAUDIT (Schema Objects) on page 4-463

AUDIT (Schema Objects)

Purpose

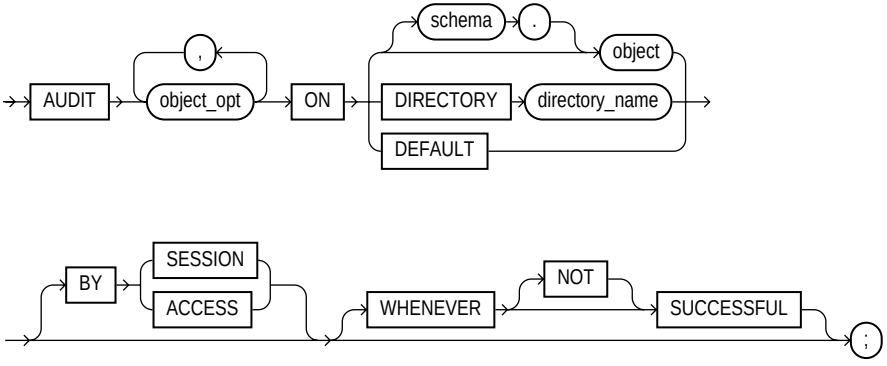
To choose a specific schema object for auditing. To choose particular SQL commands for auditing, see AUDIT (SQL Statements) on page 4-170.

Auditing keeps track of operations performed by database users. For a brief conceptual overview of auditing, including how to enable auditing, see the AUDIT (SQL Statements) on page 4-170. Note that auditing options established by the AUDIT command (Schema Objects) apply to current sessions as well as to subsequent sessions.

Prerequisites

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, you must have AUDIT ANY system privilege.

Syntax



Keywords and Parameters

object_opt	specifies a particular operation for auditing. Table 4–8 shows each object option and the types of objects to which it applies. See also "Object Options" on page 4-179.
schema	is the schema containing the object chosen for auditing. If you omit schema, Oracle assumes the object is in your own schema.
object	identifies the object chosen for auditing. The object must be a table; view; sequence; stored procedure, function, or package; snapshot; or library. You can also specify a synonym for a table, view, sequence, procedure, stored function, package, or snapshot.
ON DEFAULT	establishes the specified object options as default object options for subsequently created objects. See also "Default Auditing" on page 4-180.
DIRECTORY *directory_name*	identifies the name of the directory chosen for auditing.
BY SESSION	means that Oracle writes a single record for all operations of the same type on the same object issued in the same session.
BY ACCESS	means that Oracle writes one record for each audited operation.
If you omit both of the preceding options, Oracle audits by session.	
WHENEVER SUCCESSFUL	chooses auditing only for SQL statements that complete successfully. NOT chooses auditing only for statements that fail, or result in errors. If you omit the WHENEVER clause entirely, Oracle audits all SQL statements, regardless of success or failure.

Object Options

Table 4–8 shows the object options you can choose for each type of object.

Table 4–8 Object Auditing Options

Object Option	Table	View	Sequence	Procedure Function Package	Snapshot	Library	Directory
ALTER	X		X		X		
AUDIT	X	X	X	X	X		X
COMMENT	X	X			X		
DELETE	X	X			X		
EXECUTE				X		X	
GRANT	X	X	X	X	X	X	X
INDEX	X				X		
INSERT	X	X			X		
LOCK	X	X			X		
READ							X
RENAME	X	X		X	X		
SELECT	X	X	X		X		
UPDATE	X	X			X		

The name of each object option specifies a command to be audited. For example, if you choose to audit a table with the ALTER option, Oracle audits all ALTER TABLE statements issued against the table. If you choose to audit a sequence with the SELECT option, Oracle audits all statements that use any of the sequence's values.

Shortcuts for Object Options

Oracle provides a shortcut for specifying object auditing options:

ALL — is equivalent to specifying all object options applicable for the type of object. You can use this shortcut rather than explicitly specifying all options for an object.

Default Auditing

You can use the DEFAULT option of the AUDIT command to specify auditing options for objects that have not yet been created. Once you have established these

default auditing options, any subsequently created object is automatically audited with those options. Note that the default auditing options for a view are always the union of the auditing options for the view's base tables.

If 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 command.

Example I. To choose auditing for every SQL statement that queries the EMP table in the schema SCOTT, issue the following statement:

```
AUDIT SELECT
    ON scott.emp;
```

To choose auditing for every statement that successfully queries the EMP table in the schema SCOTT, issue the following statement:

```
AUDIT SELECT
    ON scott.emp
    WHENEVER SUCCESSFUL;
```

To choose auditing for every statement that queries the EMP table in the schema SCOTT and results in an Oracle error, issue the following statement:

```
AUDIT SELECT
    ON scott.emp
    WHENEVER NOT SUCCESSFUL;
```

Example II. To choose auditing for every statement that inserts or updates a row in the DEPT table in the schema BLAKE, issue the following statement:

```
AUDIT INSERT, UPDATE
    ON blake.dept;
```

Example III. To choose auditing for every statement that performs any operation on the ORDER sequence in the schema ADAMS, issue the following statement:

```
AUDIT ALL
    ON adams.order;
```

The above statement uses the ALL short cut to choose auditing for the following statements that operate on the sequence:

- ALTER SEQUENCE
- AUDIT

- GRANT
- any statement that accesses the sequence's values using the pseudocolumns CURRVAL or NEXTVAL

Example IV. To choose auditing for every statement that reads files from the BFILE_DIR1 directory, issue the following statement:

```
AUDIT READ ON DIRECTORY bfile_dir1;
```

Example V. 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 audited with the specified options that apply to them, provided that auditing has been enabled:

- If you create a table, Oracle automatically audits any ALTER, INSERT, UPDATE, or DELETE statements issued against the table.
- If you create a view, Oracle automatically audits any INSERT, UPDATE, or DELETE statements issued against the view.
- If you create a sequence, Oracle automatically audits any ALTER statements issued against the sequence.
- If you create a procedure, package, or function, Oracle automatically audits any ALTER statements issued against it.

Related Topics

AUDIT (SQL Statements) on page 4-170
NOAUDIT (Schema Objects) on page 4-463

COMMENT

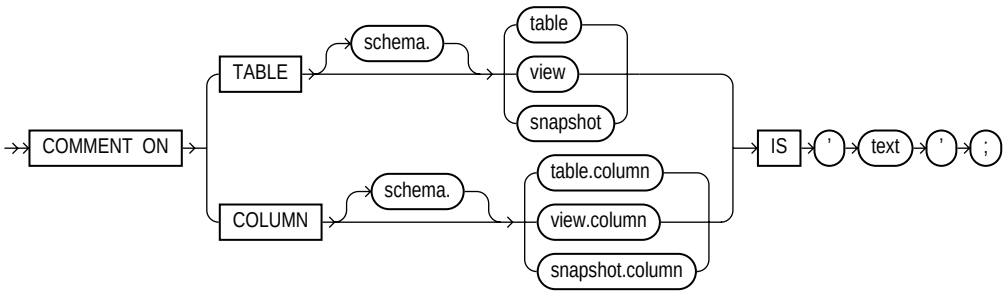
Purpose

To add a comment about a table, view, snapshot, or column into the data dictionary. See also "Using Comments" on page 4-183.

Prerequisites

The table, view, or snapshot must be in your own schema or you must have COMMENT ANY TABLE system privilege.

Syntax



Keywords and Parameters

TABLE	specifies the schema and name of the table, view, or snapshot to be commented.
COLUMN	specifies the name of the column of a table, view, or snapshot to be commented. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.
IS '*text*'	is the text of the comment. See the syntax description of 'text' in "Text" on page 2-2.

Using Comments

You can effectively drop a comment from the database by setting it to the empty string ' '. For information on the data dictionary views that contain comments, see *Oracle8 Reference*.

Example. To insert an explanatory remark on the NOTES column of the SHIPPING table, you might issue the following statement:

```
COMMENT ON COLUMN shipping.notes
    IS 'Special packing or shipping instructions';
```

To drop this comment from the database, issue the following statement:

```
COMMENT ON COLUMN shipping.notes IS ' ';
```

Related Topics

"Comments" on page 2-38

COMMIT

Purpose

To end your current transaction and make permanent all changes performed in the transaction. This command also erases all savepoints in the transaction and releases the transaction's locks. See also "About Transactions" on page 4-186.

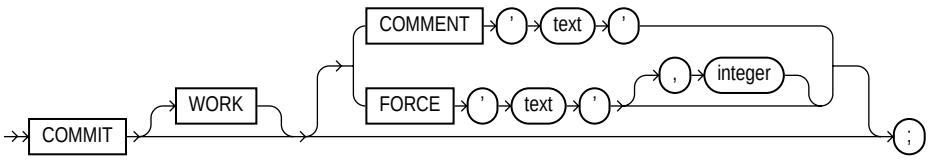
You can also use this command to commit an in-doubt distributed transaction manually. See "Ending Transactions" on page 4-187 for more information on transactions.

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



Keywords and Parameters

WORK	is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
COMMENT	specifies a comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.

FORCE	manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the integer to specifically assign the transaction a system change number (SCN). If you omit the integer, the transaction is committed using the current SCN. COMMIT statements using the FORCE clause are not supported in PL/SQL.

About Transactions

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle treats as a single unit. A transaction begins with the first executable SQL statement after a COMMIT, ROLLBACK, or connection to the database. A transaction ends with a COMMIT, ROLLBACK, or disconnection (intentional or unintentional) from the database. Note that Oracle issues an implicit COMMIT before and after any data definition language (DDL) statement.

You can also use a COMMIT or ROLLBACK statement to terminate a read-only transaction begun by a SET TRANSACTION statement.

Example I. This example inserts a row into the DEPT table and commits this change:

```
INSERT INTO dept   VALUES (50, 'MARKETING', 'TAMPA');
COMMIT WORK;
```

Example II. The following statement commits the current transaction and associates a comment with it:

```
COMMIT WORK
    COMMENT 'In-doubt transaction Code 36, Call (415) 555-2637';
```

If a network or machine failure prevents this distributed transaction from committing properly, Oracle 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.

Distributed Transactions

Oracle with the distributed option allows you to perform distributed transactions, or transactions that modify data on multiple databases. To commit a distributed transaction, you need only issue a COMMIT statement as you would to commit any other transaction. Each component of the distributed transaction is then committed on each database.

If a network or machine failure occurs during the commit process for a distributed transaction, the state of the transaction may be unknown, or in-doubt. After consultation with the administrators of the other databases involved in the transaction, you may decide to manually commit or roll back the transaction on your local database. You can manually commit the transaction on your local database by using the FORCE clause of the COMMIT command. For more information on these topics, see *Oracle8 Distributed Database Systems*.

Note that a COMMIT statement with a FORCE clause only commits the specified transaction. Such a statement does not affect your current transaction.

Example. The following statement manually commits an in-doubt distributed transaction:

```
COMMIT FORCE '22.57.53';
```

Ending Transactions

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. If you do not explicitly commit the transaction and the program terminates abnormally, 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 to roll back the current transaction.

Related Topics

COMMENT on page 4-183
COMMENT on page 4-183
SET TRANSACTION on page 4-519

CONSTRAINT clause

Purpose

To define an integrity constraint. An *integrity constraint* is a rule that restricts the values for one or more columns in a table or an index-organized table.

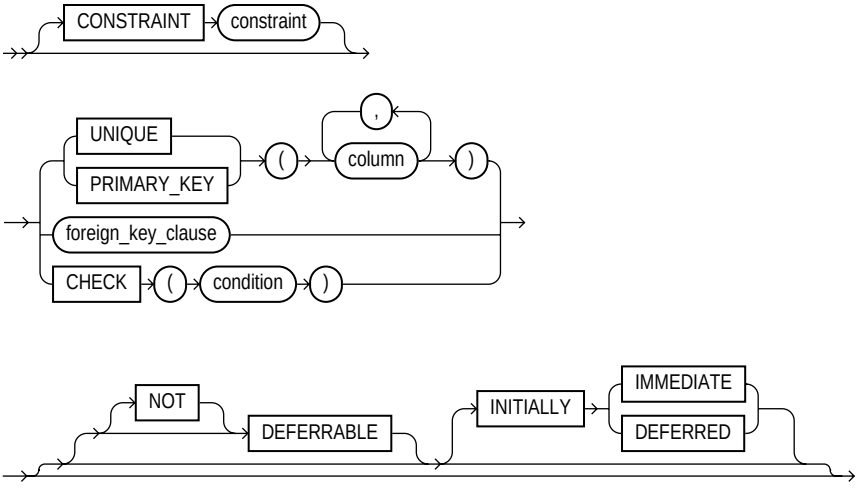
Prerequisites

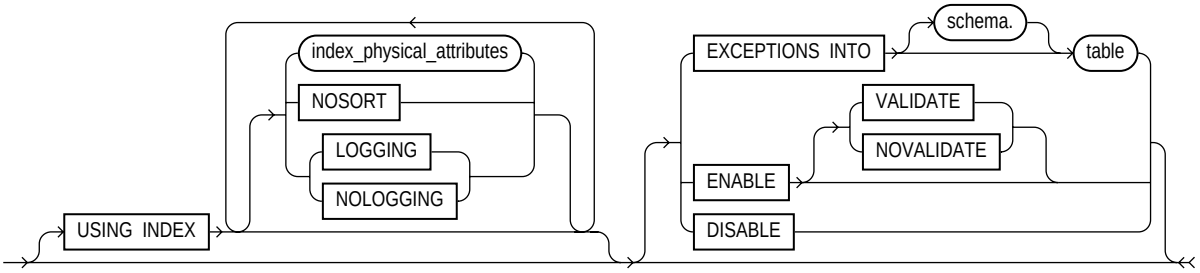
CONSTRAINT clauses can appear in either CREATE TABLE or ALTER TABLE commands. To define an integrity constraint, you must have the privileges necessary to issue one of these commands. See CREATE TABLE on page 4-306 and ALTER TABLE on page 4-106.

Defining a constraint may also require additional privileges or preconditions, depending on the type of constraint. For information on these privileges, see the descriptions of each type of integrity constraint in “Defining Integrity Constraints” on page 4-192.

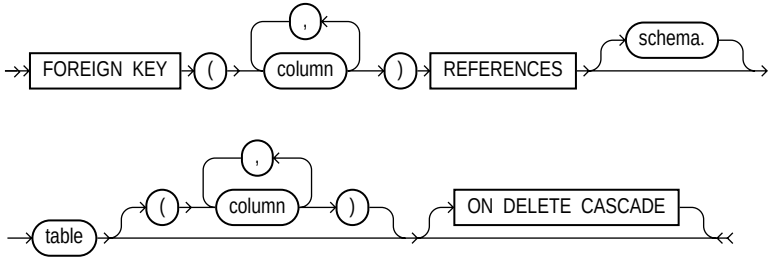
Syntax

table_constraint::=

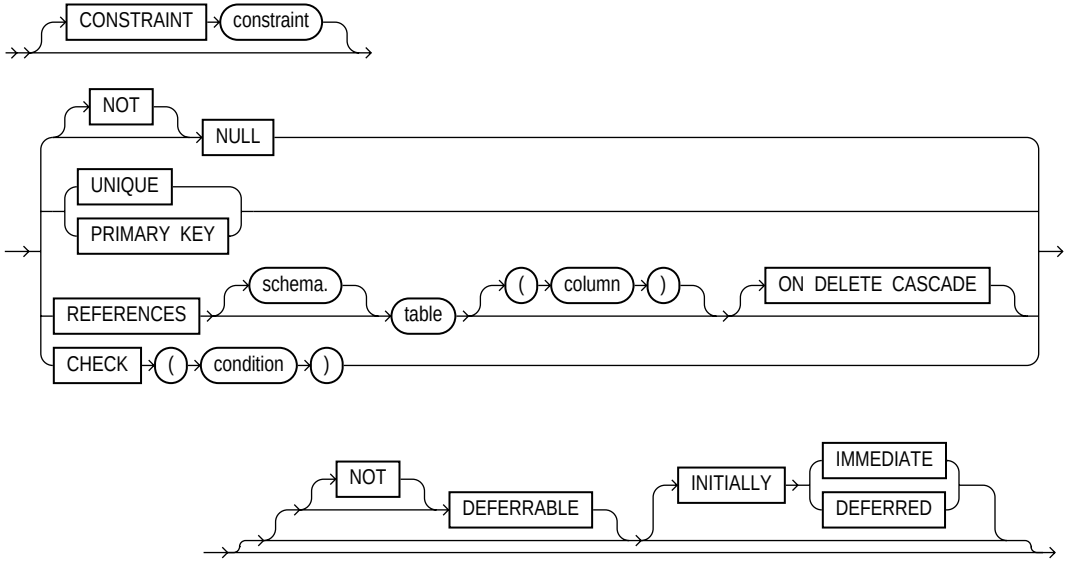


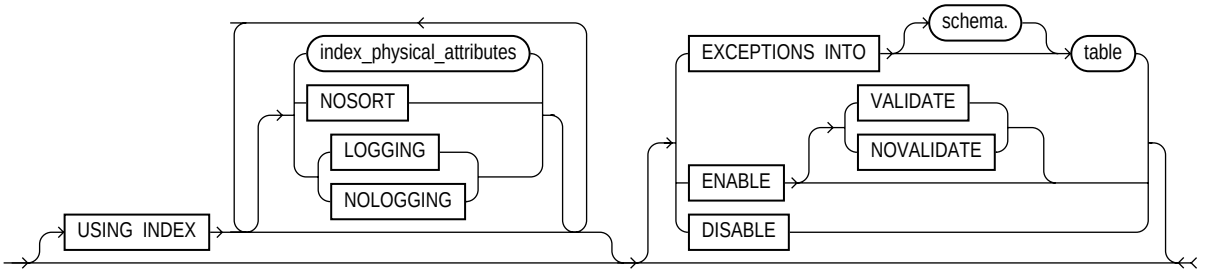


foreign_key_clause ::=

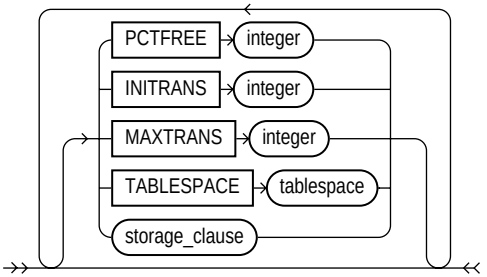


column_constraint::=





index_physical_attributes ::=



storage_clause: See the STORAGE clause on page 4-523.

Keywords and Parameters

CONSTRAINT	identifies the integrity constraint by the name *constraint*. Oracle stores this name in the data dictionary along with the definition of the integrity constraint. If you omit this identifier, Oracle generates a name with this form: SYS_Cn. See also "Defining Integrity Constraints" on page 4-192. If you do not specify NULL or NOT NULL in a column definition, NULL is the default.
UNIQUE	designates a column or combination of columns as a unique key. You cannot define UNIQUE constraints on index-organized tables. See also "UNIQUE Constraints" on page 4-193.
PRIMARY KEY	designates a column or combination of columns as the table's primary key. See also "PRIMARY KEY Constraints" on page 4-195.
FOREIGN KEY	designates a column or combination of columns as the foreign key in a referential integrity constraint.
REFERENCES	identifies the primary or unique key that is referenced by a foreign key in a referential integrity constraint. See also "Referential Integrity Constraints" on page 4-196.

ON DELETE CASCADE	specifies that Oracle maintains referential integrity by automatically removing dependent foreign key values if you remove a referenced primary or unique key value.
NULL	specifies that a column can contain null values.
NOT NULL	specifies that a column cannot contain null values. See also "NOT NULL Constraints" on page 4-193.
CHECK	specifies a condition that each row in the table must satisfy. See also "CHECK Constraints" on page 4-201.
DEFERRABLE	indicates that constraint checking can be deferred until the end of the transaction by using the SET CONSTRAINT(S) command.
NOT DEFERRABLE	indicates that this constraint is checked at the end of each DML statement. You cannot defer a NOT DEFERRABLE constraint with the SET CONSTRAINT(S) command. If you do not specify DEFERRABLE or NOT DEFERRABLE, then NOT DEFERRABLE is the default. See also "DEFERRABLE Constraints" on page 4-204.
INITIALLY IMMEDIATE	indicates that at the start of every transaction, the default is to check this constraint at the end of every DML statement. If no INITIALLY clause is specified, INITIALLY IMMEDIATE is the default.
INITIALLY DEFERRED	implies that this constraint is DEFERRABLE and specifies that, by default, the constraint is checked only at the end of each transaction.
USING INDEX	specifies parameters for the index Oracle uses to enable a UNIQUE or PRIMARY KEY constraint. The name of the index is the same as the name of the constraint. You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, STORAGE, PCTFREE, LOGGING, and NOLOGGING parameters for the index. For information on these parameters, see CREATE TABLE on page 4-306. Use this clause only when enabling UNIQUE and PRIMARY KEY constraints.
NOSORT	indicates that the rows are stored in the database in ascending order and therefore Oracle does not have to sort the rows when creating the index.
EXCEPTIONS INTO	specifies a table into which Oracle places the ROWIDs of all rows violating the constraint. **Note:** You must create an appropriate exceptions report table to accept information from the EXCEPTIONS option of the ENABLE clause before enabling the constraint. You can create an exception table by submitting the script UTLEXCPT.SQL, which creates a table named EXCEPTIONS. You can create additional exceptions tables with different names by modifying and resubmitting the script. The EXCEPTIONS INTO clause is a valid option only when validating a constraint (see the ENABLE clause on page 4-417) or when enabling a constraint with an ALTER TABLE command. See ALTER TABLE on page 4-106.

ENABLE VALIDATE	ensures that all new insert, delete, and update operations on the constrained data comply with the constraint. Checks that all old data also obeys the constraint. An enabled and validated constraint guarantees that all data is and will continue to be valid. This is the default.
ENABLE NOVALIDATE	ensures that all new insert, update, and delete operations on the constrained data comply with the constraint. Oracle does not verify that existing data in the table complies with the constraint.
DISABLE	disables the integrity constraint. If an integrity constraint is disabled, Oracle does not enable it. If you do not specify this option, Oracle automatically enables the integrity constraint.

You can also enable and disable integrity constraints with the ENABLE and DISABLE clauses of the CREATE TABLE and ALTER TABLE commands. See the ENABLE clause on page 4-417 and the DISABLE clause on page 4-380. See also "Enabling and Disabling Constraints" on page 4-205.

Disabled constraints can be made enabled with ALTER TABLE on page 4-106.

Defining Integrity Constraints

To define an integrity constraint, include a CONSTRAINT clause in a CREATE TABLE or ALTER TABLE statement. The CONSTRAINT clause has two syntactic forms:

table_constraint	The table_constraint syntax is part of the table definition. An integrity constraint defined with this syntax can impose rules on any columns in the table. The *table_constraint* syntax can appear in a CREATE TABLE or ALTER TABLE statement. This syntax can define any type of integrity constraint except a NOT NULL constraint.
column_ constraint	The column_constraint syntax is part of a column definition. Usually, an integrity constraint defined with this syntax can impose rules only on the column in which it is defined. The *column_constraint* syntax that appears in a CREATE TABLE statement can define any type of integrity constraint. *Column_ constraint* syntax that appears in an ALTER TABLE statement can only define or remove a NOT NULL constraint.

The *table_constraint* syntax and the *column_constraint* syntax are simply different syntactic means of defining integrity constraints. A constraint that references more than one column must be defined as a table constraint. There is no other functional

difference between an integrity constraint defined with *table_constraint* syntax and the same constraint defined with *column_constraint* syntax.

> **Note:** You cannot create a constraint on columns or attributes whose type is user-defined, LOB, or REF. The only exception is that Oracle supports creation of a NOT NULL constraint on columns or attributes of OBJECT type, VARRAY type, LOB, or REF.

NOT NULL Constraints

The NOT NULL constraint specifies that a column cannot contain nulls. To satisfy this constraint, every row in the table must contain a value for the column.

The NULL keyword indicates that a column can contain nulls. It does not actually define an integrity constraint. If you do not specify either NOT NULL or NULL, the column can contain nulls by default.

You can specify NOT NULL or NULL with *column_constraint* syntax only in a CREATE TABLE or ALTER TABLE statement, not with *table_constraint* syntax.

Example. The following statement alters the EMP table and defines and enables a NOT NULL constraint on the SAL column:

```
ALTER TABLE emp
   MODIFY (sal  NUMBER  CONSTRAINT nn_sal NOT NULL);
```

NN_SAL ensures that no employee in the table has a null salary.

UNIQUE Constraints

The UNIQUE constraint designates a column or combination of columns as a unique key. 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.

A unique key column cannot be of datatype LONG or LONG RAW. You cannot designate the same column or combination of columns as both a unique key and a primary key or as both a unique key and a cluster key. However, you can designate the same column or combination of columns as both a unique key and a foreign key.

Defining Unique Keys

You can define a unique key on a single column with *column_constraint* syntax.

Example. The following statement creates the DEPT table and defines and enables a unique key on the DNAME column:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9)  CONSTRAINT unq_dname UNIQUE,
      loc     VARCHAR2(10) );
```

The constraint UNQ_DNAME identifies the DNAME column as a unique key. This constraint ensures that no two departments in the table have the same name. However, the constraint does allow departments without names.

Alternatively, you can define and enable this constraint with the *table_constraint* syntax:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9),
     loc     VARCHAR2(10),
        CONSTRAINT unq_dname
        UNIQUE (dname)
    USING INDEX PCTFREE 20
        TABLESPACE user_x
        STORAGE (INITIAL 8K  NEXT 6K) );
```

The above statement also uses the USING INDEX option to specify storage characteristics for the index that Oracle creates to enable the constraint.

Defining Composite Unique Keys

A composite unique key is a unique key made up of a combination of columns. Oracle creates an index on the columns of a unique key, so a composite unique key can contain a maximum of 16 columns. To define a composite unique key, you must use *table_constraint* syntax rather than *column_constraint* syntax.

To satisfy a constraint that designates a composite unique key, no two rows in the table 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.

Example. The following statement defines and enables a composite unique key on the combination of the CITY and STATE columns of the CENSUS table:

```
ALTER TABLE census
    ADD CONSTRAINT unq_city_state
```

```
    UNIQUE (city, state)
    USING INDEX PCTFREE 5
        TABLESPACE user_y
    EXCEPTIONS INTO bad_keys_in_ship_cont;
```

The UNQ_CITY_STATE constraint ensures that the same combination of CITY and STATE values does not appear in the table more than once.

The CONSTRAINT clause also specifies other properties of the constraint:

- The USING INDEX option specifies storage characteristics for the index Oracle creates to enable the constraint.
- The EXCEPTIONS option causes Oracle to write information to the BAD_KEYS_IN_SHIP_CONT table about any rows currently in the CENSUS table that violate the constraint.

PRIMARY KEY Constraints

A PRIMARY KEY constraint designates a column or combination of columns as the table's primary key. To satisfy a PRIMARY KEY constraint, both of the following conditions must be true:

- 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.

A table can have only one primary key.

A primary key column cannot be of datatype LONG or LONG RAW. You cannot designate the same column or combination of columns as both a primary key and a unique key or as both a primary key and a cluster key. However, you can designate the same column or combination of columns as both a primary key and a foreign key.

Defining Primary Keys

You can use the *column_constraint* syntax to define a primary key on a single column.

Example. The following statement creates the DEPT table and defines and enables a primary key on the DEPTNO column:

```
CREATE TABLE dept
    (deptno  NUMBER(2) CONSTRAINT pk_dept PRIMARY KEY,
    dname   VARCHAR2(9),
    loc     VARCHAR2(10) );
```

The PK_DEPT constraint identifies the DEPTNO column as the primary key of the DEPT table. This constraint ensures that no two departments in the table have the same department number and that no department number is NULL.

Alternatively, you can define and enable this constraint with *table_constraint* syntax:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
    dname   VARCHAR2(9),
    loc   VARCHAR2(10),
    CONSTRAINT pk_dept PRIMARY KEY (deptno) );
```

Defining Composite Primary Keys

A composite primary key is a primary key made up of a combination of columns. Oracle creates an index on the columns of a primary key; therefore, a composite primary key can contain a maximum of 16 columns. To define a composite primary key, you must use the *table_constraint* syntax rather than the *column_constraint* syntax.

Example. The following statement defines a composite primary key on the combination of the SHIP_NO and CONTAINER_NO columns of the SHIP_CONT table:

```
ALTER TABLE ship_cont
    ADD PRIMARY KEY (ship_no, container_no) DISABLE;
```

This constraint identifies the combination of the SHIP_NO and CONTAINER_NO columns as the primary key of the SHIP_CONT table. The constraint ensures that no two rows in the table have the same values for both the SHIP_NO column and the CONTAINER_NO column.

The CONSTRAINT clause 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 option causes Oracle to define the constraint but not enable it.

Referential Integrity Constraints

A referential integrity constraint designates a column or combination of columns as a foreign key and establishes a relationship between that foreign key and a specified primary or unique key, called the referenced key. In this relationship, the

table containing the foreign key is called the *child table* and the table containing the referenced key is called the *parent table*. Note the following restrictions:

- The child and parent tables must be on the same database. They cannot be on different nodes of a distributed database. Oracle allows you to enable referential integrity across nodes of a distributed database with database triggers. For information on how to use database triggers for this purpose, see *Oracle8 Application Developer's Guide*.
- The foreign key and the referenced key can be in the same table. In this case, the parent and child tables are the same.

To satisfy a referential integrity constraint, each row of the child table must meet one of the following conditions:

- The value of the row's foreign key must appear as a referenced key value in one of the parent table's rows. The row in the child table is said to depend on the referenced key in the parent table.
- The value of one of the columns that makes up the foreign key must be null.

A referential integrity constraint is defined in the child table. A referential integrity constraint definition can include any of the following keywords:

FOREIGN KEY	identifies the column or combination of columns in the child table that makes up the foreign key. Ise this keyword only when you define a foreign key with a table constraint clause.
REFERENCES	identifies the parent table and the column or combination of columns that make up the referenced key. If you identify only the parent table and omit the column names, the foreign key automatically references the primary key of the parent table. The corresponding columns of the referenced key and the foreign key must match in number and datatypes.
ON DELETE CASCADE	allows deletion of referenced key values in the parent table that have dependent rows in the child table and causes Oracle to automatically delete dependent rows from the child table to maintain referential integrity.

If you omit this option, Oracle forbids deletions of referenced key values in the parent table that have dependent rows in the child table.

Before you define a referential integrity constraint in the child table, the referenced UNIQUE or PRIMARY KEY constraint on the parent table must already be defined. Also, the parent table must be in your own schema or you must have REFERENCES privilege on the columns of the referenced key in the parent table. Before you enable a referential integrity constraint, its referenced constraint must be enabled.

You cannot define a referential integrity constraint in a CREATE TABLE statement that contains an AS clause. Instead, you can create the table without the constraint and then add it later with an ALTER TABLE statement.

A foreign key column cannot be of datatype LONG or LONG RAW. 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. Also, a single column can be part of more than one foreign key.

Defining Referential Integrity Constraints

You can use *column_constraint* syntax to define a referential integrity constraint in which the foreign key is made up of a single column.

Example. The following statement creates the EMP table and defines and enables a foreign key on the DEPTNO column that references the primary key on the DEPTNO column of the DEPT table:

```
CREATE TABLE emp
   (empno      NUMBER(4),
    ename      VARCHAR2(10),
    job        VARCHAR2(9),
    mgr        NUMBER(4),
    hiredate   DATE,
    sal        NUMBER(7,2),
    comm       NUMBER(7,2),
    deptno     CONSTRAINT fk_deptno REFERENCES dept(deptno) );
```

The constraint FK_DEPTNO ensures that all departments given for employees in the EMP table are present in the DEPT table. However, employees can have null department numbers, meaning they are not assigned to any department. If you

wish to prevent the latter, you could create a NOT NULL constraint on the *deptno* column in the EMP table, in addition to the REFERENCES constraint.

Before you define and enable this constraint, you must define and enable a constraint that designates the DEPTNO column of the DEPT table as a primary or unique key. For the definition of such a constraint, see the example on page 4-195.

Note that the referential integrity constraint definition does not use the FOREIGN KEY keyword to identify the columns that make up the foreign key. Because the constraint is defined with a column constraint clause on the DEPTNO column, the foreign key is automatically on the DEPTNO column.

Note that the constraint definition identifies both the parent table and the columns of the referenced key. Because the referenced key is the parent table's primary key, the referenced key column names are optional.

Note that the above statement omits the DEPTNO column's datatype. Because this column is a foreign key, Oracle automatically assigns it the datatype of the DEPT.DEPTNO column to which the foreign key refers.

Alternatively, you can define a referential integrity constraint with *table_constraint* syntax:

```
CREATE TABLE emp
  (empno     NUMBER(4),
   ename     VARCHAR2(10),
   job       VARCHAR2(9),
   mgr       NUMBER(4),
   hiredate  DATE,
   sal       NUMBER(7,2),
   comm      NUMBER(7,2),
   deptno,
   CONSTRAINT fk_deptno
      FOREIGN  KEY (deptno)
      REFERENCES  dept(deptno) );
```

Note that the foreign key definitions in both statements of this example omit the ON DELETE CASCADE option, causing Oracle to forbid the deletion of a department if any employee works in that department.

Maintaining Referential Integrity with ON DELETE CASCADE

If you use the ON DELETE CASCADE option, Oracle permits deletions of referenced key values in the parent table and automatically deletes dependent rows in the child table to maintain referential integrity.

Example. This example creates the EMP table, defines and enables the referential integrity constraint FK_DEPTNO, and uses the ON DELETE CASCADE option:

```
CREATE TABLE emp
  (empno    NUMBER(4),
   ename    VARCHAR2(10),
   job      VARCHAR2(9),
   mgr      NUMBER(4),
   hiredate DATE,
   sal      NUMBER(7,2),
   comm     NUMBER(7,2),
   deptno   NUMBER(2)   CONSTRAINT fk_deptno
            REFERENCES dept(deptno)
            ON DELETE CASCADE );
```

Because of the ON DELETE CASCADE option, Oracle cascades any deletion of a DEPTNO value in the DEPT table to the DEPTNO values of its dependent rows of the EMP table. For example, if Department 20 is deleted from the DEPT table, Oracle deletes the department's employees from the EMP table.

Referential Integrity Constraints with Composite Keys

A composite foreign key is a foreign key made up of a combination of columns. A composite foreign key can contain as many as 16 columns. To define a referential integrity constraint with a composite foreign key, you must use *table_constraint* syntax. You cannot use *column_constraint* syntax, because this syntax can impose rules only on a single column. A composite foreign key must refer to a composite unique key or a composite primary key.

To satisfy a referential integrity constraint involving composite keys, each row in the child table must satisfy one of the following conditions:

- The values of the foreign key columns must match the values of the referenced key columns in a row in the parent table.
- The value of at least one of the columns of the foreign key must be null.

Example. The following statement defines and enables a foreign key on the combination of the AREACO and PHONENO columns of the PHONE_CALLS table:

```
ALTER TABLE phone_calls
```

```
ADD CONSTRAINT fk_areaco_phoneno
    FOREIGN KEY (areaco, phoneno)
    REFERENCES customers(areaco, phoneno)
    EXCEPTIONS INTO wrong_numbers;
```

The constraint FK_AREACO_PHONENO ensures that all the calls in the PHONE_CALLS table are made from phone numbers that are listed in the CUSTOMERS table. Before you define and enable this constraint, you must define and enable a constraint that designates the combination of the AREACO and PHONENO columns of the CUSTOMERS table as a primary or unique key.

The EXCEPTIONS option causes Oracle to write information to the WRONG_NUMBERS table about any rows in the PHONE_CALLS table that violate the constraint.

CHECK Constraints

The CHECK constraint explicitly defines a condition. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null). For information on conditions, see the syntax description of *condition* in "Conditions" on page 3-90. The condition of a CHECK constraint can refer to any column in the table, but it cannot refer to columns of other tables. CHECK constraint conditions cannot contain the following constructs:

- queries to refer to values in other rows
- calls to the functions SYSDATE, UID, USER, or USERENV
- the pseudocolumns CURRVAL, NEXTVAL, LEVEL, or ROWNUM
- date constants that are not fully specified

Whenever Oracle evaluates a CHECK constraint condition for a particular row, any column names in the condition refer to the column values in that row.

If you create multiple CHECK constraints for a column, design them carefully so their purposes do not conflict. Oracle does not verify that CHECK conditions are not mutually exclusive.

Example I. The following statement creates the DEPT table and defines a CHECK constraint in each of the table's columns:

```
CREATE TABLE dept  (deptno NUMBER  CONSTRAINT check_deptno
          CHECK (deptno BETWEEN 10 AND 99)
          DISABLE,
dname VARCHAR2(9)  CONSTRAINT check_dname
          CHECK (dname = UPPER(dname))
```

```
          DISABLE,
loc VARCHAR2(10)  CONSTRAINT check_loc
          CHECK (loc IN ('DALLAS','BOSTON',
          'NEW YORK','CHICAGO'))
          DISABLE);
```

Each constraint restricts the values of the column in which it is defined:

CHECK_DEPTNO	ensures that no department numbers are less than 10 or greater than 99.
CHECK_DNAME	ensures that all department names are in uppercase.
CHECK_LOC	restricts department locations to Dallas, Boston, New York, or Chicago.

Because each CONSTRAINT clause contains the DISABLE option, Oracle only defines the constraints and does not enable them.

Unlike other types of constraints, a CHECK constraint defined with *column_constraint* syntax can impose rules on any column in the table, rather than only on the column in which it is defined.

Example II. The following statement creates the EMP table and uses a table constraint clause to define and enable a CHECK constraint:

```
CREATE TABLE emp
    (empno          NUMBER(4),
     ename          VARCHAR2(10),
     job            VARCHAR2(9),
     mgr            NUMBER(4),
     hiredate       DATE,
     sal            NUMBER(7,2),
     comm           NUMBER(7,2),
     deptno         NUMBER(2),
     CHECK (sal + comm <= 5000) );
```

This constraint uses an inequality condition to limit an employee's total compensation, the sum of salary and commission, to $5000:

- If an employee has non-null values for both salary and commission, the sum of these values must not be more than $5000 to satisfy the constraint.
- If an employee has a null salary or commission, 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.

Example III. The following statement defines and enables a PRIMARY KEY constraint, two referential integrity 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 scott.order (order_id),
   part_no            NUMBER
      CONSTRAINT fk_pno REFERENCES scott.part (part_no),
   quantity            NUMBER
      CONSTRAINT nn_qty NOT NULL
      CONSTRAINT check_qty_low 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, the following conditions must be true:

- No two rows in the table can contain the same combination of values in the ORDER_ID and the PART_NO columns.
- No row in the table can have a null in either the ORDER_ID column or the PART_NO column.

FK_OID — identifies the ORDER_ID column as a foreign key that references the ORDER_ID column in the ORDER table in SCOTT's schema. All new values added to the column ORDER_DETAIL.ORDER_ID must already appear in the column SCOTT.ORDER.ORDER_ID.

FK_PNO — identifies the PART_NO column as a foreign key that references the PART_NO column in the PART table owned by SCOTT. All new values added to the column ORDER_DETAIL.PART_NO must already appear in the column SCOTT.PART.PART_NO.

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:

- *Table_constraint* syntax and column definitions can appear in any order. In this example, note that the *table_constraint* syntax that defines the PK_OD constraint precedes the column definitions. In Example IV in this section, the *table_constraint* syntax defining the table's primary key follows the column definitions.
- A column definition can use *column_constraint* syntax multiple times. 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, is better than a single CHECK constraint with a complicated condition enforcing multiple business rules. When a constraint is violated, Oracle returns an error message identifying the constraint. Such an error message more precisely identifies the violated business rule if the identified constraint enables a single business rule.

DEFERRABLE Constraints

You can specify table and column constraints as DEFERRABLE or NOT DEFERRABLE. DEFERRABLE means that the constraint will not be checked until the transaction is committed. The default is NOT DEFERRABLE.

If you specify DEFERRABLE, you can also specify the constraint's initial state as INITIALLY DEFERRED and thereby start the transaction in DEFERRED mode. Or you can specify a DEFERRABLE constraint's initial state as INITIALLY IMMEDIATE and start the transaction in NOT DEFERRED mode.

Example I. The following statement creates table GAMES with a NOT DEFERRABLE INITIALLY IMMEDIATE constraint check on the SCORES column:

```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

Example III. To define a unique constraint on a column as INITIALLY DEFERRED DEFERRABLE, issue the following statement:

```
CREATE TABLE orders
```

```
(ord_num NUMBER CONSTRAINT unq_num UNIQUE (ord_num)
 INITIALLY DEFERRED DEFERRABLE);
```

A constraint cannot be defined as NOT DEFERRABLE INITIALLY DEFERRED.

Use SET CONSTRAINT(S) on page 4-514 to set, for a single transaction, whether a deferrable constraint is checked following each DML statement or when the transaction is committed. You cannot alter a constraint's deferrability status; you must drop the constraint and re-create it.

See *Oracle8 Administrator's Guide* and *Oracle8 Concepts* for more information about deferred constraints.

Enabling and Disabling Constraints

Constraints can have one of three states: DISABLE, ENABLE NOVALIDATE, or ENABLE VALIDATE.

Taking a constraint from a disabled to enable validated state requires an exclusive lock on the table, because while all old data is being checked for validity, no new data can be entered into the table. Due to this behavior, only one constraint can be enabled at a time, and each new constraint must check all existing rows by serial scan.

To avoid locking the table, place the constraint in the ENABLE NOVALIDATE state, using the ENABLE clause on page 4-417. This state ensures that all new DML statements on the table are validated, therefore Oracle does not need to prevent concurrent access to the table.

ENABLE NOVALIDATE also allows you to place several of the table's constraints in the ENABLE VALIDATE state concurrently. Each scan that Oracle performs to validate existing data can also be performed in parallel when possible.

Placing constraints concurrently in the ENABLE VALIDATE state requires that you issue multiple ALTER TABLE commands from separate sessions.

Enabling Primary Key and Unique Key Constraints

Enabling a primary key or unique key constraint automatically creates a unique index to enforce the constraint. This index is dropped if the constraint is subsequently disabled, thus causing Oracle to rebuild the index every time the constraint is enabled.

To avoid this behavior, create new primary key and unique key constraints initially disabled; then create nonunique indexes or use existing nonunique indexes to enforce the constraint. Because Oracle does not drop the nonunique index when the

constraint is disabled, any ENABLE operation on a primary key or unique key constraint occurs almost instantly, because the index already exists. Redundant indexes are also eliminated.

For more information about PRIMARY KEY and UNIQUE constraints, see the ENABLE clause on page 4-417.

Related Topics

CREATE TABLE on page 4-306
ALTER TABLE on page 4-106
ENABLE clause on page 4-417
DISABLE clause on page 4-380
SET CONSTRAINT(S) on page 4-514
ALTER SESSION on page 4-58

CREATE CLUSTER

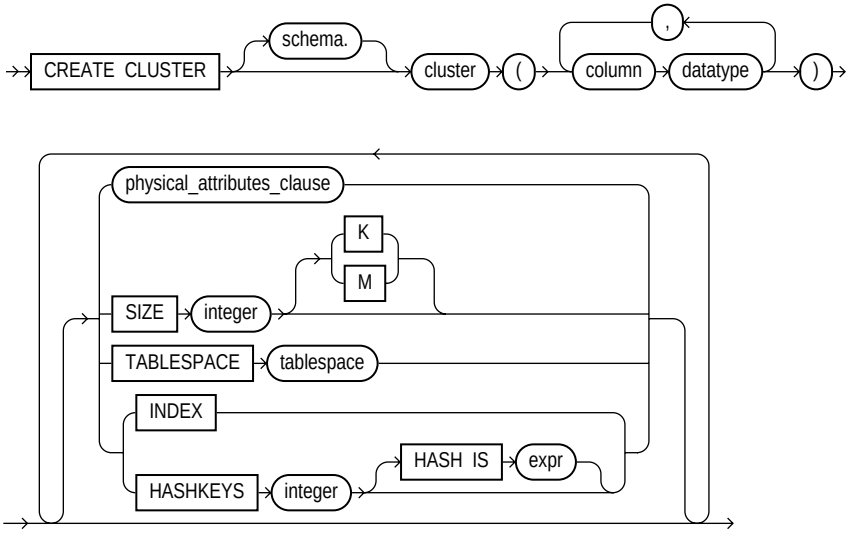
Purpose

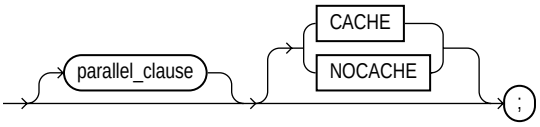
To create a cluster. A *cluster* is a schema object that contains one or more tables, all of which have one or more columns in common. See also "About Clusters" on page 4-210 and "Adding Tables to a Cluster" on page 4-213.

Prerequisites

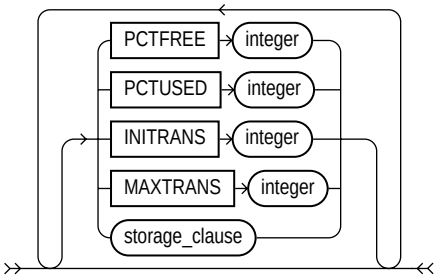
To create a cluster in your own schema, you must have CREATE CLUSTER system privilege. To create a cluster in another user's schema, you must have CREATE ANY CLUSTER system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or UNLIMITED TABLESPACE system privilege.

Syntax





physical_attributes_clause ::=



storage_clause: See the STORAGE clause on page 4-523.

parallel_clause: See the PARALLEL clause on page 4-465.

Keywords and Parameters

schema	is the schema to contain the cluster. If you omit *schema*, Oracle creates the cluster in your current schema.
cluster	is the name of the cluster to be created.
column	is the name of a column in the cluster key. See also "Cluster Keys" on page 4-210.
datatype	is the datatype of a cluster key column. A cluster key column can have any datatype except LONG or LONG RAW. You cannot use the HASH IS clause if any column datatype is not INTEGER or NUMBER with scale 0. For information on datatypes, see the section "Datatypes" on page 2-5.
physical_attributes_clause:	
PCTUSED	specifies the limit that Oracle uses to determine when additional rows can be added to a cluster's data block. The value of this parameter is expressed as a whole number and interpreted as a percentage.
PCTFREE	specifies the space reserved in each of the cluster's data blocks for future expansion. The value of the parameter is expressed as a whole number and interpreted as a percentage.

INITRANS	specifies the initial number of concurrent update transactions allocated for data blocks of the cluster. The value of this parameter for a cluster cannot be less than 2 or more than the value of the MAXTRANS parameter. The default value is the greater of the INITRANS value for the cluster's tablespace and 2.
MAXTRANS	specifies the maximum number of concurrent update transactions for any given data block belonging to the cluster. The value of this parameter cannot be less than the value of the INITRANS parameter. The maximum value of this parameter is 255. The default value is the MAXTRANS value for the tablespace to contain the cluster. For a complete description of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters, see CREATE TABLE on page 4-306.
SIZE	specifies the amount of space in bytes to store all rows with the same cluster key value or the same hash value. You can use K or M to specify this space in kilobytes or megabytes. If you omit this parameter, Oracle reserves one data block for each cluster key value or hash value. See also "Cluster Size" on page 4-212.
TABLESPACE	specifies the tablespace in which the cluster is created.
storage_clause	specifies how data blocks are allocated to the cluster. See the STORAGE clause on page 4-523.
INDEX	creates an indexed cluster. In an indexed cluster, rows are stored together based on their cluster key values. See also "Types of Clusters" on page 4-211.
HASHKEYS	creates a hash cluster and specifies the number of hash values for a hash cluster. Oracle rounds the HASHKEYS value up 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 option and the HASHKEYS parameter, Oracle creates an indexed cluster by default. See also "Types of Clusters" on page 4-211.
HASH IS	specifies a 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 datatype as long as the entire expression evaluates to a number of scale 0—for example, NUM_COLUMN * length(VARCHAR2_COLUMN) ■ cannot reference user-defined PL/SQL functions ■ cannot reference SYSDATE, USERENV, TO_DATE, UID, USER, LEVEL, ROWNUM ■ cannot evaluate to a constant ■ cannot contain a subquery ■ cannot contain columns qualified with a schema or object name (other than the cluster name) If you omit the HASH IS clause, Oracle uses an internal hash function for the hash cluster.

	The cluster key of a hash column can have one or more columns of any datatype. Hash clusters with composite cluster keys or cluster keys made up of noninteger columns must use the internal hash function.
parallel_clause	specifies the degree of parallelism to use when creating the cluster and the default degree of parallelism to use when querying the cluster after creation. See the PARALLEL clause on page 4-465.
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 option is useful for small lookup tables.
NOCACHE	specifies 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. This is the default behavior.

About Clusters

A *cluster* is a schema object that contains one or more tables that all have one or more columns in common. Rows of one or more tables that share the same value in these common columns are physically stored together within the database.

Clustering provides more control over the physical storage of rows within the database. Clustering can reduce both the time it takes to access clustered tables and the space needed to store the table. 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 nonclustered tables.

If you cannot fit all rows for one hash value into a data block, do not use hash clusters. Performance is very poor in this circumstance because an insert or update of a row in a hash cluster with a size exceeding the data block size fills the block and performs row chaining to contain the rest of the row.

Generally, you should only cluster tables that are frequently joined on the cluster key columns in SQL statements. Clustering multiple tables improves the performance of joins, but it is likely to reduce the performance of full table scans, INSERT statements, and UPDATE statements that modify cluster key values. Before clustering, consider its benefits and trade-offs in light of the operations you plan to perform on your data. For more information on the performance implications of clustering, see *Oracle8 Tuning*.

Cluster Keys

The columns defined by the CREATE CLUSTER command make up the *cluster key*. These cluster columns must correspond in both datatype 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.

Types of Clusters

A cluster can be either an indexed cluster or a hash cluster.

Indexed Clusters

In an *indexed cluster*, Oracle 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. This saves disk space and improves performance for many operations.

You may want to use indexed clusters in the following cases:

- Your queries retrieve rows over a range of cluster key values.
- Your clustered tables may grow unpredictably.

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*. For information on creating a cluster index, see CREATE INDEX on page 4-237. As with the columns of any index, the order of the columns in the cluster key affects the structure of the cluster index.

A cluster index provides quick access to rows within a cluster based on the cluster key. If you issue a SQL statement that searches for a row in the cluster based on its cluster key value, Oracle searches the cluster index for the cluster key value and then locates the row in the cluster based on its ROWID.

Hash Clusters

In a *hash cluster*, Oracle stores together rows that have the same hash key value. The *hash value* for a row is the value returned by the cluster's hash function. When you create a hash cluster, you can either specify a hash function or use the Oracle internal hash function. Hash values are not actually stored in the cluster, although cluster key values are stored for every row in the cluster.

You may want to use hash clusters in the following cases:

- Your queries retrieve rows based on equality conditions involving all cluster key columns.

- Your clustered tables are static or you can determine the maximum number of rows and the maximum amount of space required by the cluster when you create the cluster.

The hash function provides access to rows in the table based on the cluster key value. If you issue a SQL statement that locates a row in the cluster based on its cluster key value, Oracle applies the hash function to the given cluster key value and uses the resulting hash value to locate the matching rows. Because multiple cluster key values can map to the same hash value, Oracle must also check the row's cluster key value. This process often results in less I/O than the process for the indexed cluster, because the index search is not required.

Oracle's internal hash function returns values ranging from 0 to the value of HASHKEYS - 1. If you specify a column with the HASH IS clause, the column values need not fall into this range. Oracle divides the column value by the HASHKEYS value and uses the remainder as the hash value. The hash value for null is HASHKEYS - 1. Oracle also rounds the HASHKEYS value up to the nearest prime number to obtain the actual number of hash values. This rounding reduces the likelihood of *hash collisions*, or multiple cluster key values having the same hash value.

You cannot create a cluster index for a hash cluster, and you need not create an index on a hash cluster key.

Cluster Size

Oracle uses the value of the SIZE parameter to determine the space reserved for rows corresponding to one cluster key value or one hash value. This space then determines the maximum number of cluster or hash values stored in a data block. If the SIZE value is not a divisor of the data block size, Oracle uses the next largest divisor. If the SIZE value is larger than the data block size, Oracle uses the operating system block size, reserving at least one data block per cluster or hash value.

Oracle 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 does not apply to hash clusters because hash values are not actually stored in the cluster.

Although the maximum number of cluster and hash key values per data block is fixed on a per '-cluster basis, Oracle does not reserve an equal amount of space for each cluster or hash key value. Varying this space stores data more efficiently, because the data stored per cluster or hash key value is rarely fixed.

A SIZE value smaller than the space needed by the average cluster or hash key value may require the data for one cluster key or hash key value to occupy multiple data blocks. A SIZE value much larger results in wasted space.

When you create a hash cluster, Oracle immediately allocates space for the cluster based on the values of the SIZE and HASHKEYS parameters. For more information on how Oracle allocates space for clusters, see *Oracle8 Concepts*.

Adding Tables to a Cluster

You can add tables to an existing cluster by issuing a CREATE TABLE statement with the CLUSTER clause. A cluster can contain as many as 32 tables, although the performance gains of clustering are often lost in clusters of more than four or five tables.

All tables in the cluster have the cluster's storage characteristics as specified by the PCTUSED, PCTFREE, INITRANS, MAXTRANS, TABLESPACE, and STORAGE parameters.

Example I. The following statement creates an indexed cluster named PERSONNEL with the cluster key column DEPARTMENT_NUMBER, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
    ( department_number  NUMBER(2) )
    SIZE 512
    STORAGE (INITIAL 100K NEXT 50K PCTINCREASE 10);
```

The following statements add the EMP and DEPT tables to the cluster:

```
CREATE TABLE emp
    (empno     NUMBER        PRIMARY KEY,
     ename     VARCHAR2(10)  NOT NULL
                             CHECK (ename = UPPER(ename)),
     job       VARCHAR2(9),
     mgr       NUMBER        REFERENCES scott.emp(empno),
     hiredate  DATE          CHECK (hiredate >= SYSDATE),
     sal       NUMBER(10,2)  CHECK (sal > 500),
     comm      NUMBER(9,0)   DEFAULT NULL,
     deptno   NUMBER(2)      NOT NULL )
     CLUSTER personnel (deptno);

CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9),
```

```
    loc     VARCHAR2(9))
    CLUSTER personnel (deptno);
```

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 insert rows into either the EMP or DEPT tables.

Example II. The following statement creates a hash cluster named PERSONNEL with the cluster key column DEPARTMENT_NUMBER, a maximum of 503 hash key values, each of size 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
( department_number  NUMBER )
SIZE 512  HASHKEYS 500
    STORAGE (INITIAL 100K  NEXT 50K  PCTINCREASE 10);
```

Because the above statement omits the HASH IS clause, Oracle uses the internal hash function for the cluster.

Example III. The following statement creates a hash cluster named PERSONNEL with the cluster key made up of the columns HOME_AREA_CODE and HOME_PREFIX, and uses a SQL expression containing these columns for the hash function:

```
CREATE CLUSTER personnel
    ( home_area_code  NUMBER,
    home_prefix     NUMBER )
    HASHKEYS 20
    HASH IS MOD(home_area_code + home_prefix, 101);
```

Related Topics

CREATE INDEX on page 4-237
CREATE TABLE on page 4-306
“Index-Organized Tables” on page 4-125
STORAGE clause on page 4-523

CREATE CONTROLFILE

Purpose

To re-create a control file in one of the following cases:

- All copies of your existing control files have been lost through media failure.
- You want to change the name of the database.
- You want to change the maximum number of redo log file groups, redo log file members, archived redo log files, datafiles, or instances that can concurrently have the database mounted and open.

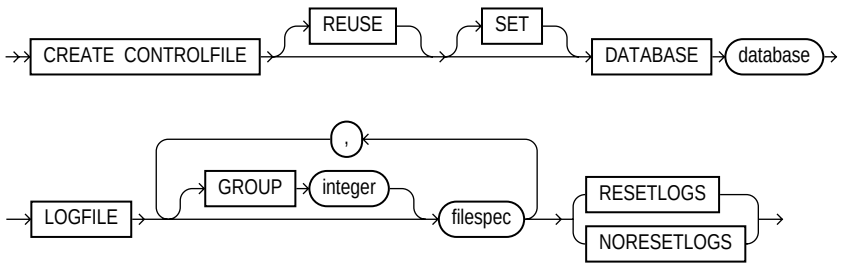
See also "Re-creating Control Files" on page 4-218.

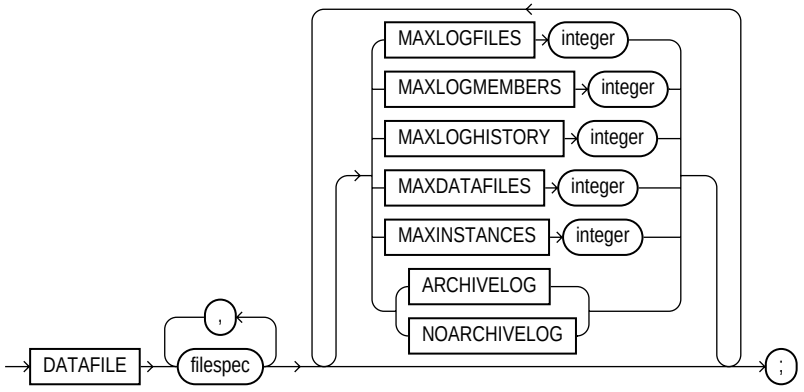
> **WARNING:** Oracle recommends that you perform a full backup of all files in the database before using this command.

Prerequisites

You must have the OSDBA role enabled. The database must not be mounted by any instance.

Syntax





filespec: See "Filespec" on page 4-431.

Keywords and Parameters

REUSE	specifies that existing control files identified by the initialization parameter CONTROL_FILES can be reused, thus ignoring and overwriting any information they may currently contain. If you omit this option and any of these control files already exists, Oracle returns an error.
DATABASE	specifies 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	changes the name of the database. The name of a database can be as long as eight bytes.
LOGFILE	specifies the redo log file groups for your database. You must list all members of all redo log file groups. See the syntax description of filespec in "Filespec" on page 4-431.
RESETLOGS	ignores the contents of the files listed in the LOGFILE clause. These files do not have to exist. Each filespec in the LOGFILE clause must specify the SIZE parameter. Oracle assigns all redo log file groups to thread 1 and enables this thread for public use by any instance. After using this option, you must open the database using the RESETLOGS option of the ALTER DATABASE command.
NORESETLOGS	specifies that all files in the LOGFILE clause should be used as they were when the database was last open. These files must exit and must be the current redo log files rather than restored backups. Oracle reassigns the redo log file groups to the threads to which they were previously assigned and reenables the threads as they were previously enabled. If you specify GROUP values, Oracle verifies these values with the GROUP values when the database was last open.

DATAFILE	specifies the datafiles of the database. You must list all datafiles. These files must all exist, although they may be restored backups that require media recovery. See the syntax description of filespec in “Filespec” on page 4-431.
MAXLOGFILES	specifies the maximum number of redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate 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. Note that the number of redo log file groups accessible to your instance is also limited by the initialization parameter LOG_FILES.
MAXLOGMEMBERS	specifies the maximum number of members, or copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log file. The minimum value is 1. The maximum and default values depend on your operating system.
MAXLOGHISTORY	specifies the maximum number of archived redo log file groups for automatic media recovery of the Oracle8 Parallel Server. Oracle uses this value to determine how much space in the control file to allocate for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCE value and depends on your operating system. The maximum value is limited only by the maximum size of the control file. This parameter is useful only if you are using Oracle with the Parallel Server option in both parallel mode and archivelog mode.
MAXDATAFILES	specifies the initial sizing of the datafiles 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 Oracle8 control file to expand automatically so that the datafiles section can accommodate more files. Note that the number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.
MAXINSTANCES	specifies 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	establishes the mode of archiving the contents of redo log files before reusing them. This option prepares for the possibility of media recovery as well as instance recovery.
NOARCHIVELOG	If you omit both the ARCHIVELOG and NOARCHIVELOG options, Oracle chooses noarchivelog mode by default. After creating the control file, you can change between archivelog mode and noarchivelog mode with the ALTER DATABASE command.

Re-creating Control Files

Oracle recommends that you take a full backup of all files in the database before issuing a CREATE CONTROLFILE statement.

When you issue a CREATE CONTROLFILE statement, Oracle creates a new control file based on the information you specify in the statement. If you omit any of the options from the statement, Oracle uses the default options, rather than the options for the previous control file. After successfully creating the control file, Oracle mounts the database in the mode specified by the initialization parameter PARALLEL_SERVER. You then must perform media recovery before opening the database. It is recommended that you then shutdown the instance and take a full backup of all files in the database.

For more information about using this command, see the *Oracle8 Administrator's Guide*.

Example. This example re-creates a control file:

```
CREATE CONTROLFILE REUSE
DATABASE orders_2
LOGFILE GROUP 1 ('diskb:log1.log', 'diskc:log1.log') SIZE 50K,
GROUP 2 ('diskb:log2.log', 'diskc:log2.log') SIZE 50K
NORESETLOGS
DATAFILE 'diska:dbone.dat' SIZE 2M
MAXLOGFILES 5
MAXLOGHISTORY 100
MAXDATAFILES 10
MAXINSTANCES 2
ARCHIVELOG;
```

Related Topics

CREATE DATABASE command on page 4-219
"Filespec" on page 4-431

CREATE DATABASE

Purpose

To create a database, making it available for general use, with the following options:

- to establish a maximum number of instances, datafiles, redo log files groups, or redo log file members
- to specify names and sizes of datafiles and redo log files
- to choose a mode of use for the redo log
- to specify the national and database character sets

For examples of some of these purposes, see “Examples” on page 4-223.

> **WARNING:** This command prepares a database for initial use and erases any data currently in the specified files. Use this command only when you understand its ramifications.

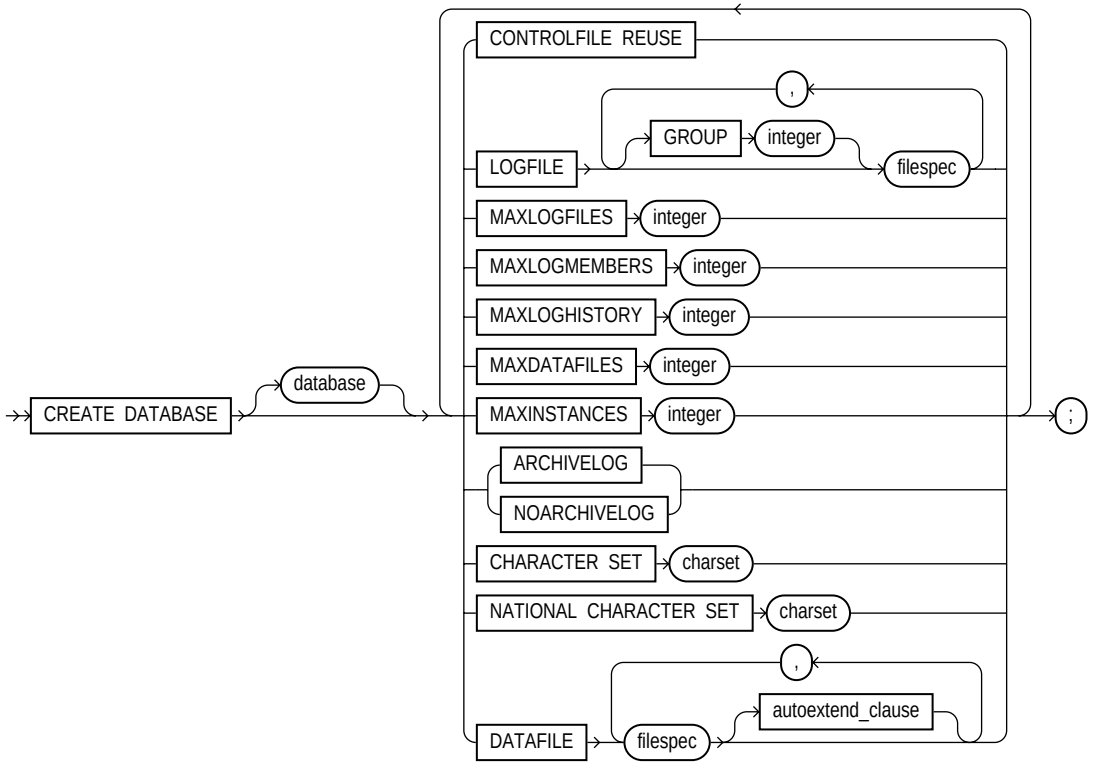
This command erases all data in any specified datafiles that already exist to prepare them for initial database use. If you use the command on an existing database, all data in the datafiles is lost.

After creating the database, this command mounts it in the mode specified by the PARALLEL_SERVER initialization parameter and opens it, making it available for normal use.

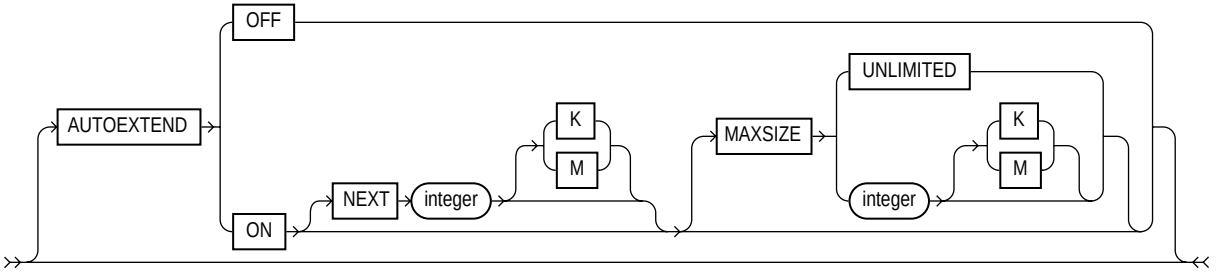
Prerequisites

You must have the OSDBA role enabled.

Syntax



autoextend_clause ::=



Keyword and Parameters

database is the name of the database to be created and can be up to eight bytes long. The database name can contain only ASCII characters. Oracle writes this name into the control file. If you subsequently issue an ALTER DATABASE statement and that explicitly specifies a database name, Oracle verifies that name with the name in the control file. Database names should also adhere to the rules described in "Schema Object Naming Rules" on page 2-47.

Note: You cannot use special characters from European or Asian character sets in a database name. For example, the umlaut is not allowed.

If you omit the database name from a CREATE DATABASE statement, Oracle uses the name specified by the initialization parameter DB_NAME.

CONTROLFILE REUSE reuses existing control files identified by the initialization parameter CONTROL_FILES, thus ignoring and overwriting any information they currently contain. Normally you use this option only when you are re-creating a database, rather than creating one for the first time. You cannot use this option 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 option and any of the files specified by CONTROL_FILES already exist, Oracle returns an error message.

LOGFILE specifies one or more files to be used as redo log files. Each *filespec* specifies a redo log file group containing one or more redo log file members, or copies. See the syntax description of filespec in "Filespec" on page 4-431. All redo log files specified in a CREATE DATABASE statement are added to redo log thread number 1.

GROUP uniquely identifies a redo log file group and can range from 1 to the value of the MAXLOGFILES parameter. You cannot specify multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance table V$LOG.

If you omit the LOGFILE clause, Oracle creates two redo log file groups by default. The names and sizes of the default files depends on your operating system.

MAXLOGFILES specifies the maximum number of redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default, minimum, and maximum values depends on your operating system.

The number of redo log file groups accessible to your instance is also limited by the initialization parameter LOG_FILES.

MAXLOGMEMBERS	specifies the maximum number of members, or copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.
MAXLOGHISTORY	specifies the maximum number of archived redo log files for automatic media recovery of Oracle with the Parallel Server option. Oracle uses this value to determine how much space in the control file to allocate 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. **Note:** This parameter is useful only if you are using Oracle with the Parallel Server option in parallel mode, and archivelog mode enabled.
MAXDATAFILES	specifies the initial sizing of the datafiles 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 Oracle8 control file to expand automatically so that the datafiles section can accommodate more files. Note that the number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.
MAXINSTANCES	specifies 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 and default values depend on your operating system.
ARCHIVELOG	establishes archivelog mode for redo log file groups. In this mode, the contents of a redo log file group must be archived before the group can be reused. This option prepares for the possibility of media recovery.
NOARCHIVELOG	establishes noarchivelog mode for redo log files groups. In this mode, the contents of a redo log file group need not be archived before the group can be reused. This option does not prepare 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 command.
CHARACTER SET	specifies the character set the database uses to store data. You cannot change the database character set after creating the database. The supported character sets and default value of this parameter depend on your operating system.

	You can specify any supported character set except the following fixed-width, multibyte character sets, which can be used only as the national character set: JA16SJISFIXED JA16EUCFIXED JA16DBCSFIXED For more information about valid character sets, see in the *Oracle8 Reference*.	
NATIONAL CHARACTER SET	specifies the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. You cannot change the national character set after creating the database. If not specified, the national character set defaults to the database character set. See *Oracle8 Reference* for valid character set names.	
DATAFILE	specifies one or more files to be used as datafiles. See the syntax description of *filespec* in "Filespec" on page 4-431. All these files become part of the SYSTEM tablespace. If you omit this clause, Oracle creates one datafile by default. The name and size of this default file depend on your operating system. **Note:** Oracle recommends that the total initial space allocated for the SYSTEM tablespace be a minimum of 5 megabytes.	
AUTOEXTEND	enables or disables the automatic extension of a datafile. If you do not specify this clause, datafiles are not automatically extended.	
	OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in ALTER DATABASE AUTOEXTEND or ALTER TABLESPACE AUTOEXTEND commands.
	ON	enables autoextend.
	NEXT	specifies the size in bytes of the next increment of disk space to be allocated to the datafile automatically when more extents are required. You can also use K or M to specify this size in kilobytes or megabytes. The default is one data block.
	MAXSIZE	specifies the maximum disk space allowed for automatic extension of the datafile.
	UNLIMITED	sets no limit on the allocation of disk space to the datafile.

Examples

Example I. The following statement creates a small database using defaults for all arguments:

```
CREATE DATABASE;
```

Example II. The following statement creates a database and fully specifies each argument:

```
CREATE DATABASE newtest
CONTROLFILE REUSE
LOGFILE
GROUP 1 ('diskb:log1.log', 'diskc:log1.log') SIZE 50K,
GROUP 2 ('diskb:log2.log', 'diskc:log2.log') SIZE 50K
MAXLOGFILES 5
MAXLOGHISTORY 100
DATAFILE 'diska:dbone.dat' SIZE 2M
MAXDATAFILES 10
MAXINSTANCES 2
ARCHIVELOG
CHARACTER SET US7ASCII
NATIONAL CHARACTER SET JA16SJISFIXED
DATAFILE
'disk1:df1.dbf' AUTOEXTEND ON
'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED;
```

Related Topics

ALTER DATABASE on page 4-15
CREATE ROLLBACK SEGMENT on page 4-275
CREATE TABLESPACE on page 4-328

CREATE DATABASE LINK

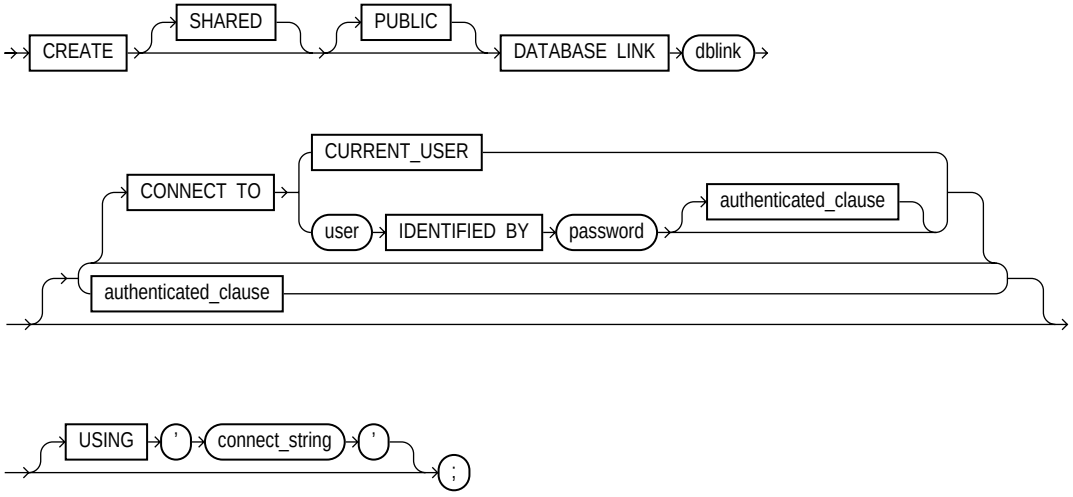
Purpose

To create a database link. A *database link* is a schema object in the local database that allows you to access objects on a remote database. The remote database can be either an Oracle or a non-Oracle system. See also "Creating Database Links" on page 4-226.

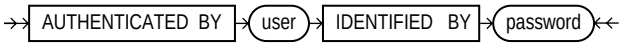
Prerequisites

To create a private database link, you must have CREATE DATABASE LINK system privilege. To create a public database link, you must have CREATE PUBLIC DATABASE LINK system privilege. Also, you must have CREATE SESSION privilege on the remote Oracle database. Net8 must be installed on both the local and remote Oracle databases. To access non-Oracle systems you must use the Oracle8 Heterogeneous Services.

Syntax



authenticated_clause ::=



Keyword and Parameters

SHARED	uses a single network connection to create a public database link that can be shared between multiple users. This option is available only with the multithreaded server configuration. For more information about shared database links, see *Oracle8 Distributed Database Systems*.
PUBLIC	creates a public database link available to all users. If you omit this option, the database link is private and is available only to you.
dblink	is the complete or partial name of the database link. For guidelines for naming database links, see "Referring to Objects in Remote Databases" on page 2-54 and "Current-User Database Links" on page 4-227.
CONNECT TO	enables a connection to the remote database.
CURRENT_USER	creates a current user database link. To use a current database link, the current user must be a global user authenticated by the Oracle Security Server. See also "Current-User Database Links" on page 4-227.
user IDENTIFIED BY *password*	is the username and password used to connect to the remote database (fixed user database link). If you omit this clause, the database link uses the username and password of each user who is connected to the database (connected user database link).
authenticated_clause	specifies 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. You must specify this clause when using the SHARED option.
USING '*connect string*'	specifies the service name of a remote database. For information on specifying remote databases, see *Net8 Administrator's Guide*.

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 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.

Once you have created a database link, you can use it to refer to tables and views on the remote database. You can refer to a remote table or view in a SQL statement by appending *@dblink* to the table or view name. You can query a remote table or view with the SELECT command. If you are using Oracle with the distributed option, you can also access remote tables and views using any of the following commands:

- DELETE on page 4-374
- INSERT on page 4-451
- LOCK TABLE on page 4-458
- UPDATE on page 4-542

See *Oracle8 Application Developer's Guide* for information about accessing remote tables or views with PL/SQL functions, procedures, packages, and datatypes.

The number of different database links that can appear in a single statement is limited to the value of the initialization parameter OPEN_LINKS.

Current-User Database Links

A current user database link is one that contains no user credentials and that enables a connection to a remote database as the current user. To use the link, the current user must be a global user with global accounts on both the local and remote databases. Both databases must be members of the same security domain.

To create a global user, see CREATE USER on page 4-357. For detailed information about current database links, see *Oracle8 Distributed Database Systems.*

CURRENT_USER

When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, CURRENT_USER is the username that created the stored object, and not the username 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, the current user is SCOTT.

If the database link is used directly, that is, NOT from within a stored object, then the current user is the same as the connected user.

Examples

Example I. The following example defines a current-user database link:

```
CREATE DATABASE LINK sales.hq.acme.com
CONNECT TO CURRENT_USER
USING 'sales';
```

Example II. The following statement defines a fixed-user database link named SALES.HQ.ACME.COM:

```
CREATE DATABASE LINK sales.hq.acme.com
CONNECT TO scott IDENTIFIED BY tiger
USING 'sales'
```

Once this database link is created, you can query tables in the schema SCOTT on the remote database in this manner:

```
SELECT *
FROM emp@sales.hq.acme.com
```

You can also use DML commands to modify data on the remote database:

```
INSERT INTO accounts@sales.hq.acme.com(acc_no, acc_name, balance)
VALUES (5001, 'BOWER', 2000)

UPDATE accounts@sales.hq.acme.com
SET balance = balance + 500

DELETE FROM accounts@sales.hq.acme.com
WHERE acc_name = 'BOWER'
```

You can also access tables owned by other users on the same database. This example assumes SCOTT has access to ADAM's DEPT table:

```
SELECT *
FROM adams.dept@sales.hq.acme.com
```

The previous statement connects to the user SCOTT on the remote database and then queries ADAM's DEPT table.

A synonym may be created to hide the fact that SCOTT's EMP table is on a remote database. The following statement causes all future references to EMP to access a remote EMP table owned by SCOTT:

```
CREATE SYNONYM emp
FOR scott.emp@sales.hq.acme.com;
```

Example III. The following statement defines a shared public fixed user database link named SALES.HQ.ACME.COM that refers to user SCOTT with password TIGER on the database specified by the string service name 'SALES':

```
CREATE SHARED PUBLIC DATABASE LINK sales.hq.acme.com
CONNECT TO scott IDENTIFIED BY tiger
AUTHENTICATED BY anupam IDENTIFIED BY bhide
USING 'sales';
```

Example IV. The following example creates a current user database link:

```
CREATE DATABASE LINK sales.hq.acme.com
CONNECT TO CURRENT_USER
USING 'sales';
```

Related Topics

CREATE SYNONYM on page 4-302
CREATE USER on page 4-357
DELETE on page 4-374
INSERT on page 4-451
LOCK TABLE on page 4-458
SELECT on page 4-489
UPDATE on page 4-542
PL/SQL User's Guide and Reference
Oracle8 Distributed Database Systems

CREATE DIRECTORY

Purpose

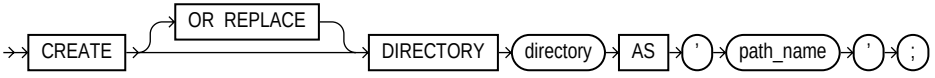
Use CREATE DIRECTORY to create a directory object, which represents an operating system directory for administering access to, and the use of, BFILEs stored outside the database. A *directory* is an alias for a full pathname on the server's file system where the files are actually located.

Prerequisites

You must have CREATE ANY DIRECTORY system privileges to create directories.

You must also create a corresponding operating system directory for file storage. Your system or database administrator must ensure that the operating system directory has the correct read permissions for Oracle processes.

Syntax



Keywords and Parameters

OR REPLACE — re-creates the directory database object if it already exists. You can use this option 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.

directory — is the name of the directory object to be created. The maximum length of *directory* is 30 bytes. You cannot qualify a directory object with a schema name. See also "Directory Objects" on page 4-231.

Note: Oracle 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 pathnames, be sure you specify the directory in the correct format. (However, you need not include a trailing slash at the end of the pathname.)

'path_name'	is the full pathname of the operating system directory on the server where the files are located. Note that the single quotes are required, with the result that the path name is case sensitive.

Directory Objects

A directory object specifies an alias name for a directory on the server's file system where external binary file LOBs (BFILEs) are located. You can use directory names when referring to BFILEs in your PL/SQL code and OCI calls, rather than hard-coding the operating system pathname, thereby allowing greater file management flexibility.

The Oracle BFILE datatype provides access to the external file system. A BFILE column or attribute contains a locator to an external file on the operating system, rather than the file itself. The locator maintains the directory alias and the filename.

All directories are created in a single namespace and are not owned by an individual's schema. You can secure access to the BFILEs stored within the directory structure by granting object privileges on the directories to specific users. When you create a directory, you are automatically granted the READ object privilege and can grant READ privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

Privileges granted for the directory are created independently of the permissions defined for the operating system directory; therefore, the two may or may not correspond exactly. For example, an error occurs if user SCOTT is granted READ privilege on the directory schema object, but the corresponding operating system directory does not have READ permission defined for Oracle processes.

Example. The following statement redefines directory database object BFILE_DIR to enable access to BFILEs stored in the operating system directory /PRIVATE1/LOB/FILES:

```
CREATE OR REPLACE DIRECTORY bfile_dir AS '/private1/LOB/files';
```

Related Topics

DROP DIRECTORY on page 4-389
GRANT (System Privileges and Roles) on page 4-434
"Large Object (LOB) Datatypes" on page 2-15

CREATE FUNCTION

Purpose

To create a *stored function* or to register an *external function*.

A *stored function* (also called a *user 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. For a general discussion of procedures and functions, see CREATE PROCEDURE on page 4-259. For examples of creating functions, see "Examples" on page 4-235

An external function is a third-generation language (3GL) routine stored in a shared library that can be called from SQL or PL/SQL. To call an external function, you must provide information in your PL/SQL function about where to find the external function, how to call it, and what to pass to it.

The CREATE FUNCTION command creates a function as a standalone schema object. You can also create a function as part of a package. For information on creating packages, see CREATE PACKAGE on page 4-250.

For more information about registering external functions, see the *PL/SQL User's Guide and Reference*.

Prerequisites

Before a stored function can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

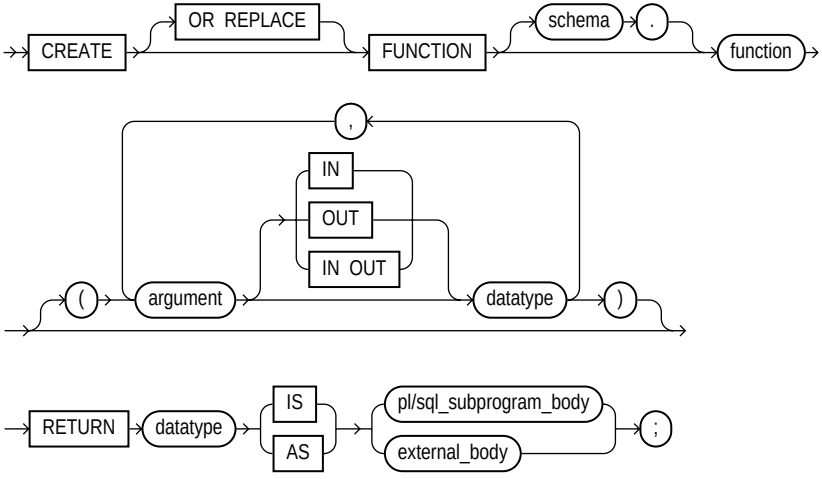
To create a function in your own schema, you must have CREATE PROCEDURE system privilege. To create a function in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To call an external function, you must have EXECUTE privileges on the callout library in which the function resides.

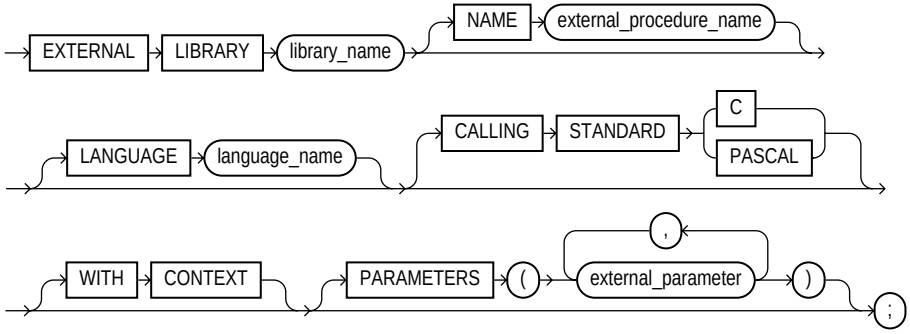
To create a stored function, you must be using Oracle with PL/SQL installed. For more information, see *PL/SQL User's Guide and Reference*.

To embed a CREATE FUNCTION 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.

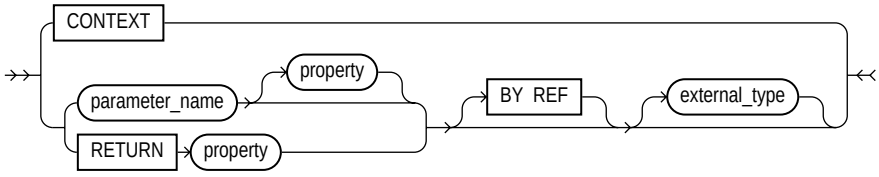
Syntax

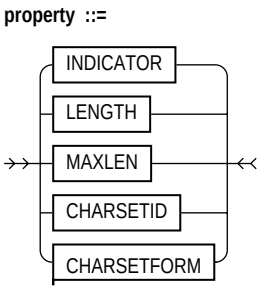


external_body::=



external_parameter ::=





Keywords and Parameters

OR REPLACE	re-creates the function if it already exists. Use this option 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, Oracle recompiles it. For information on recompiling functions, see ALTER FUNCTION on page 4-26. Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.
schema	is the schema to contain the function. If you omit *schema*, Oracle creates the function in your current schema.
function	is the name of the function to be created.
argument	is the name of an argument to the function. If the function does not accept arguments, you can omit the parentheses following the function name.
IN	specifies that you must supply a value for the argument when calling the function. This is the default.
OUT	specifies the function will set the value of the argument.
IN OUT	specifies that a value for the argument can be supplied by you and may be set by the function.
datatype	is the datatype of an argument. An argument can have any datatype supported by PL/SQL. The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of an argument from the environment from which the function is called.
RETURN *datatype*	specifies the datatype of the function's return value. Because every function must return a value, this clause is required. The return value can have any datatype supported by PL/SQL.

	The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of the return value from the environment from which the function is called. For information on PL/SQL datatypes, see *PL/SQL User's Guide and Reference.*
pl/sql_ subprogram_body	is the definition of the function. Function definitions are writing in PL/SQL. For information on PL/SQL, see *PL/SQL User's Guide and Reference.*
external_body_ clause	identifies the external function to be registered.
AS EXTERNAL	identifies an external 3GL function stored in a shareable library. The AS EXTERNAL clause is the interface between PL/SQL and the external function.
LIBRARY	specifies the shared library in which the external function is stored. You must have EXECUTE privileges on the library. See CREATE LIBRARY on page 4-248 for the syntax.
library_name	is a PL/SQL identifier. Enclosing *library_name* in double quotes makes it case sensitive, but quotes are not required.
NAME *external_ function_name*	specifies the external function to be called. Enclosing *external_function_name* in double quotes makes it case sensitive, but quotes are not required. If you omit the name, it defaults to the PL/SQL subprogram (uppercase) name.
LANGUAGE	specifies the 3GL in which the external function was written. Currently, the only language name supported is C. If you omit the name, it defaults to C.
CALLING STANDARD	specifies the calling standard (C or Pascal) under which the external function was compiled. If you omit the calling standard, it defaults to C.
WITH CONTEXT	specifies that a context pointer will be the first parameter passed to the external function. The context is opaque to the external function but is available to access functions called by the external function. For more information about the WITH CONTEXT clause, see *PL/ SQL User's Guide and Reference*.
PARAMETERS	specifies the positions and datatypes of parameters passed to the external function. It can also specify parameter properties such as current length and maximum length, and the preferred parameter passing method (by value or by reference). For more information about parameter passing see *PL/SQL User's Guide and Reference*.

Examples

Example I. The following statement creates the function GET_BAL:

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
RETURN NUMBER
IS
acc_bal NUMBER(11,2);
```

```
BEGIN
SELECT balance
INTO acc_bal
FROM accounts
WHERE account_id = acc_no;
RETURN(acc_bal);
END;
```

The GET_BAL function returns the balance of a specified account.

When you call the function, you must specify the argument ACC_NO, the number of the account whose balance is sought. The datatype of ACC_NO is NUMBER.

The function returns the account balance. The RETURN clause of the CREATE FUNCTION statement specifies the datatype of the return value to be NUMBER.

The function uses a SELECT statement to select the BALANCE column from the row identified by the argument ACC_NO in the ACCOUNTS table. The function uses a RETURN statement to return this value to the environment in which the function is called.

The function created above can be used in a SQL statement. For example:

```
SELECT get_bal(100) FROM DUAL;
```

Example II. The following statement creates PL/SQL standalone function GET_VAL that registers the C routine C_GET_VAL as an external function:

```
CREATE FUNCTION get_val
( x_val IN BINARY_INTEGER,
y_val IN BINARY_INTEGER,
image IN LONG RAW )
RETURN BINARY_INTEGER AS EXTERNAL LIBRARY c_utils
NAME "c_get_val"
LANGUAGE C;
```

Related Topics

ALTER FUNCTION on page 4-26
CREATE LIBRARY on page 4-248
CREATE PACKAGE on page 4-250
CREATE PACKAGE BODY on page 4-254
CREATE PROCEDURE on page 4-259
DROP FUNCTION on page 4-390
PL/SQL User's Guide and Reference

CREATE INDEX

Purpose

To create an index on

- one or more columns of a table, a partitioned table, or a cluster
- OBJ one or more scalar typed object attributes of a table or a cluster
- OBJ 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. A *partitioned index* consists of partitions containing an entry for each value that appears in the indexed column(s) of the table. See also "Creating Indexes" on page 4-243.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

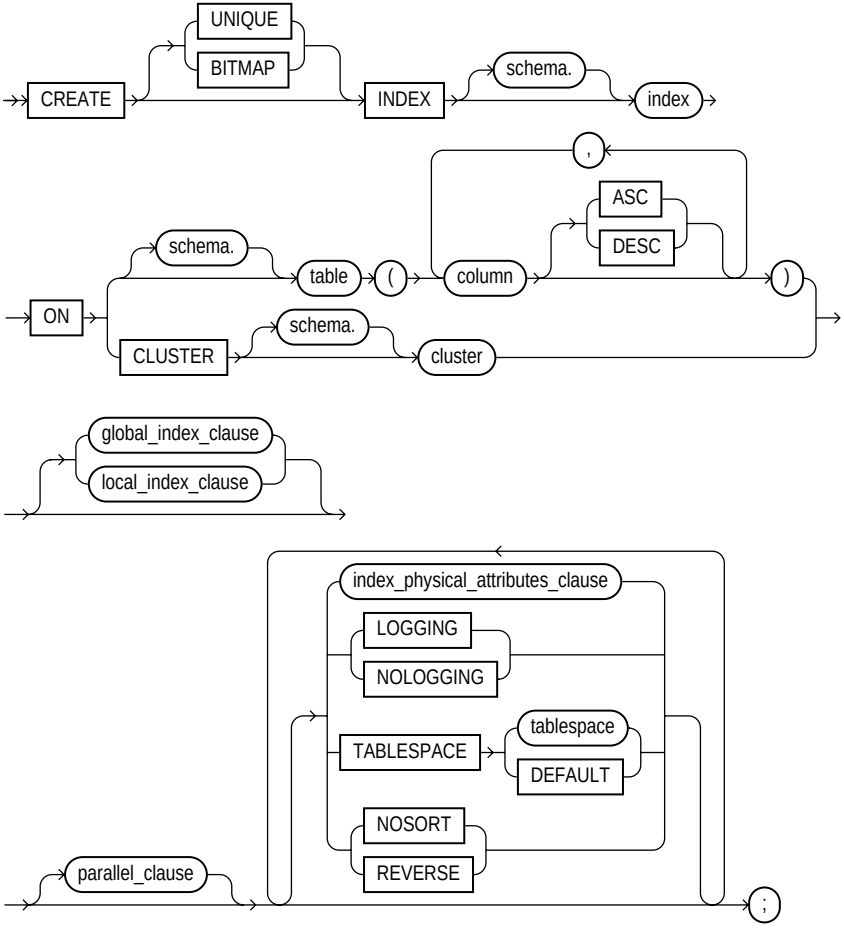
- The table or cluster to be indexed must be in your own schema.
- You must have INDEX privilege on the table to be indexed.
- You must have CREATE ANY INDEX system privilege.

To create an index in another schema, you must have CREATE ANY INDEX system privilege.

Also, the owner of the schema to contain the index must have either space quota on the tablespaces to contain the index or index partitions, or UNLIMITED TABLESPACE system privilege.

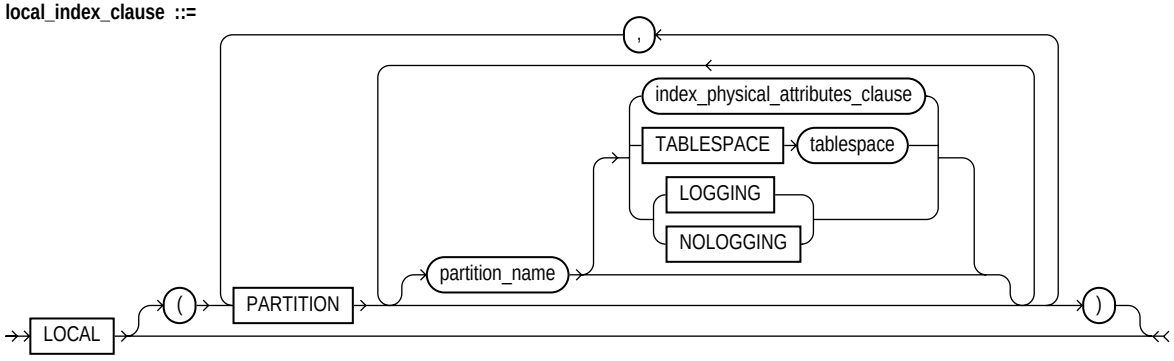
See also "Index Columns" on page 4-243.

Syntax

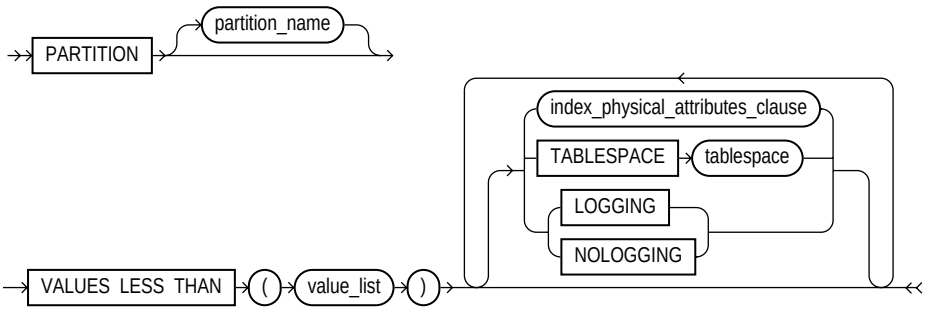


global_index_clause ::=

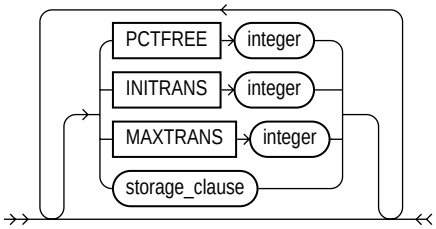
local_index_clause ::=



global_partition_clause ::=



index_physical_attributes_clause ::=



parallel_clause: See PARALLEL clause on page 4-465.

storage_clause: See STORAGE clause on page 4-523

Keywords and Parameters

UNIQUE — specifies that the value of the column (or combination of columns) in the table to be indexed must be unique.

If the index is local nonprefixed (see LOCAL clause below), then the index key must contain the partitioning key.

BITMAP — specifies that *index* is to be created as a bitmap, rather than as a B-tree. See also "Creating Bitmap Indexes" on page 4-246.

Note: You cannot use this keyword when creating a global partitioned index.

You can specify either UNIQUE or BITMAP, but you cannot create a unique bitmap index.

schema — is the schema to contain the index. If you omit schema, Oracle creates the index in your own schema.

index — is the name of the index to be created. (See also "Multiple Indexes Per Table" on page 4-244.) An *index* can contain several partitions.

You cannot range partition a cluster index or an index defined on a clustered table.

table — is the name of the table for which the index is to be created. If you do not qualify table with *schema*, Oracle assumes the table is contained in your own schema.

If the index is LOCAL, then *table* must be partitioned.

You cannot create an index on an index-organized table.

OBJ You can create an index on a nested table storage table.

column — is the name of a column in the table. An index can have as many as 32 columns. A column of an index cannot be of datatype LONG or LONG RAW. See also "Index Columns" on page 4-243.

OBJ 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, the column name must be qualified with the table name. If you specify a nested table column attribute, 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. See also "Creating Indexes on Nested Table Columns" on page 4-247.

See also "Nulls" on page 4-245.

ASC / DESC — are allowed for DB2 syntax compatibility, although indexes are always created in ascending order. Indexes on character data are created in ascending order of the character values in the database character set.

CLUSTER — specifies the cluster for which a cluster index is to be created. If you do not qualify cluster with *schema*, Oracle assumes the cluster is contained in your current schema. You cannot create a cluster index for a hash cluster. See also "Creating Cluster Indexes" on page 4-245.

index_physical_ attributes_clause	establishes values for the INITRANS, MAXTRANS, and PCTFREE parameters and storage characteristics for the index. See CREATE TABLE on page 4-306.
PCTFREE	is the percentage of space to leave free for updates and insertions within each of the index's data blocks.
storage_clause	establishes the storage characteristics for the index. See the STORAGE clause on page 4-523.
TABLESPACE	is the name of the tablespace to hold the index or index partition. If you omit this option, Oracle creates the index in the default tablespace of the owner of the schema containing the index. For a partitioned index, this is the tablespace name. For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions added to the local index will be created in the same tablespace(s) as the corresponding partition(s) of the underlying table.
NOSORT	indicates to Oracle that the rows are stored in the database in ascending order; therefore Oracle does not have to sort the rows when creating the index. You cannot specify REVERSE with this option. See also "The NOSORT Option" on page 4-244.
REVERSE	stores the bytes of the index block in reverse order, excluding the ROWID. You cannot specify NOSORT with this option. You cannot reverse a bitmap index.
LOGGING / NOLOGGING	specifies that the creation of the index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. It also specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against the index are logged or not logged. LOGGING is the default. If *index* is nonpartitioned, this is the logging attribute of the index. For partitioned index, the logging attribute specified is the default physical attribute of the segments associated with the index partitions. The default logging value applies to all partitions specified in the CREATE statement (and on subsequent ALTER TABLE ADD PARTITION statements) unless you specify LOGGING / NOLOGGING in the PARTITION description clause. 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, since the redo data is not logged. Thus if you cannot afford to lose this index, it is important to take a backup after the NOLOGGING operation. If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation will re-create the index. However, media recovery from a backup taken before the NOLOGGING operation will not re-create the index.

	The logging attribute of the index is independent of that of its base table. If the [NO]LOGGING clause is omitted, the logging attribute of the index defaults to the logging attribute of the tablespace in which it resides. For more information about the LOGGING option and Parallel DML, see "NOLOGGING" on page 4-245, *Oracle8 Concepts* and *Oracle8 Parallel Server Concepts and Administration*.
GLOBAL	specifies that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes.
PARTITION BY RANGE	specifies that the global index is partitioned on the ranges of values from the columns specified in *column_list*. You cannot specify this clause for a LOCAL index.
(column_list)	is the name of the column(s) of a table on which the index is partitioned. The *column_list* must specify a left prefix of the index column list. You cannot specify more than 32 columns in *column_list*, and the columns cannot contain the ROWID pseudocolumn or a column of type ROWID.
LOCAL	specifies that the index is range partitioned on the same columns, with the same number of partitions, and the same partition bounds as *table*. Oracle automatically maintains LOCAL index partitioning as the underlying table is repartitioned.
PARTITION *partition_name*	describes the individual partitions. The number of clauses determines the number of partitions. If the index is local, the number of index partitions must be equal to the number of the table partitions, and in the same order. The *partition_name* is the name of the physical index partition. If *partition_name* is omitted, Oracle generates a name with the form SYS_P*n*. For local indexes, if *partition_name* is omitted, Oracle generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, the form SYS_P*n* is used. See also "Creating Partitioned Indexes" on page 4-246.
VALUES LESS THAN (*value_list*)	specifies the (noninclusive) upper bound for the current partition in a global index. The *value_list* is a comma-separated, ordered list of literal values corresponding to *column_list* in the PARTITION BY RANGE clause. Always specify MAXVALUE as the *value_list* of the last partition. You cannot specify this clause for a local index.
parallel_clause	specifies the degree of parallelism for creating the index. See the PARALLEL clause on page 4-465.

Creating Indexes

An index is an ordered list of all the values that reside in a group of one or more columns at a given time. Such a list makes queries that test the values in those columns vastly more efficient. However, indexes take up data storage space and must be changed whenever the data is changed. Therefore, you should make a cost-benefit analysis in each case to determine whether and how indexes should be used. Oracle can use indexes to improve performance when:

- searching for rows with specified index column values
- accessing tables in index column order

When you initially insert rows into a new table, it is generally faster to create the table, insert the rows, and then create the index. If you create the index before inserting the rows, Oracle must update the index for every row inserted.

Oracle recommends that you do not explicitly define UNIQUE indexes on tables; uniqueness is strictly a logical concept and should be associated with the definition of a table. Instead, define UNIQUE integrity constraints on the desired columns. Oracle enforces UNIQUE integrity constraints by automatically defining a unique index on the unique key. Exceptions to this recommendation are usually performance related. For example, using a CREATE TABLE ... AS SELECT with a UNIQUE constraint is very much slower than creating the table without the constraint and then manually creating the UNIQUE index.

If indexes contain NULLs, the NULLS generally are considered distinct values. There is, however, one exception: if all the non-NULL values in two or more rows of an index are identical, the rows are considered identical; therefore, UNIQUE indexes prevent this from occurring. This does not apply if there are no non-NULL values—in other words, if the rows are entirely NULL.

Note: You cannot create an index on columns or attributes whose type is user-defined, LOB, or REF. The only exception is that Oracle supports creation of an index on REF type columns or attributes that have been defined with a SCOPE clause.

Index Columns

An index can contain a maximum of 32 columns. The index entry becomes the concatenation of all data values from each column. You can specify the columns in any order. The order you choose is important to how Oracle uses the index.

When appropriate, Oracle uses the entire index or a leading portion of the index. Assume an index named IDX1 is created on columns A, B, and C of table TAB1 (in the order A, B, C). Oracle uses the index for references to columns A, B, C (the entire index); A, B; or just column A. References to columns B and C do not use the IDX1 index. Of course, you can also create another index just for columns B and C.

Multiple Indexes Per Table

You can create unlimited indexes for a table provided that the combination of columns differs for each index. You can create more than one index using the same columns provided that you specify distinctly different combinations of the columns. For example, the following statements specify valid combinations:

```
CREATE INDEX emp_idx1 ON emp (ename, job);
CREATE INDEX emp_idx2 ON emp (job, ename);
```

You cannot create an index that references only one column in a table if another such index already exists.

Note that each index increases the processing time needed to maintain the table during updates to indexed data. Thus, updating a table with a single index will take less time than if the table had five indexes.

The NOSORT Option

The NOSORT option can substantially reduce the time required to create an index. Normal index creation first sorts the rows of the table based on the index columns and then builds the index. The sort operation is often a substantial portion of the total work involved. If the rows are already physically stored in ascending order (based on the indexed column values), then the NOSORT option causes Oracle to bypass the sort phase of the process.

You cannot use the NOSORT option to create a cluster index, partitioned index, or a bitmap index.

The NOSORT option also reduces the amount of space required to build the index. Oracle uses temporary segments during the sort. Since a sort is not performed, the index is created with much less temporary space.

To use the NOSORT option, you must guarantee that the rows are physically sorted in ascending order. However, you run no risk by trying the NOSORT option. If your rows are not in the ascending order, Oracle returns an error. You can issue another CREATE INDEX without the NOSORT option. Because of the physical data independence inherent in relational database management systems, especially Oracle, there is no way to force a physical internal order on a table. The CREATE

INDEX command with the NOSORT option should be used immediately after the initial load of rows into a table.

NOLOGGING

The NOLOGGING option may substantially reduce the time required to create a large index. This feature is particularly useful after creating a large index in parallel. For backup and recovery considerations, see *Oracle8 Backup and Recovery Guide* and *Oracle8 Administrator's Guide.*

Example. To quickly create an index in parallel on a table that was created using a fast parallel load (so all rows are already sorted), you might issue the following statement:

```
CREATE INDEX i_loc
ON big_table (akey)
NOSORT
NOLOGGING
PARALLEL (DEGREE 5);
```

Nulls

Oracle does not index table rows in which all key columns are NULL, except in the case of bitmap indexes.

Example. Consider the following statement:

```
SELECT ename
FROM emp
WHERE comm IS NULL;
```

The above query does not use an index created on the COMM column unless it is a bitmap index.

Creating Cluster Indexes

Oracle does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language statements cannot be issued against clustered tables until a cluster index has been created.

Example. To create an index for the EMPLOYEE cluster, issue the following statement:

```
CREATE INDEX ic_emp
ON CLUSTER employee
```

Note that no index columns are specified, because the index is automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

Creating Partitioned Indexes

Indexes can be local prefixed (unique or nonunique), local nonprefixed (unique, but only when the partitioning key is a subset of the index key or nonunique), or global prefixed (unique or nonunique). Oracle does not support global nonprefixed indexes. Local indexes are always partitioned. Global indexes can be nonpartitioned or partitioned.

Index partitions must be listed in order. For a global index, this means that the partition bound of the first partition listed must be *less than* the partition bound of the second partition listed, and the partition bound of the second partition listed must be *less than* the third, and so on. For a local index, you must list the partitions in the same order as the partitions of the underlying table to which they correspond.

Example. The following statement creates a global prefixed index STOCK_IX on table STOCK_XACTIONS with two partitions, one for each half of the alphabet. The index partition names are system generated:

```
CREATE INDEX stock_ix ON stock_xactions
  (stock_symbol, stock_series)
   GLOBAL PARTITION BY RANGE (stock_symbol)
     (PARTITION VALUES LESS THAN ('N') TABLESPACE ts3,
      PARTITION VALUES LESS THAN (MAXVALUE) TABLESPACE ts4);
```

Creating Bitmap Indexes

Bitmap indexes store the ROWIDs associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible ROWID, and if the bit is set, 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. See *Oracle8 Concepts* and *Oracle8 Tuning* for more information about using bitmap indexes.

Example. To create a bitmap partitioned index on a table with four partitions, issue the following statement:

```
CREATE BITMAP INDEX partno_ix
ON lineitem(partno)
TABLESPACE ts1
```

```
LOCAL (PARTITION quarter1 TABLESPACE ts2,
PARTITION quarter2 STORAGE (INITIAL 10K NEXT 2K),
PARTITION quarter3 TABLESPACE ts2,
PARTITION quarter4);
```

You cannot create bitmap indexes, unique bitmap indexes, or global partitioned indexes.

OBJ Creating Indexes on Nested Table Columns

Creating a table with nested table columns implicitly creates a storage table for each nested table column. The storage table stores the rows of the nested table values and the nested table identifier values assigned to each row. These identifier values are contained in a storage table pseudocolumn called NESTED_TABLE_ID.

You create an index on a nested table column by creating the index on the nested table storage table. You can include the NESTED_TABLE_ID pseudocolumn to create a UNIQUE index, which effectively ensures that the rows of a nested table value are distinct.

Example. In the following example, UNIQUE index UNIQ_PROJ_INDX is created on storage table NESTED_PROJECT_TABLE. Including pseudocolumn NESTED_TABLE_ID ensures distinct rows in nested table column PROJS_MANAGED:

```
CREATE TYPE proj_table_type AS TABLE OF proj_type;

CREATE TABLE employee ( emp_num NUMBER, emp_name CHAR(31),
projs_managed proj_table_type )
NESTED TABLE projs_managed STORE AS nested_project_table;

CREATE UNIQUE INDEX uniq_proj_indx
ON nested_project_table ( NESTED_TABLE_ID, proj_num);
```

Related Topics

ALTER INDEX on page 4-28
CREATE TABLE on page 4-306
“Index-Organized Tables” on page 4-125
DROP INDEX on page 4-392
CONSTRAINT clause on page 4-188
STORAGE clause on page 4-523

CREATE LIBRARY

Purpose

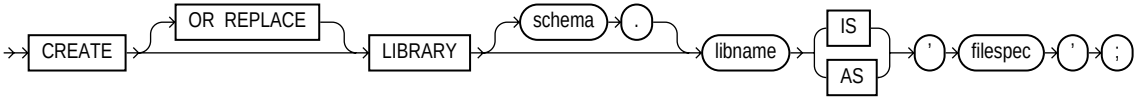
To create a schema object (*library*), which represents an operating-system shared library, from which SQL and PL/SQL can call external third-generation-language (3GL) functions and procedures. See "Examples" on page 4-249.

Prerequisites

You must have CREATE ANY LIBRARY system privileges. To use the procedures and functions stored in the library, you must have EXECUTE object privileges on the library.

The CREATE LIBRARY command is valid only on platforms that support shared libraries and dynamic linking.

Syntax



filespec: See "Filespec" on page 4-431.

Keywords and Parameters

OR REPLACE	re-creates the library if it already exists. Use this option to change the definition of an existing library without dropping, re-creating, and regranting schema 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.
libname	is the name of the library (schema object) from which SQL and PL/SQL will call external 3GL functions and procedures.
'*filespec*'	is a non-zero-length string, enclosed in single quotes. The '*filespec*' is not interpreted by PL/SQL. The directory and filename specified in '*filespec*' are not interpreted by PL/SQL; therefore the existence of the specification is not checked until procedure run time.

Examples

Example I. The following statement creates library EXT_LIB:

```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
```

Example II. The following example re-creates library EXT_LIB:

```
CREATE OR REPLACE ext_lib IS '/OR/newlib/ext_lib.so';
```

Related Topics

CREATE FUNCTION on page 4-232
CREATE PROCEDURE on page 4-259
PL/SQL User's Guide and Reference

CREATE PACKAGE

Purpose

To create the specification for a stored package. A *package* is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The *specification* declares these objects.

Prerequisites

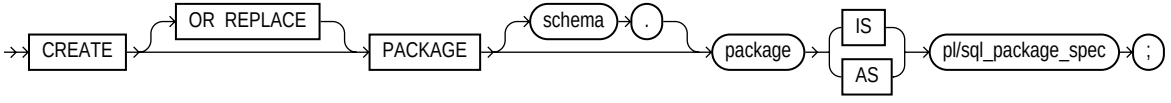
Before a package can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE 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.

For more information, see *PL/SQL User's Guide and Reference*.

Syntax



Keywords and Parameters

OR REPLACE re-creates the package specification if it already exists. Use this option 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, Oracle recompiles it. For information on recompiling package specifications, see ALTER PROCEDURE on page 4-41.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

schema	is the schema to contain the package. If you omit *schema,* Oracle creates the package in your own schema.
package	is the name of the package to be created. See also "Packages" on page 4-251.
pl/sql_package_spec	is the package specification. The package specification can declare program objects. Package specifications are written in PL/SQL. For information on PL/SQL, including writing package specifications, see *PL/SQL User's Guide and Reference*.

Packages

A *package* is an encapsulated collection of related program objects stored together in the database. Program objects are: procedures, functions, variables, constants, cursors, and exceptions.

Using packages is an alternative to creating procedures and functions as standalone schema objects. Packages have many advantages over stand-alone procedures and functions. They:

- allow you to organize your application development more efficiently.
- allow you to grant privileges more efficiently.
- allow you to modify package objects without recompiling dependent schema objects.
- enable Oracle to read multiple package objects into memory at once.
- can contain global variables and cursors that are available to all procedures and functions in the package.
- allow you to overload procedures or functions. *Overloading* a procedure means creating multiple procedures with the same name in the same package, each taking arguments of different number or datatype.

For more information on these and other benefits of packages, see *Oracle8 Application Developer's Guide.*

How to Create Packages

To create a package, you must perform two distinct steps:

1. **Create the package specification with the CREATE PACKAGE command.** You can declare program objects in the package specification. Such objects are called *public* objects. Public objects can be referenced outside the package as well as by other objects in the package.
2. **Create the package body with the CREATE PACKAGE BODY command.** You can declare and define program objects in the package body:

- You must define public objects declared in the package specification.
- You can also declare and define additional package objects. Such objects are called *private* objects. Private objects are declared in the package body rather than in the package specification, so they can be referenced only by other objects in the package. They cannot be referenced outside the package.

See CREATE PACKAGE BODY on page 4-254.

The Separation of Specification and Body

Oracle stores the specification and body of a package separately in the database. Other schema objects that call or reference public program objects depend only on the package specification, not on the package body. This distinction allows you to change the definition of a program object in the package body without causing Oracle to invalidate other schema objects that call or reference the program object. Oracle invalidates dependent schema objects only if you change the declaration of the program object in the package specification.

Example. This SQL statement creates the specification of the EMP_MGMT package:

```
CREATE PACKAGE emp_mgmt AS
   FUNCTION hire(ename VARCHAR2, job VARCHAR2, mgr NUMBER,
                 sal NUMBER, comm NUMBER, deptno NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
      RETURN NUMBER;
   PROCEDURE remove_emp(empno NUMBER);
   PROCEDURE remove_dept(deptno NUMBER);
   PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER);
   PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER);
      no_comm EXCEPTION;
      no_sal EXCEPTION;
END emp_mgmt;
```

The specification for the EMP_MGMT package declares the following public program objects:

- the functions HIRE and CREATE_DEPT
- the procedures REMOVE_EMP, REMOVE_DEPT, INCREASE_SAL, and INCREASE_COMM
- the exceptions NO_COMM and NO_SAL

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that call any of the package's public procedures or functions or raise any of the package's public exceptions.

Before you can call this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a CREATE PACKAGE BODY statement that creates the body of the EMP_MGMT package, see CREATE PACKAGE BODY on page 4-254.

Related Topics

ALTER PACKAGE on page 4-38
CREATE FUNCTION on page 4-232
CREATE PROCEDURE on page 4-259
CREATE PACKAGE BODY on page 4-254
DROP PACKAGE on page 4-394

CREATE PACKAGE BODY

Purpose

To create the body of a stored package. A *package* is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The *body* defines these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects. For a discussion of packages, including how to create packages, see CREATE PACKAGE on page 4-250. For some illustrations, see "Examples" on page 4-255.

Prerequisites

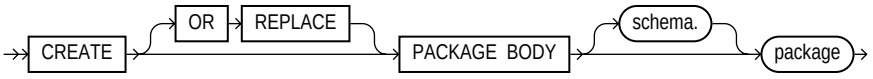
Before a package can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

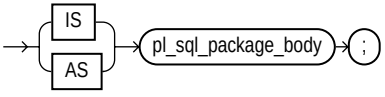
To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE BODY 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.

For more information, see *PL/SQL User's Guide and Reference*.

Syntax





Keywords and Parameters

OR REPLACE	re-creates the package body if it already exists. Use this option 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, Oracle recompiles it. For information on recompiling package bodies, see ALTER PACKAGE on page 4-38 Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.
schema	is the schema to contain the package. If you omit *schema*, Oracle creates the package in your current schema.
package	is the name of the package to be created.
pl/sql_package_body	is the package body. The package body can declare and define program objects. Package bodies are written in PL/SQL. For information on PL/SQL, including writing package bodies, see *PL/SQL User's Guide and Reference*.

Examples

Example. This SQL statement creates the body of the EMP_MGMT package:

```
CREATE PACKAGE BODY emp_mgmt AS
   tot_emps  NUMBER;
   tot_depts NUMBER;

FUNCTION hire
   (ename VARCHAR2,
    job VARCHAR2,
    mgr NUMBER,
    sal NUMBER,
    comm NUMBER,
    deptno NUMBER)

RETURN NUMBER IS
   new_empno NUMBER(4);
BEGIN
   SELECT empseq.NEXTVAL
      INTO new_empno
      FROM DUAL;
   INSERT INTO emp
      VALUES (new_empno, ename, job, mgr, sal, comm, deptno,
         tot_emps := tot_emps + 1;
   RETURN(new_empno);
END;
```

```
FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
   RETURN NUMBER IS
      new_deptno NUMBER(4);
   BEGIN
      SELECT deptseq.NEXTVAL
         INTO new_deptno
         FROM dual;
      INSERT INTO dept
         VALUES (new_deptno, dname, loc);
            tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;

PROCEDURE remove_emp(empno NUMBER) IS
   BEGIN
      DELETE FROM emp
      WHERE emp.empno = remove_emp.empno;
         tot_emps := tot_emps - 1;
   END;

PROCEDURE remove_dept(deptno NUMBER) IS
   BEGIN
      DELETE FROM dept
      WHERE dept.deptno = remove_dept.deptno;
         tot_depts := tot_depts - 1;
      SELECT COUNT(*)
         INTO tot_emps
         FROM emp;
         /* In case Oracle deleted employees from the EMP table
         to enforce referential integrity constraints, reset
         the value of the variable TOT_EMPS to the total
         number of employees in the EMP table. */
   END;

PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER) IS
   curr_sal NUMBER(7,2);
   BEGIN
      SELECT sal
      INTO curr_sal
      FROM emp
      WHERE emp.empno = increase_sal.empno;
      IF curr_sal IS NULL
         THEN RAISE no_sal;
      ELSE
```

```
         UPDATE emp
         SET sal = sal + sal_incr
         WHERE empno = empno;
      END IF;
   END;

PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER) IS
      curr_comm NUMBER(7,2);
   BEGIN
      SELECT comm
      INTO curr_comm
      FROM emp
      WHERE emp.empno = increase_comm.empno
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE emp
         SET comm = comm + comm_incr;
      END IF;
   END;

END emp_mgmt;
```

This package body corresponds to the package specification in the example of the CREATE PACKAGE command earlier in this chapter. The package body defines the public program objects declared in the package specification:

- the functions HIRE and CREATE_DEPT
- the procedures REMOVE_EMP, REMOVE_DEPT, INCREASE_SAL, and INCREASE_COMM

These objects are *declared* in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure INCREASE_ALL_COMMS separate from the EMP_MGMT package that calls the INCREASE_COMM procedure.

These objects are *defined* in the package body, so you can change their definitions without causing Oracle to invalidate dependent schema objects. For example, if you subsequently change the definition of HIRE, Oracle need not recompile INCREASE_ALL_COMMS before executing it.

The package body in this example also declares private program objects, the variables TOT_EMPS and TOT_DEPTS. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in

the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable TOT_DEPTS. However, the function CREATE_DEPT is part of the package, so CREATE_DEPT can change the value of TOT_DEPTS.

Related Topics

ALTER PACKAGE on page 4-38
CREATE FUNCTION on page 4-232
CREATE PROCEDURE on page 4-259
CREATE PACKAGE on page 4-250
DROP PACKAGE on page 4-394

CREATE PROCEDURE

Purpose

To create a standalone stored procedure or to register an external procedure. A *procedure* is a group of PL/SQL statements that you can call by name. An *external procedure* is a third-generation language (3GL) routine stored in a shared library which can be called from SQL or PL/SQL. To call an external procedure, you must provide information in your PL/SQL function about where to find the external procedure, how to call it, and what to pass to it. See also "Using Procedures" on page 4-262.

For more information about registering external procedures, see the *PL/SQL User's Guide and Reference*.

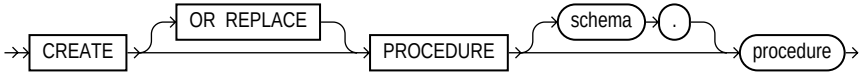
Prerequisites

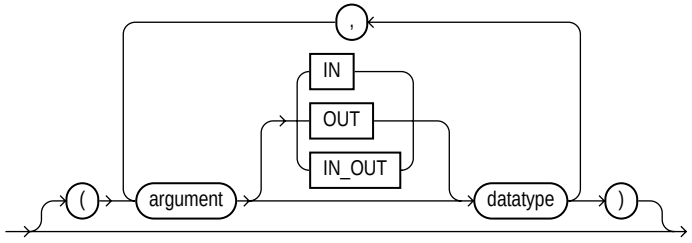
Before creating a procedure, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depends on your operating system.

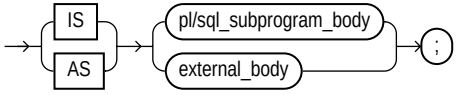
To create a procedure in your own schema, you must have CREATE PROCEDURE system privilege. To create a procedure in another schema, you must have CREATE ANY PROCEDURE system privilege. To replace a procedure in another schema, you must have ALTER ANY PROCEDURE system privilege.

To call an external procedure, you must have EXECUTE privileges on the callout library in which the procedure resides.

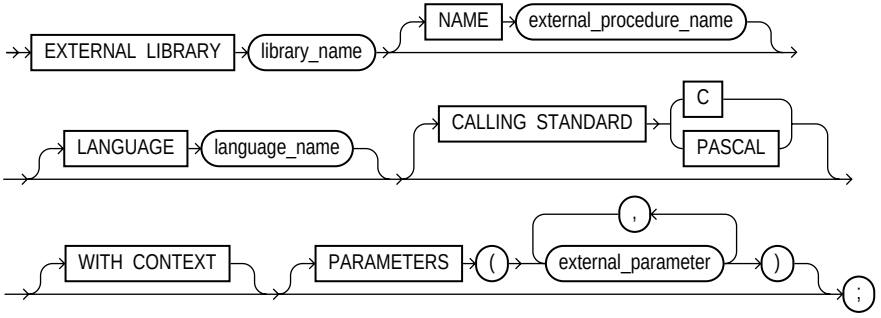
Syntax



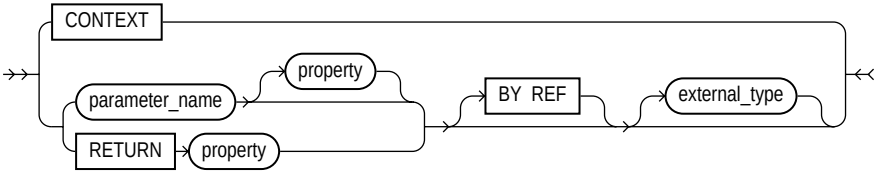




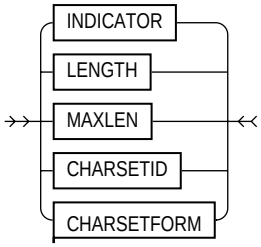
external_body ::=



external_parameter ::=



property ::=



Keywords and Parameters

OR REPLACE	re-creates the procedure if it already exists. Use this option 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, Oracle recompiles it. For information on recompiling procedures, see ALTER PROCEDURE on page 4-41. Users who had previously been granted privileges on a redefined procedure can still access the procedure without being regranted the privileges.
schema	is the schema to contain the procedure. If you omit *schema*, Oracle creates the procedure in your current schema.
procedure	is the name of the procedure to be created.
argument	is the name of an argument to the procedure. If the procedure does not accept arguments, you can omit the parentheses following the procedure name.
IN	specifies that you must specify a value for the argument when calling the procedure.
OUT	specifies that the procedure passes a value for this argument back to its calling environment after execution.
IN OUT	specifies that you must specify a value for the argument when calling the procedure and that the procedure passes a value back to its calling environment after execution. If you omit IN, OUT, and IN OUT, the argument defaults to IN.
datatype	is the datatype of the argument. As long as no length specifier is used, an argument can have any datatype supported by PL/SQL. For information on PL/SQL datatypes, see *PL/SQL User's Guide and Reference*. Datatypes are specified without a length, precision, or scale. For example, VARCHAR2(10) is not valid, but VARCHAR2 is valid. Oracle derives the length, precision, and scale of an argument from the environment from which the procedure is called.

IS *pl/sql_ subprogram_body*	is the definition of the procedure. Procedure definitions are written in PL/SQL. For information on PL/SQL, including how to write a PL/SQL subprogram body, see *PL/SQL User's Guide and Reference.*
AS *external_body*	identifies an external 3GL procedure stored in a sharable library. The AS *external_body* clause is the interface between PL/SQL and the external procedure.
LIBRARY	specifies the shared library in which the external procedure is stored. You must have EXECUTE privileges on the library. See CREATE LIBRARY on page 4-248 for the syntax.
library_name	is a PL/SQL identifier. Enclosing *library_name* in double quotes makes it case sensitive, but quotes are not required.
NAME *external_ procedure_name*	specifies the external procedure to be called. Enclosing *external_procedure_name* in double quotes makes it case sensitive, but quotes are not required. If you omit the name, it defaults to the PL/SQL subprogram (uppercase) name.
LANGUAGE	specifies the 3GL in which the external procedure was written. Currently, the only language name supported is C. If you omit the name, it defaults to C.
CALLING STANDARD	specifies the calling standard (C or PASCAL) under which the external procedure was compiled. If you omit the calling standard, it defaults to C.
WITH CONTEXT	specifies that a context pointer will be the first parameter passed to the external procedure. The context is opaque to the external procedure but is available to access functions called by the external procedure. For more information about the WITH CONTEXT clause, see the *PL/SQL User's Guide and Reference*.
PARAMETERS	specifies the positions and datatypes of parameters passed to the external procedure. It can also specify parameter properties such as current length and maximum length, and the preferred parameter passing method (by value or by reference). For more information about parameter passing, see the *PL/SQL User's Guide and Reference*.

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.

Using Procedures

A *procedure* is a group of PL/SQL statements that you can call by name. Stored procedures and stored functions are similar in many ways. This discussion applies to functions as well as to procedures. For information specific to functions, see CREATE FUNCTION on page 4-232.

With PL/SQL, you can group multiple SQL statements together with procedural PL/SQL statements similar to those in programming languages such as Ada and C. With the CREATE PROCEDURE command, you can create a procedure and store it

in the database. You can call a stored procedure from any environment from which you can issue a SQL statement.

Stored procedures offer advantages in the areas of development, integrity, security, performance, and memory allocation. For more information on stored procedures, including how to call stored procedures, see *Oracle8 Application Developer's Guide*.

The CREATE PROCEDURE command creates a procedure as a standalone schema object. You can also create a procedure as part of a package. For information on creating packages, see CREATE FUNCTION on page 4-232.

Example I. The following statement creates the procedure CREDIT in the schema SAM:

```
CREATE PROCEDURE sam.credit (acc_no IN NUMBER, amount IN NUMBER)
AS BEGIN
UPDATE accounts
SET balance = balance + amount
WHERE account_id = acc_no;
END;
```

The CREDIT procedure credits a specified bank account with a specified amount. When you call the procedure, you must specify the following arguments:

ACC_NO	is the number of the bank account to be credited. The argument's datatype is NUMBER.
AMOUNT	is the amount of the credit. The argument's datatype is NUMBER.

The procedure uses an UPDATE statement to increase the value in the BALANCE column of the ACCOUNTS table by the value of the argument AMOUNT for the account identified by the argument ACC_NO.

Example II. In the following example, external procedure C_FIND_ROOT expects a pointer as a parameter. Procedure FIND_ROOT passes the parameter by reference using the BY REF phrase:

```
CREATE PROCEDURE
( x IN REAL ) AS
EXTERNAL
EXTERNAL
LIBRARY c_utils
NAME "c_find_root"
PARAMETERS
```

```
( x BY REF );
```

See the *PL/SQL User's Guide and Reference* for information about external procedures.

Related Topics

ALTER PROCEDURE on page 4-41
CREATE FUNCTION on page 4-232
CREATE LIBRARY on page 4-248
CREATE PACKAGE on page 4-250
DROP PROCEDURE on page 4-396
PL/SQL User's Guide and Reference

CREATE PROFILE

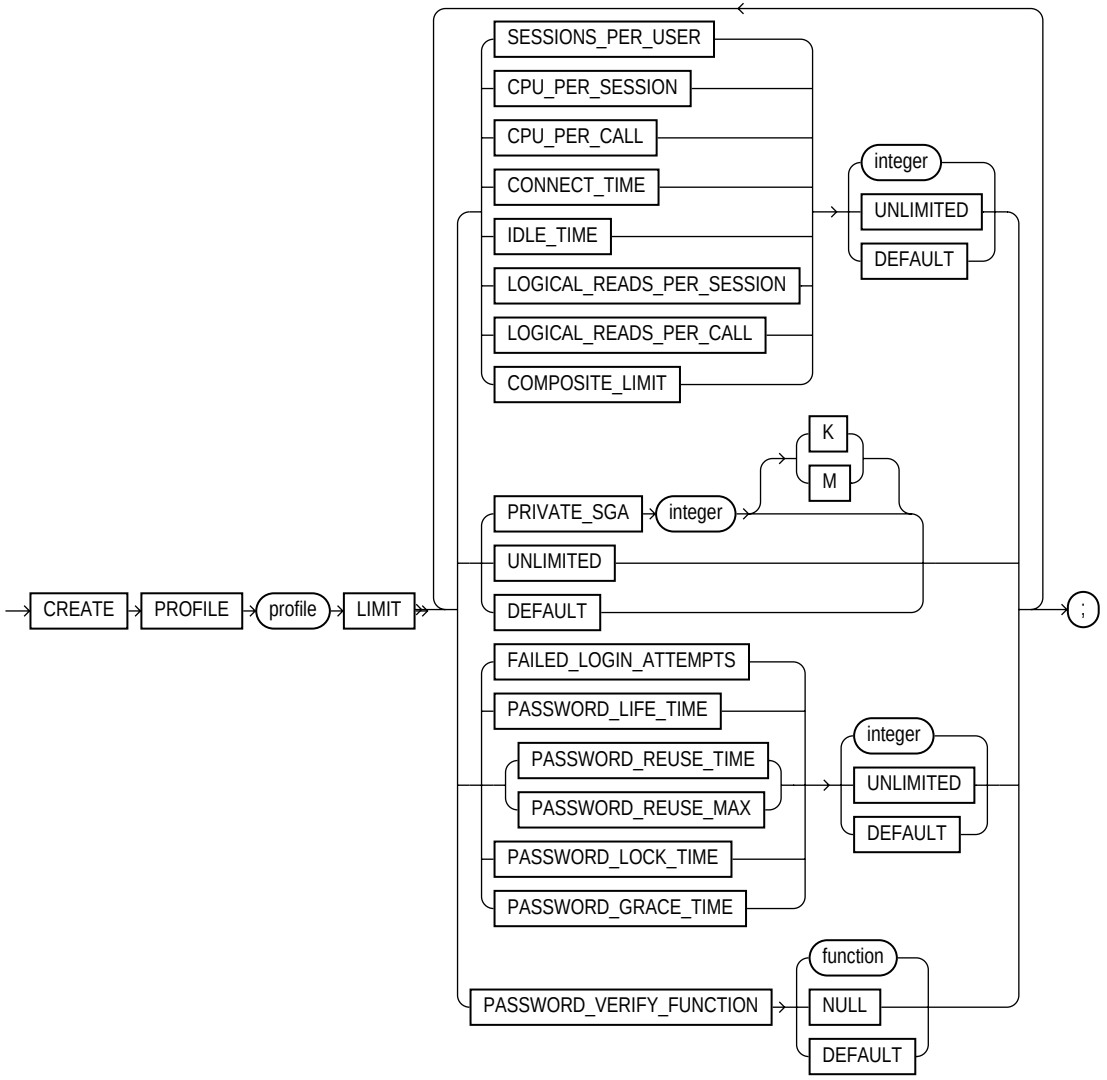
Purpose

To create a profile. A *profile* is a set of limits on database resources. If you assign the profile to a user, that user cannot exceed these limits.

Prerequisites

You must have CREATE PROFILE system privilege.

Syntax



Keywords and Parameters

profile is the name of the profile to be created. See also "Using Profiles" on page 4-268.

SESSIONS_PER_USER	limits a user to *integer* concurrent sessions.
CPU_PER_SESSION	limits the CPU time for a session, expressed in hundredth of seconds
CPU_PER_CALL	limits the CPU time for a call (a parse, execute, or fetch), expressed in hundredths of seconds.
CONNECT_TIME	limits the total elapsed time of a session, expressed in minutes.
IDLE_TIME	limits periods of continuous inactive time during a session, expressed in minutes. Long-running queries and other operations are not subject to this limit.
LOGICAL_READS_PER_SESSION	specifies the number of data blocks read in a session, including blocks read from memory and disk.
LOGICAL_READS_PER_CALL	specifies the number of data blocks read for a call to process a SQL statement (a parse, execute, or fetch).
PRIVATE_SGA	specifies the amount of private space a session can allocate in the shared pool of the system global area (SGA), expressed in bytes. You can use K or M to specify this limit in kilobytes or megabytes. This limit applies only if you are using multithreaded 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.
FAILED_LOGIN_ATTEMPTS	specifies the number of failed attempts to log in to the user account before the account is locked.
PASSWORD_LIFE_TIME	limits the number of days the same password can be used for authentication. The password expires if it is not changed within this period, and further connections are rejected. See also “Fractions in Dates” on page 4-269.
PASSWORD_REUSE_TIME	specifies the number of days before which a password cannot be reused. If you set PASSWORD_REUSE_TIME to an integer value, then you must set PASSWORD_REUSE_MAX to UNLIMITED.
PASSWORD_REUSE_MAX	specifies the number of password changes required before the current password can be reused. If you set PASSWORD_REUSE_MAX to an integer value, then you must set PASSWORD_REUSE_TIME to UNLIMITED.
PASSWORD_LOCK_TIME	specifies the number of days an account will be locked after the specified number of consecutive failed login attempts.
PASSWORD_GRACE_TIME	specifies the number of days after the grace period begins during which a warning is issued and login is allowed. If the password is not changed during the grace period, the password expires.

PASSWORD_VERIFY_ FUNCTION	allows a PL/SQL password complexity verification script to be passed as an argument to the CREATE PROFILE command. Oracle provides a default script, but you can create your own routine or use third-party software instead.	
	function	is the name of the password complexity verification routine.
	NULL	indicates that no password verification is performed.
	DEFAULT	omits a limit for this resource in this profile. A user assigned this profile is subject to the limit on the resource specified in the default profile.
COMPOSITE_LIMIT	specifies the total resources cost for a session, expressed in *service units*. Oracle calculates the total service units as a weighted sum of CPU_PER_ SESSION, CONNECT_TIME, LOGICAL_READS_PER_SESSION, and PRIVATE_SGA. For information on how to specify the weight for each session resource, see ALTER RESOURCE COST on page 4-48.	
UNLIMITED	indicates that a user assigned this profile can use an unlimited amount of this resource.	
DEFAULT	omits 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. See also "The DEFAULT Profile" on page 4-269.	

Using Profiles

A *profile* is a set of limits on database resources. You can use profiles to limit the database resources available to a user for a single call or a single session. Oracle enforces resource limits in the following ways:

- If a user exceeds the CONNECT_TIME or IDLE_TIME session resource limit, Oracle rolls back the current transaction and ends the session. When the user process next issues a call to Oracle, an error message is returned.
- If a user attempts to perform an operation that exceeds the limit for other session resources, Oracle aborts the operation, rolls back the current statement, and immediately returns an error. The user can then commit or roll back the current transaction. The user must then end the session.
- If a user attempts to perform an operation that exceeds the limit for a single call, Oracle aborts the operation, rolls back the current statement, and returns an error message, leaving the current transaction intact.

Fractions in Dates

You can use fractions of days for all parameters, with days as units. Fractions are expressed as *x/y*. For example, 1 hour is 1/24 and 1 minute is 1/1440.

For a detailed description and explanation of how to use password management and protection, see the *Oracle8 Administrator's Guide*.

To specify resource limits for a user, you must perform both of the following operations:

Enable resource limits

You can enable resource limits in one of two ways:

- with the initialization parameter RESOURCE_LIMIT. Note that this parameter does not apply to password resources. Password resources are always enabled.
- dynamically, with the ALTER SYSTEM command. See ALTER SYSTEM on page 4-88.

Specify resource limits

To specify a resource limit for a user, you must perform following steps:

1. Create a profile that defines the limits using the CREATE PROFILE command.
2. Assign the profile to the user using the CREATE USER or ALTER USER command.

Note that you can specify resource limits for users regardless of whether resource limits are enabled. However, Oracle does not enforce these limits until you enable them.

The DEFAULT Profile

Oracle automatically creates a default profile named DEFAULT. This profile initially defines unlimited resources. You can change the limits defined in this profile with the ALTER PROFILE command.

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, the user is subject to the limits on those resources defined by the DEFAULT profile.

Example I. The following statement creates the profile SYSTEM_MANAGER:

```
CREATE PROFILE system_manager
```

```
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 then assign the SYSTEM_MANAGER profile to a user, 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 command.
- Since the SYSTEM_MANAGER profile omits a limit for IDLE_TIME, the user is subject to the limit on this resource specified in the DEFAULT profile.

Example II. The following example creates the profile PROF:

```
CREATE PROFILE prof
  LIMIT PASSWORD_REUSE_MAX DEFAULT
        PASSWORD_REUSE_TIME UNLIMITED;
```

Example III. The following example creates profile MYPROFILE with password profile limits values set:

```
CREATE PROFILE myprofile LIMIT
FAILED_LOGIN_ATTEMPTS 5
PASSWORD_LIFE_TIME 60
PASSWORD_REUSE_TIME 60
PASSWORD_REUSE_MAX UNLIMITED
```

```
PASSWORD_VERIFY_FUNCTION verify_function
PASSWORD_LOCK_TIME 1/24
PASSWORD_GRACE_TIME 10;
```

Related Topics

ALTER PROFILE on page 4-43
ALTER RESOURCE COST on page 4-48
ALTER SYSTEM on page 4-88
ALTER USER on page 4-150
DROP PROFILE on page 4-398

CREATE ROLE

Purpose

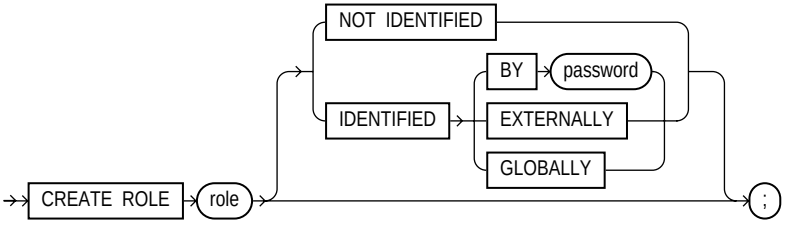
To create a role. A *role* is a set of privileges that can be granted to users or to other roles. See also "Using Roles" on page 4-273.

For a detailed description and explanation of using global roles, see *Oracle8 Distributed Database Systems.*

Prerequisites

You must have CREATE ROLE system privilege.

Syntax



Keywords and Parameters

role is the name of the role to be created. Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters. See also "Roles Predefined by Oracle" on page 4-273.

NOT IDENTIFIED indicates that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED indicates that a user must be authorized by the specified method before the role is enabled with the SET ROLE command:

- BY *password* — The user must specify the password to Oracle when enabling the role. The password can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.
- EXTERNALLY — indicates that a 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. For more information about third-party service, see *Oracle Security Server Guide*.
GLOBALLY	indicates that a user must be authorized to use the role by the Oracle Security Service before the role is enabled with the SET ROLE command, or at login.

If you omit both the NOT IDENTIFIED option and the IDENTIFIED clause, the role defaults to NOT IDENTIFIED.

Using Roles

A *role* 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. For information on enabling roles, see ALTER USER on page 4-150.

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 command.

When you create a role, Oracle grants you the role with ADMIN OPTION. The ADMIN OPTION allows you to

- grant the role to another user or role, unless the role is a GLOBAL role
- revoke the role from another user or role
- alter the role to change the authorization needed to access it
- drop the role

Note: When you create a role IDENTIFIED GLOBALLY, Oracle does not grant you the role as it does with nonglobal roles.

Roles Predefined by Oracle

Some roles are defined by SQL scripts provided on your distribution media. The following roles are predefined:

- CONNECT
- RESOURCE
- DBA

- EXP_FULL_DATABASE
- IMP_FULL_DATABASE
- DELETE_CATALOG_ROLE
- EXECUTE_CATALOG_ROLE
- SELECT_CATALOG_ROLE

The CONNECT, RESOURCE, and DBA roles are provided for compatibility with previous versions of Oracle. You should not rely on these roles; rather, Oracle recommends that you to design your own roles for database security. These roles may not be created automatically by future versions of Oracle.

The SELECT_CATALOG_ROLE, EXECUTE_CATALOG_ROLE, and DELETE_CATALOG_ROLE roles are provided for accessing exported data dictionary views and packages. For more information on these roles, see *Oracle8 Application Developer's Guide*.

The EXP_FULL_DATABASE and IMP_FULL_DATABASE roles are provided for convenience in using the Import and Export utilities.

Oracle also creates other roles that authorize you to administer the database. On many operating systems, these roles are called OSOPER and OSDBA. Their names may be different on your operating system.

Example I. The following example creates global role VENDOR:

```
CREATE ROLE vendor IDENTIFIED GLOBALLY;
```

Example II. The following statement creates the role TELLER:

```
CREATE ROLE teller
IDENTIFIED BY cashflow;
```

Users who are subsequently granted the TELLER role must specify the password CASHFLOW to enable the role with the SET ROLE command.

Related Topics

ALTER ROLE on page 4-51
DROP ROLE on page 4-399
GRANT (System Privileges and Roles) on page 4-434
REVOKE (System Privileges and Roles) on page 4-475
SET ROLE on page 4-516

CREATE ROLLBACK SEGMENT

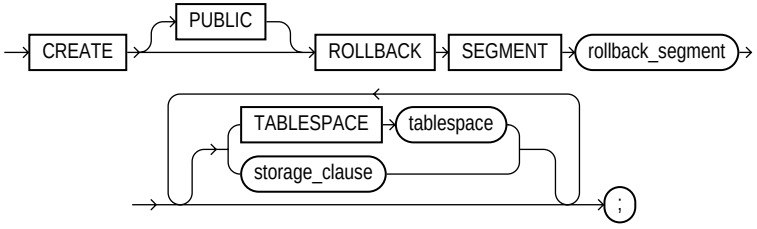
Purpose

To create a rollback segment. A *rollback segment* is an object that Oracle uses to store data necessary to reverse, or undo, changes made by transactions.

Prerequisites

You must have CREATE ROLLBACK SEGMENT system privilege. Also, you must have either space quota on the tablespace to contain the rollback segment or UNLIMITED TABLESPACE system privilege.

Syntax



storage_clause: See the STORAGE clause on page 4-523.

Keyword and Parameters

PUBLIC	specifies that the rollback segment is public and is available to any instance. If you omit this option, the rollback segment is private and is available only to the instance naming it in its initialization parameter ROLLBACK_SEGMENTS.
rollback_segment	is the name of the rollback segment to be created.
TABLESPACE	identifies the tablespace in which the rollback segment is created. If you omit this option, Oracle creates the rollback segment in the SYSTEM tablespace. See also "Rollback Segments and Tablespaces" on page 4-276.

storage_clause	specifies the characteristics for the rollback segment. See the STORAGE clause on page 4-523. **Note**: The PCTINCREASE option of the *storage_clause* is not permitted with CREATE ROLLBACK SEGMENT.
OPTIMAL	This part of the STORAGE clause specifies an optimal size in bytes for a rollback segment. You can use K or M to specify this size in kilobytes or megabytes. 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. NULL specifies no optimal size for the rollback segment, meaning that Oracle never deallocates the rollback segment's extents. This is the default behavior. The value of this parameter cannot be less than the space initially allocated for the rollback segment specified by the MINEXTENTS, INITIAL, and NEXT parameters. The maximum value depends on your operating system. Oracle rounds values to the next multiple of the data block size.

Rollback Segments and Tablespaces

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 instance, you must bring it online using one of the following:

- ALTER ROLLBACK SEGMENT command
- ROLLBACK_SEGMENTS initialization parameter

For more information on creating rollback segments and making them available, see *Oracle8 Administrator's Guide.*

A tablespace can have multiple rollback segments. Generally, multiple rollback segments improve performance.

Example . The following statement creates a rollback segment with default storage values in the system tablespace:

```
CREATE ROLLBACK SEGMENT rbs_2
TABLESPACE system;
```

The above statement is equivalent to the following:

```
CREATE ROLLBACK SEGMENT rbs_2
TABLESPACE system
```

```
STORAGE
( INITIAL 10 K
  NEXT 10 K
  MAXEXTENTS UNLMIITED);
```

Related Topics

CREATE TABLESPACE on page 4-328
CREATE TABLESPACE on page 4-328
ALTER ROLLBACK SEGMENT on page 4-53
DROP ROLLBACK SEGMENT on page 4-400
STORAGE clause on page 4-523

CREATE SCHEMA

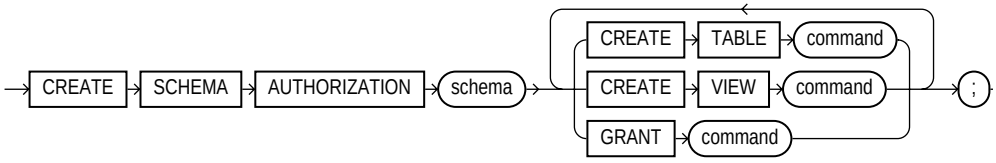
Purpose

To create multiple tables and views and perform multiple grants in a single transaction. See also “Creating Schemas” on page 4-278.

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



Keyword and Parameters

schema	is the name of the schema. The schema name must be the same as your Oracle username.
CREATE TABLE *command*	is a CREATE TABLE statement to be issued as part of this CREATE SCHEMA statement. See the CREATE TABLE on page 4-306
CREATE VIEW *command*	is a CREATE VIEW statement to be issued as part of this CREATE SCHEMA statement. See the CREATE VIEW on page 4-363.
GRANT *command*	is a GRANT statement (Object Privileges) to be issued as part of this CREATE SCHEMA statement. See GRANT (Object Privileges) on page 4-444.
	The CREATE SCHEMA statement supports the syntax of these commands only as defined by standard SQL, rather than the complete syntax supported by Oracle.

Creating Schemas

With the CREATE SCHEMA command, you can issue multiple data definition language (DDL) statements in a single transaction. To execute a CREATE SCHEMA statement, Oracle executes each included statement. If all statements execute

successfully, Oracle commits the transaction. If any statement results in an error, Oracle rolls back all the statements.

Terminate a CREATE SCHEMA statement just as you would any other SQL statement using the terminator character specific to your tool. For example, if you issue a CREATE SCHEMA statement in SQL*Plus or Server Manager, terminate the statement with a semicolon (;). Do not separate the individual statements within a CREATE SCHEMA statement with the terminator character.

The order in which you list the CREATE TABLE, CREATE VIEW, and GRANT statements is unimportant:

- A CREATE VIEW statement can create a view that is based on a table that is created by a later CREATE TABLE statement.
- A CREATE TABLE statement can create a table with a foreign key that depends on the primary key of a table that is created by a later CREATE TABLE statement.
- A GRANT statement can grant privileges on a table or view that is created by a later CREATE TABLE or CREATE VIEW statement.

The statements within a CREATE SCHEMA statement can also reference existing objects:

- A CREATE VIEW statement can create a view on a table that existed before the CREATE SCHEMA statement.
- A GRANT statement can grant privileges on a previously existing object.

Note: The syntax of the PARALLEL clause is allowed for a CREATE TABLE, INDEX, or CLUSTER, when used in CREATE SCHEMA, but parallelism is **not** used when creating the objects.

Example. The following statement creates a schema named BLAIR for the user BLAIR, creates the table SOX, creates the view RED_SOX, and grants SELECT privilege on the RED_SOX view to the user WAITES.

```
CREATE SCHEMA AUTHORIZATION blair
CREATE TABLE sox
(color VARCHAR2(10)  PRIMARY KEY, quantity NUMBER)
CREATE VIEW red_sox
AS SELECT color, quantity FROM sox WHERE color = 'RED'
GRANT select ON red_sox TO waites;
```

Related Topics

CREATE TABLE on page 4-306
CREATE VIEW on page 4-363
GRANT (Object Privileges) on page 4-444

CREATE SEQUENCE

Purpose

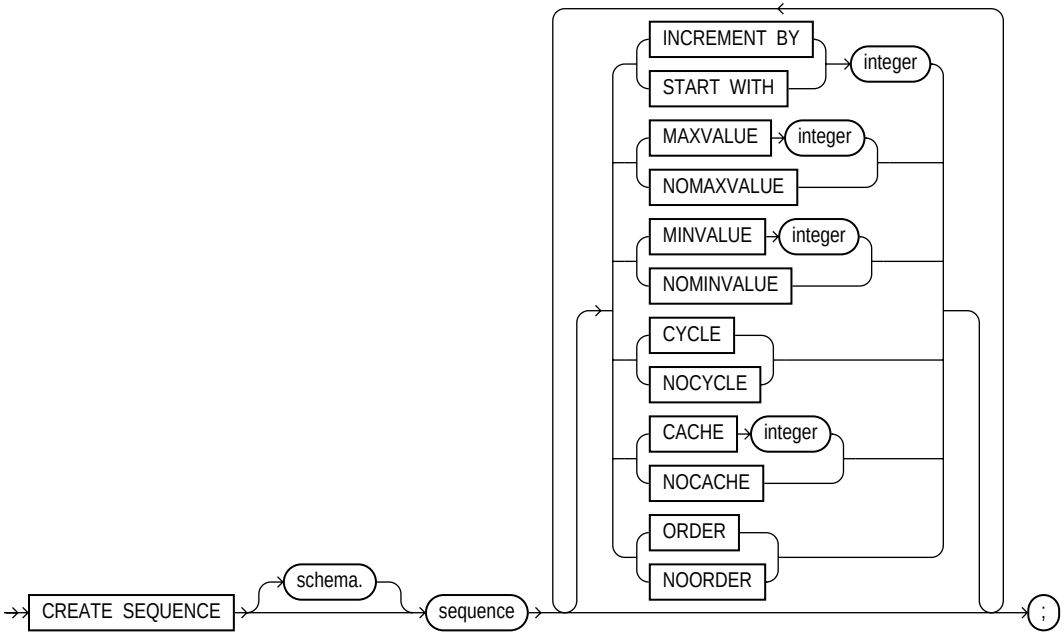
To create a sequence. A *sequence* is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values. See also "Using Sequences" on page 4-283, "Sequence Defaults" on page 4-284, and "Accessing Sequence Values" on page 4-285.

Prerequisites

To create a sequence in your own schema, you must have CREATE SEQUENCE privilege.

To create a sequence in another user's schema, you must have CREATE ANY SEQUENCE privilege.

Syntax



Keywords and Parameters

schema	is the schema to contain the sequence. If you omit *schema*, Oracle creates the sequence in your own schema.
sequence	is the name of the sequence to be created.
INCREMENT BY	specifies 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. 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 increment is positive, then the sequence ascends. If you omit this clause, the interval defaults to 1. See also "Incrementing Sequence Values" on page 4-284.
START WITH	specifies the first sequence number to be generated. Use this option 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 sequence's minimum value. For descending sequences, the default value is the sequence's maximum value. This integer value can have 28 or fewer digits.
MAXVALUE	specifies the maximum value the sequence can generate. This integer value can have 28 or fewer digits. MAXVALUE must be equal to or less than START WITH and must be greater than MINVALUE.
NOMAXVALUE	specifies a maximum value of 10^27 for an ascending sequence or -1 for a descending sequence. This is the default.
MINVALUE	specifies the sequence's minimum value. This integer value can have 28 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.
NOMINVALUE	specifies a minimum value of 1 for an ascending sequence or -(10^26) for a descending sequence. This is the default.
CYCLE	specifies 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.
NOCYCLE	specifies that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.
CACHE	specifies how many values of the sequence Oracle 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; thus, the maximum value allowed for CACHE must be less than the value determined by the following formula:

(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)

See also “Caching Sequence Numbers” on page 4-284.

NOCACHE specifies that values of the sequence are not preallocated.

If you omit both the CACHE parameter and the NOCACHE option, Oracle caches 20 sequence numbers by default.

ORDER guarantees that sequence numbers are generated in order of request. You may want to use this option if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys.

NOORDER does not guarantee sequence numbers are generated in order of request.

If you omit both the ORDER and NOORDER options, Oracle chooses NOORDER by default. Note that the ORDER option is necessary only to guarantee ordered generation if you are using Oracle with the Parallel Server option in parallel mode. If you are using exclusive mode, sequence numbers are always generated in order.

Using Sequences

You can use sequence numbers to automatically generate unique primary key values for your data, and you can also coordinate the keys across multiple rows or tables.

Values for a given sequence are automatically generated by special Oracle routines and, consequently, sequences avoid the performance bottleneck that results from implementation of sequences at the application level. For example, one common application-level implementation is to force each transaction to lock a sequence number table, increment the sequence, and then release the table. Under this implementation, only one sequence number can be generated at a time. In contrast, Oracle sequences permit the simultaneous generation of multiple sequence numbers while guaranteeing that every sequence number is unique.

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, 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. Once 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.

Sequence Defaults

The sequence defaults are designed so that if you specify none of the clauses, 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.

Incrementing Sequence Values

You can create a sequence so that its values increment in one of following ways:

- The sequence values increment without bound.
- The sequence values increment to a predefined limit and then stop.
- The sequence values increment to a predefined limit and then restart.

To create a sequence that increments without bound, omit the MAXVALUE parameter or specify the NOMAXVALUE option for ascending sequences or omit the MINVALUE parameter or specify the NOMINVALUE for descending sequences.

To create a sequence that stops at a predefined limit, specify a value for the MAXVALUE parameter for an ascending sequence or a value for the MINVALUE parameter for a descending sequence. Also specify the NOCYCLE option. 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 the CYCLE option. If you do not specify MINVALUE, then it defaults to NOMINVALUE; that is, the value 1.

The value of the START WITH parameter establishes the initial value generated after the sequence is created. Note that this value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.

Caching Sequence Numbers

The number of values cached in memory for a sequence is specified by the value of the sequence's CACHE parameter. Cached sequences allow faster generation of sequence numbers. A cache for a given sequence is populated at the first request for a number from that sequence. The cache is repopulated every CACHE requests.

If there is a system failure, all cached sequence values that have not been used in committed data manipulation language (DML) statements are lost. The potential number of lost values is equal to the value of the CACHE parameter.

A CACHE of 20 future sequence numbers is the default.

Accessing Sequence Values

Once a sequence is created, you can access its values in SQL statements with the following pseudocolumns:

CURRVAL returns the current value of the sequence.

NEXTVAL increments the sequence and returns the new value.

For more information on using the above pseudocolumns, see the section "Pseudocolumns" on page 2-32.

Example. The following statement creates the sequence ESEQ:

```
CREATE SEQUENCE eseq
  INCREMENT BY 10
```

The first reference to ESEQ.NEXTVAL returns 1. The second returns 11. Each subsequent reference will return a value 10 greater than the one previous.

Related Topics

ALTER SEQUENCE on page 4-56
DROP SEQUENCE on page 4-401

CREATE SNAPSHOT

Purpose

To create a snapshot. A *snapshot* is a table that contains the results of a query of one or more tables, often located on a remote database.

Prerequisites

The following prerequisites apply to creating snapshots:

- To create a snapshot in your own schema, you must have the CREATE SNAPSHOT, CREATE TABLE, and CREATE VIEW system privileges.
- To create a snapshot in another user's schema, you must have the CREATE ANY SNAPSHOT system privilege.
- The schema that contains the snapshot must have sufficient quota in the target tablespace to store the snapshot's base table and index or have the UNLIMITED TABLESPACE system privilege.
- To create and refresh a snapshot, both the creator and snapshot owner must be able to issue the defining query of the snapshot. This capability depends directly on the database link that the snapshot's defining query uses.

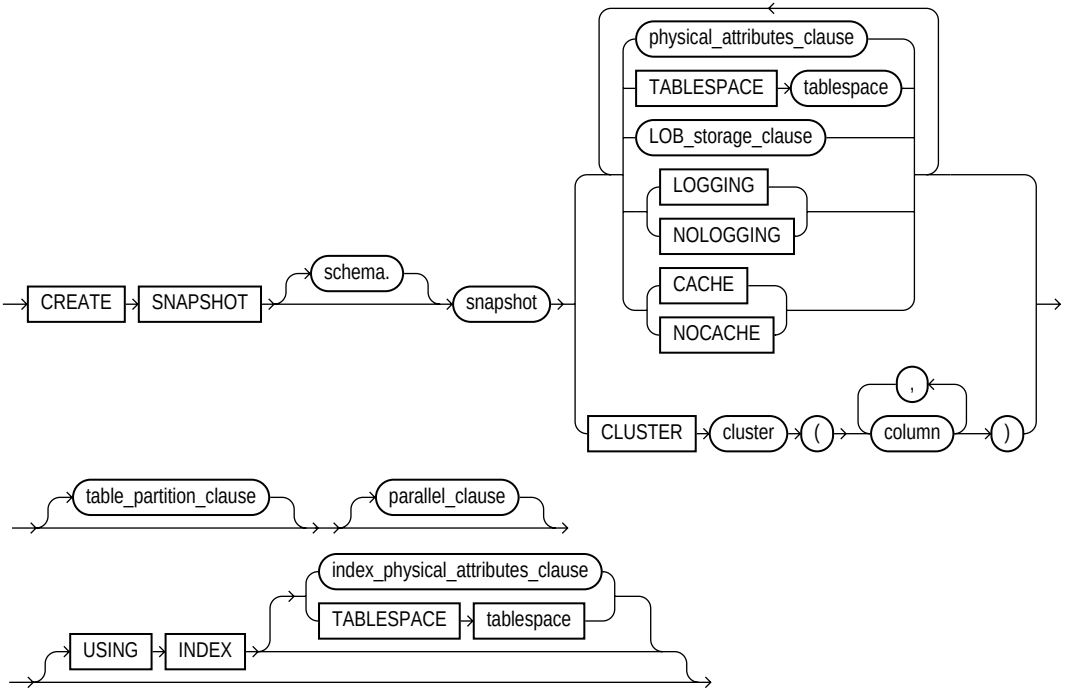
By default, Oracle creates all new snapshots as primary key snapshots. To create a snapshot:

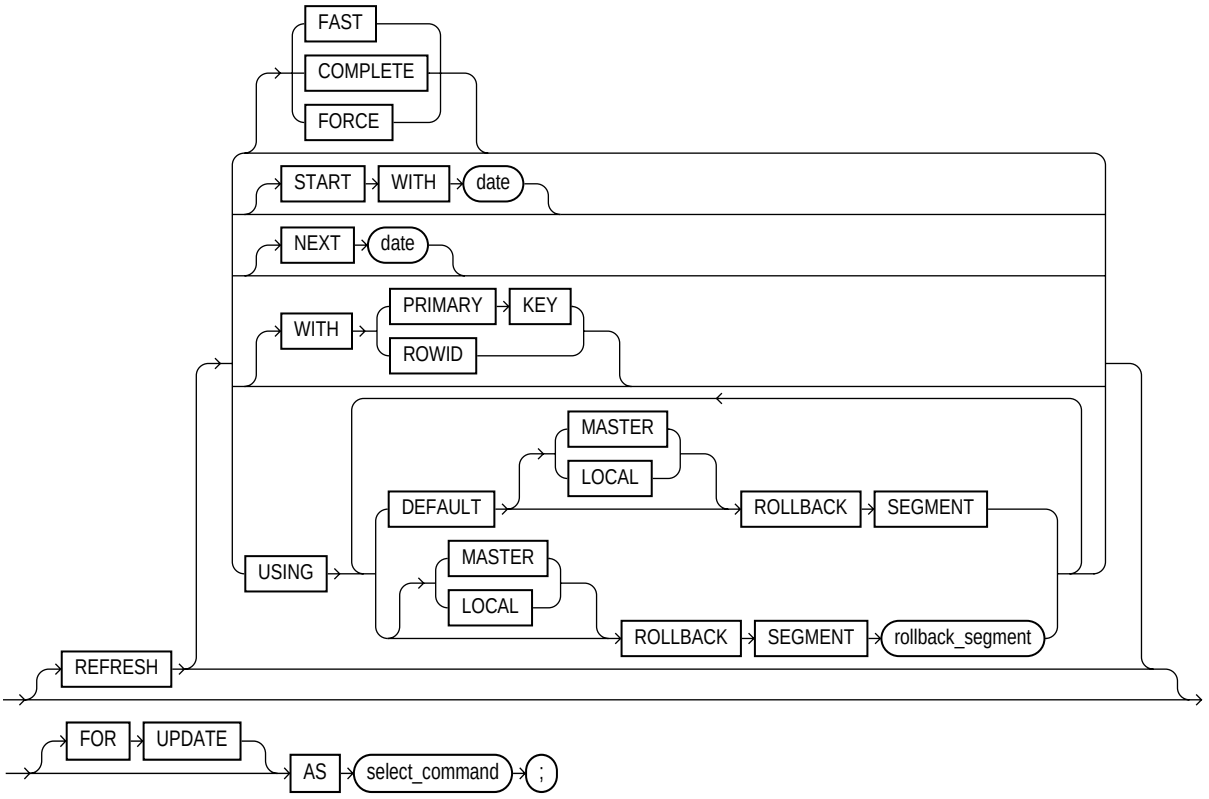
- The master table for a snapshot must contain an enabled PRIMARY KEY constraint before you create the new snapshot.
- The defining query of the snapshot must reference all columns in the master table's primary key.

When you create a snapshot, Oracle creates one table, one view, and at least one index in the schema of the snapshot. Oracle uses these objects to maintain the snapshot's data. You must have the privileges necessary to create these objects. For information on these privileges, see CREATE TABLE on page 4-306, CREATE VIEW on page 4-363, and CREATE INDEX on page 4-237.

For complete information about the prerequisites that apply to creating snapshots, see *Oracle8 Replication*.

Syntax


CREATE
SNAPSHOT
schema.
snapshot
physical_attributes_clause
TABLESPACE
tablespace
LOB_storage_clause
LOGGING
NOLOGGING
CACHE
NOCACHE
CLUSTER
cluster
(
,
column
)
table_partition_clause
parallel_clause
USING
INDEX
index_physical_attributes_clause
TABLESPACE
tablespace



physical_attributes_clause: See ALTER TABLE on page 4-106.

parallel_clause: See the PARALLEL clause on page 4-465.

index_physical_attributes_clause: See ALTER INDEX on page 4-28.

select_command: See SELECT on page 4-489.

LOB_storage_clause: See CREATE TABLE on page 4-306.

table_partition_clause: See CREATE TABLE on page 4-306.

Keywords and Parameters

schema is the schema to contain the snapshot. If you omit *schema*, Oracle creates the snapshot in your schema.

snapshot	is the name of the snapshot to be created. Oracle generates names for the table, view, and indexes used to maintain the snapshot by adding a prefix or suffix to the snapshot name. Oracle recommends that you limit your snapshot names to 19 bytes, so that the Oracle-generated names will be 30 bytes or less and will contain the entire snapshot name. See also "About Snapshots" on page 4-291.
physical_ attributes_clause	establishes values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters (or, when used in the USING INDEX clause, for the INITRANS and MAXTRANS parameters only) and the storage parameters for the internal table Oracle uses to maintain the snapshot's data. For information on the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters, see CREATE TABLE on page 4-306. For information, about the STORAGE clause, see the STORAGE clause on page 4-523.
TABLESPACE	specifies the tablespace in which the snapshot is to be created. If you omit this option, Oracle creates the snapshot in the default tablespace of the owner of the snapshot's schema.
LOB_storage_ clause	specifies the LOB storage characteristics. For detailed information about specifying the parameters of the LOB storage clause, see CREATE TABLE on page 4-306.
STORAGE	establishes storage characteristics for the table Oracle uses to maintain the snapshot's data.
CLUSTER	creates the snapshot as part of the specified cluster. Since a clustered snapshot uses the cluster's space allocation, do not use the *physical_attributes_clause* or the TABLESPACE option with the CLUSTER option.
table_partition_ clause	specifies that the table is partitioned on specified ranges of values. For detailed information about specifying the parameters of the table partition clause, see CREATE TABLE on page 4-306. See also "Partitioned Snapshots" on page 4-296.
USING INDEX	specifies parameters for the index Oracle creates to maintain the snapshot. See *physical_ attributes_clause*, above.
USING ROLLBACK SEGMENT	specifies the local snapshot and / or remote master rollback segments to be used during snapshot refresh.

rollback_segment	is the name of the rollback segment to be used.
DEFAULT	specifies that Oracle will choose which rollback segment to use.
MASTER	specifies the rollback segment to be used at the remote master for the individual snapshot.
LOCAL	specifies the rollback segment to be used for the local refresh group that contains the snapshot.

If you do not specify MASTER or LOCAL, Oracle uses LOCAL by default. If you do not specify *rollback_segment*, Oracle automatically chooses the rollback segment to be used. If you specify DEFAULT, you cannot specify *rollback_segment*. See also "Specifying Rollback Segments" on page 4-294.

REFRESH specifies how and when Oracle automatically refreshes the snapshot:

- FAST specifies a fast refresh, or a refresh using only the updated data stored in the snapshot log associated with the master table.
- COMPLETE specifies a complete refresh, or a refresh that reexecutes the snapshot's query.
- FORCE specifies a fast refresh if one is possible or complete refresh if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time.

If you omit the FAST, COMPLETE, and FORCE options, Oracle uses FORCE by default. See also "Refreshing Snapshots" on page 4-292.

- START WITH specifies a date expression for the first automatic refresh time.
- NEXT specifies 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, Oracle determines the first automatic refresh time by evaluating the NEXT expression when you create the snapshot. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the snapshot only once. If you omit both the START WITH and NEXT values, or if you omit the REFRESH clause entirely, Oracle does not automatically refresh the snapshot.

- WITH PRIMARY KEY specifies that primary-key snapshots are to be created. Primary-key snapshots allow snapshot master tables to be reorganized without affecting the snapshot's ability to continue to fast refresh.

  Primary-key snapshots can also be defined as simple snapshots with subqueries.
- WITH ROWID specifies that ROWID snapshots are to be created.

  ROWID snapshots provide compatibility with Oracle7 Release 7.3 master tables.

If you omit both WITH PRIMARY KEY and WITH ROWID, Oracle creates primary-key snapshots by default. See also "Specifying Primary-Key or ROWID Snapshots" on page 4-295.

FOR UPDATE allows a simple snapshot to be updated. When used in conjunction with the Replication Option, these updates will be propagated to the master. For more information, see *Oracle8 Replication*.

AS *select_ command*	specifies the snapshot query. When you create the snapshot, Oracle executes this query and places the results in the snapshot. This query is any valid SQL query, but not all queries are fast refreshable. See also "Types of Snapshots" on page 4-291.

About Snapshots

A *snapshot* is a table that contains the results of a query of one or more tables, often located on a remote database. The tables in the query are called *master tables*. The databases containing the master tables are called the *master databases*. Note that a snapshot query cannot select from tables or views owned by the user SYS.

Snapshots are useful in distributed databases. Snapshots allow you to maintain read-only copies of remote data on your local node. You can select data from a snapshot as you would from a table or view.

Oracle recommends that you qualify each table and view in the FROM clause of the snapshot query with the schema containing it. For some additional caveats, see "The View Query" on page 4-366 (in the context of the CREATE VIEW command). The same recommendations apply to creating snapshots.

Snapshots cannot contain long columns.

For more information on snapshots, see *Oracle8 Replication*.

Types of Snapshots

You can create two types of snapshots: simple and complex.

A *simple* snapshot is based on a single remote table, or is defined on multiple tables using restricted types of subqueries. For more information about simple snapshots with subqueries, see *Oracle8 Replication*.

Simple snapshots do not contain any of the following items in the snapshot query (*select_command_clause*):

- GROUP BY clause
- CONNECT BY clause
- distinct or aggregate functions
- joins (other than allowable types of subqueries)
- set operations

A *complex* snapshot is one in which the snapshot query contains one or more of the constructs not allowed in the query of a simple snapshot. A complex snapshot can be based on multiple master tables on multiple master databases.

Refreshing Snapshots

A snapshot's master tables can be modified, so the data in a snapshot must be updated occasionally to ensure that the snapshot accurately reflects the data currently in its master tables. The process of updating a snapshot for this purpose is called *refreshing* the snapshot. With the REFRESH clause of the CREATE SNAPSHOT command, you can schedule the times and specify the mode for Oracle to refresh the snapshot automatically.

After you create a snapshot, you can subsequently change its automatic refresh mode and time with the REFRESH clause of the ALTER SNAPSHOT command. You can also refresh a snapshot immediately with the DBMS_SNAPSHOT.REFRESH() procedure.

Specifying Refresh Modes

Use the FAST or COMPLETE options of the REFRESH clause to specify the refresh mode.

Fast . To perform a *fast refresh*, Oracle updates the snapshot with the changes to the master table recorded in its snapshot log. For more information on snapshot logs, see CREATE SNAPSHOT LOG on page 4-297.

Oracle can only perform a fast refresh if all of the following conditions are true:

- The snapshot is a simple snapshot.
- The snapshot's master table has a snapshot log.
- The snapshot log was created before the snapshot was last refreshed or created.

If you specify a fast refresh and all of above conditions are true, then Oracle performs a fast refresh. If any of the conditions are not true, Oracle returns an error at refresh time and does not refresh the snapshot.

Complete. To perform a *complete refresh*, Oracle reexecutes the snapshot query and places the results in the snapshot. If you specify a complete refresh, Oracle performs a complete refresh regardless of whether a fast refresh is possible.

A fast refresh is often faster than a complete refresh because it sends less data from the master database across the network to the snapshot's database. A fast refresh sends only changes to master table data since the last refresh, while a complete refresh sends the complete result of the snapshot query.

You can also use the FORCE option of the REFRESH clause to allow Oracle to decide how to refresh the snapshot at the scheduled refresh time. If a fast refresh is

possible based on the fast refresh conditions, then Oracle performs a fast refresh. If a fast refresh is not possible, then Oracle performs a complete refresh.

Example. The following statement creates the simple snapshot EMP_SF that contains the data from SCOTT's employee table in New York:

```
CREATE SNAPSHOT emp_sf
PCTFREE 5 PCTUSED 60
TABLESPACE users
STORAGE INITIAL 50K NEXT 50K
REFRESH FAST NEXT sysdate + 7
AS
SELECT * FROM scott.emp@ny;
```

The statement does not include a START WITH parameter, so Oracle determines the first automatic refresh time by evaluating the NEXT value using the current SYSDATE. Provided a snapshot log currently exists for the employee table in New York, Oracle performs a fast refresh of the snapshot every 7 days, beginning 7 days after the snapshot is created.

The above statement also establishes for the table storage characteristics that Oracle uses to maintain the snapshot.

Specifying Automatic Refresh Times

To cause Oracle to refresh a snapshot automatically, you must perform the following tasks:

1. Specify the START WITH and NEXT parameters in the REFRESH clause of the CREATE SNAPSHOT statement. These parameters establish the time of the first automatic refresh and the interval between automatic refreshes.
2. Enable one or more job queue processes using the initialization parameters SNAPSHOT_REFRESH_PROCESSES, SNAPSHOT_REFRESH_INTERVAL, SNAPSHOT_REFRESH_KEEP_CONNECTIONS. The job queue processes then examine the automatic refresh time of each snapshot in the database. For each snapshot that is scheduled to be refreshed at or before the current time, one job queue process performs the following operations:
   - reevaluates the snapshot's NEXT value to determine the next automatic refresh time
   - refreshes the snapshot
   - stores the next automatic refresh time in the data dictionary

For more information on these initialization parameters, see *Oracle8 Reference*.

Example. The following statement creates the complex snapshot ALL_EMPS that queries the employee tables in Dallas and Baltimore:

```
CREATE SNAPSHOT all_emps
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE INITIAL 50K NEXT 50K
   USING INDEX STORAGE (INITIAL 25K NEXT 25K)
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE, 'MONDAY') + 15/24
AS
   SELECT * FROM fran.emp@dallas
      UNION
   SELECT * FROM marco.emp@balt;
```

Oracle automatically refreshes this snapshot tomorrow at 11:00 am and subsequently every Monday at 3:00 pm. This command does not specify either fast or complete refreshes, so Oracle must decide how to refresh the snapshot. Since ALL_EMPS is a complex snapshot, Oracle must perform a complete refresh.

The above statement also establishes storage characteristics for both the table and the index that Oracle uses to maintain the snapshot:

- The first STORAGE clause establishes the sizes of the first and second extents of the table as 50 kilobytes each.
- The second STORAGE clause (appearing with the USING INDEX option) establishes the sizes of the first and second extents of the index as 25 kilobytes each.

Specifying Rollback Segments

You can specify the rollback segments to be used during a refresh for both the master site and the local site.

The local snapshot rollback segment is stored at the refresh group level. If the auto-refresh parameters are specified, a new refresh group is automatically created to refresh the snapshot with a background process. The local rollback segment, if specified, is associated with this new refresh group. An error is raised if you specify a local rollback segment but do not specify the auto-refresh parameters.

The master rollback segment is stored on a per-snapshot basis. The master rollback segment is validated during snapshot creation and refresh. If the snapshot is complex, the master rollback segment, if specified, is ignored.

Note: Specifying DEFAULT is most useful with ALTER SNAPSHOT. See ALTER SNAPSHOT on page 4-76.

To direct Oracle to select the rollback segment automatically after one has been specified using CREATE SNAPSHOT or ALTER SNAPSHOT, specify DEFAULT with ALTER SNAPSHOT.

Example. The following example creates snapshot SALE_EMP with rollback segment MASTER_SEG at the remote master and rollback segment SNAP_SEG for the local refresh group that contains the snapshot:

```
CREATE SNAPSHOT sales_emp
REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
USING MASTER ROLLBACK SEGMENT master_seg
LOCAL ROLLBACK SEGMENT snap_seg
AS SELECT * FROM bar;
```

The following statement is incorrect and generates an error because it specifies a segment name with a DEFAULT rollback segment:

```
CREATE SNAPSHOT bogus
REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
USING DEFAULT ROLLBACK SEGMENT snap_seg
AS SELECT * FROM faux;
```

Specifying Primary-Key or ROWID Snapshots

To create a primary-key snapshot you must:

- include all columns of the primary key in the snapshot definition
- have an enabled primary-key constraint defined on the snapshot master

To fast refresh primary-key snapshots, you must first create a snapshot master log specifying WITH PRIMARY KEY. The snapshot master log can also record ROWIDs.

Primary-key snapshots are the default if the WITH clause is not specified.

The above conditions *must* be met in order to create a primary-key snapshot.

Example I. The following example creates primary-key snapshot HUMAN_GENOME:

```
CREATE SNAPSHOT human_genome
```

```
   REFRESH FAST START WITH SYSDATE NEXT  SYSDATE + 1/4096
   WITH PRIMARY KEY
AS SELECT * FROM genome_catalog;
```

Example II. The following example creates a ROWID snapshot:

```
CREATE SNAPSHOT emp_data WITH ROWID
AS SELECT * FROM emp_table73;
```

Partitioned Snapshots

Partitioned snapshots are the same as partitioned tables, because snapshots are basically tables. The options have the same syntax and semantics as the partitioned table options for CREATE TABLE and ALTER TABLE. The only difference is that the following operations are not allowed on snapshots and snapshot logs:

- DROP PARTITION
- TRUNCATE PARTITION
- EXCHANGE PARTITION

You cannot perform bulk deletions by dropping or truncating partitions on master tables. Thus, after dropping or truncating a partition, all snapshots must be manually refreshed. A fast refresh will probably produce incorrect results, but Oracle will not raise an error.

Related Topics

ALTER SNAPSHOT on page 4-76
CREATE SNAPSHOT LOG on page 4-297
DROP SNAPSHOT on page 4-402
SELECT on page 4-489

CREATE SNAPSHOT LOG

Purpose

To create a snapshot log. A *snapshot log* is a table associated with the master table of a snapshot. Oracle stores changes to the master table's data in the snapshot log and then uses the snapshot log to refresh the master table's snapshots. See also "Using Snapshot Logs" on page 4-299.

Prerequisites

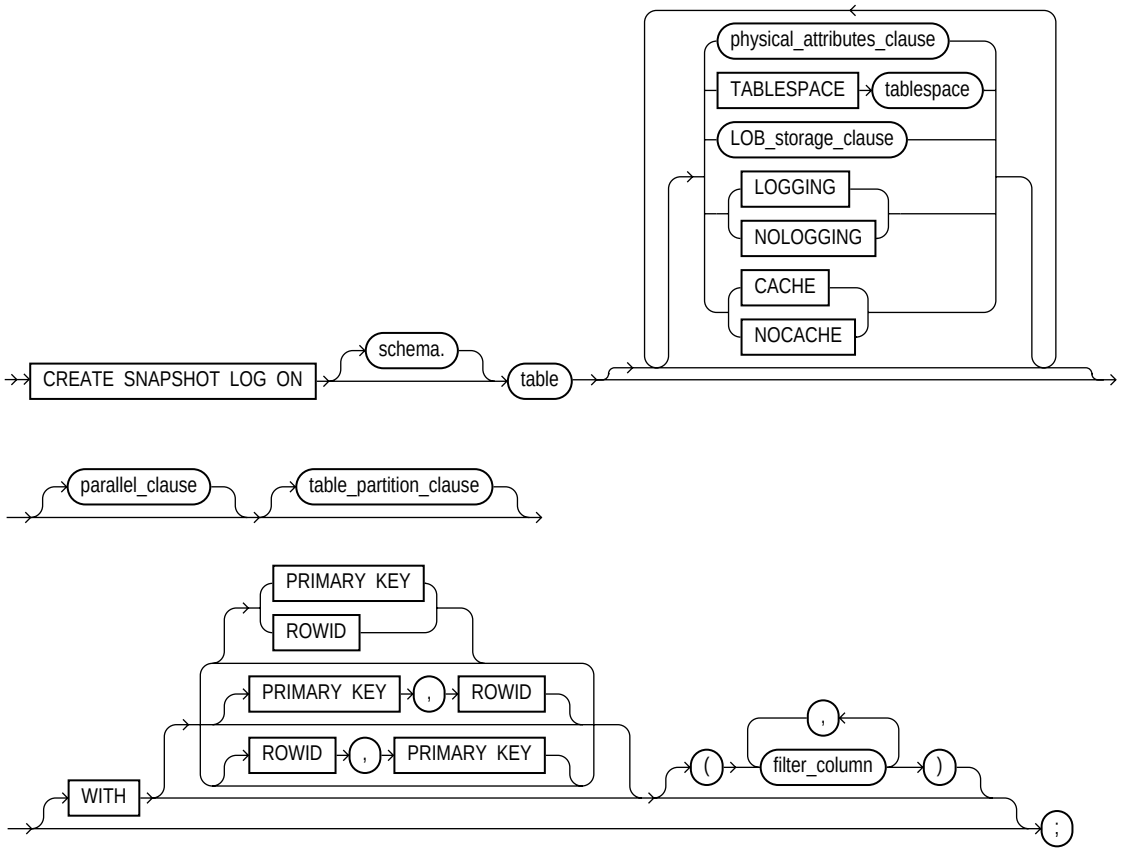
The privileges required to create a snapshot log directly relate to the privileges necessary to create the underlying objects associated with a snapshot log.

- If you own the master table, you can create an associated snapshot log if you have the CREATE TABLE privilege.
- If you are creating a snapshot log for a table in another user's schema, you must have the CREATE ANY TABLE and COMMENT ANY TABLE privileges, as well as either the SELECT privilege for the master table or SELECT ANY TABLE.

In either case, the owner of the snapshot log must have sufficient quota in the tablespace intended to hold the snapshot log.

For detailed information about the prerequisites for creating a snapshot log, see *Oracle8 Replication*.

Syntax



parallel_clause: See PARALLEL clause on page 4-465.

storage_clause: See STORAGE clause on page 4-523.

LOB_storage_clause: See CREATE TABLE on page 4-306.

table_partition_clause: See CREATE TABLE on page 4-306.

physical_attributes_clause: See CREATE TABLE on page 4-306.

Keywords and Parameters

schema is the schema containing the snapshot log's master table. If you omit *schema*, Oracle assumes the master table is contained in your own schema. Oracle creates the snapshot log in the schema of its master table. You cannot create a snapshot log for a table in the schema of the user SYS.

table is the name of the master table for which the snapshot log is to be created. You cannot create a snapshot log for a view.

WITH specifies whether the snapshot log should record the primary key, ROWID, or both primary key and ROWID when rows in the master are updated.

This clause also specifies whether the snapshot log records filter columns—non-primary-key columns referenced by snapshots defined as simple snapshots with subqueries. See also "Recording Primary Keys, ROWIDs, and Filter Columns" on page 4-300.

- PRIMARY KEY — specifies that the primary key of all rows updated should be recorded in the snapshot log.
- ROWID — specifies that the ROWID of all rows updated should be recorded in the snapshot log.
- *filter_column* — is a comma-separated list that specifies the list of filter columns to be recorded in the snapshot log.

Oracle records the primary key of all rows updated in the master by default.

physical_attributes_clause establishes values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics for the snapshot log. See the descriptions of these parameters in CREATE TABLE on page 4-306 and STORAGE clause on page 4-523.

TABLESPACE specifies the tablespace in which the snapshot log is to be created. If you omit this option, Oracle creates the snapshot log in the default tablespace the owner of the snapshot log's schema.

STORAGE establishes storage characteristics for the snapshot log. See the STORAGE clause on page 4-523.

LOB_*storage*_*clause* specifies the LOB storage characteristics. For detailed information about specifying the parameters of the LOB storage clause, see STORAGE clause on page 4-523.

table_partition_clause specifies that the table is partitioned on specified ranges of values. For detailed information about specifying the parameters of the table partition clause, see CREATE TABLE on page 4-306.

Using Snapshot Logs

A *snapshot log* is a table associated with the master table of a snapshot. When changes are made to the master table's data, Oracle adds rows describing these

changes to the snapshot log. Later Oracle can use these rows to refresh snapshots based on the master table. This process is called a *fast refresh*. Without a snapshot log, Oracle must reexecute the snapshot query to refresh the snapshot. This process is called a *complete refresh*. Usually, a fast refresh takes less time than a complete refresh.

A snapshot log is located in the master database in the same schema as the master table. You need only a single snapshot log for a master table. Oracle can use this snapshot log to perform fast refreshes for all simple snapshots based on the master table. For more information on snapshots, including how Oracle refreshes snapshots, see CREATE SNAPSHOT on page 4-286 and *Oracle8 Replication*.

Example . The following statement creates a snapshot log on an employee table that records only primary-key values:

```
CREATE SNAPSHOT LOG ON emp
PCTFREE 5
TABLESPACE users
STORAGE (INITIAL 10K NEXT 10K PCTINCREASE 50);
```

Oracle can use this snapshot log to perform a fast refresh on any simple primary key snapshot subsequently created on the EMP table.

Recording Primary Keys, ROWIDs, and Filter Columns

For Oracle to perform primary-key snapshots, the primary key of updated rows in the master table must be recorded in the snapshot log. Similarly, for ROWID snapshots, the ROWID must be recorded in the snapshot log. Both primary keys and ROWIDs can be recorded to support configurations with both primary-key and ROWID snapshots.

For primary-key snapshots defined as simple snapshots with subqueries, all filter columns referenced by the defining subquery must be recorded in the snapshot log.

Example I. The following examples create snapshot logs that record only the primary keys of updated rows:

```
CREATE SNAPSHOT LOG ON emp;
CREATE SNAPSHOT LOG ON emp WITH PRIMARY KEY;
```

Example II. The following example creates a snapshot log that records both primary keys and ROWIDs of updated rows:

```
CREATE SNAPSHOT LOG ON sales WITH ROWID, PRIMARY KEY;
```

Example III. The following example creates a snapshot log that records primary keys and updates to the filter column ZIP:

```
CREATE SNAPSHOT LOG ON address WITH (zip);
```

Related Topics

ALTER SNAPSHOT LOG on page 4-84
CREATE SNAPSHOT on page 4-286
CREATE TABLE on page 4-306
DROP SNAPSHOT on page 4-402

CREATE SYNONYM

Purpose

To create a synonym. A *synonym* is an alternative name for a table, view, sequence, procedure, stored function, package, snapshot, or another synonym. See also "Using Synonyms" on page 4-303.

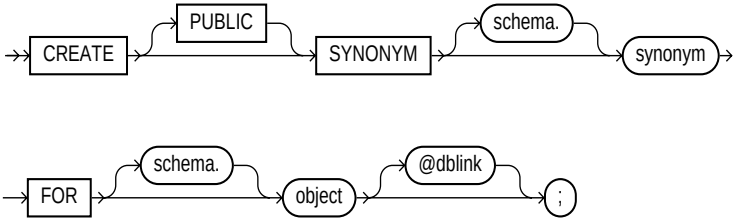
Prerequisites

To create a private synonym in your own schema, you must have CREATE SYNONYM system privilege.

To create a private synonym in another user's schema, you must have CREATE ANY SYNONYM system privilege.

To create a PUBLIC synonym, you must have CREATE PUBLIC SYNONYM system privilege.

Syntax



Keywords and Parameters

PUBLIC	creates a public synonym. Public synonyms are accessible to all users. If you omit this option, the synonym is private and is accessible only within its schema.
schema	is the schema to contain the synonym. If you omit *schema*, Oracle creates the synonym in your own schema. You cannot specify schema if you have specified PUBLIC. See also "Scope of Synonyms" on page 4-304.
synonym	is the name of the synonym to be created.

FOR	identifies the object for which the synonym is created. If you do not qualify object with *schema*, Oracle assumes that the schema object is in your own schema. The schema object can be of the following types: ■ table ■ object table ■ view ■ object view ■ sequence ■ stored procedure, function, or package ■ snapshot ■ synonym OBJ You can create a synonym for an object table or an object view, but not for object types. The schema object cannot be contained in a package. Note that the schema object need not currently exist and you need not have privileges to access the object.
dblink	You can use a complete or partial *dblink* to create a synonym for a schema object on a remote database where the object is located. For more information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-54. If you specify *dblink* and omit *schema*, 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*, Oracle assumes the object is located on the local database.

Using Synonyms

You can use a synonym to stand for its base object in any of the following statements:

DML Statements	DDL Statements
SELECT	AUDIT
INSERT	NOAUDIT
UPDATE	GRANT
DELETE	REVOKE
EXPLAIN PLAN	COMMENT
LOCK TABLE	

Synonyms are used for security and convenience. Creating a synonym for an object allows you to:

- reference the object without specifying its owner
- reference the object without specifying the database on which it is located
- provide another name for the object

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.

Scope of Synonyms

A private synonym name must be unique in its schema. Oracle attempts to resolve references to objects at the schema level before resolving them at the PUBLIC synonym level. Oracle uses a public synonym only when resolving references to an object if both of the following cases are true:

- the object is not prefaced by a schema
- the object is not followed by a database link

For example, assume the schemas SCOTT and BLAKE each contain tables named DEPT and the user SYSTEM creates a PUBLIC synonym named DEPT for BLAKE.DEPT. If the user SCOTT then issues the following statement, Oracle returns rows from SCOTT.DEPT:

```
SELECT *
   FROM dept;
```

To retrieve rows from BLAKE.DEPT, the user SCOTT must preface DEPT with the schema name:

```
SELECT *
   FROM blake.dept;
```

If the user ADAM's schema does not contain an object named DEPT, then ADAM can access the DEPT table in BLAKE's schema by using the public synonym DEPT:

```
SELECT *
   FROM dept;
```

Example I. To define the synonym MARKET for the table MARKET_RESEARCH in the schema SCOTT, issue the following statement:

```
CREATE SYNONYM market
   FOR scott.market_research;
```

Example II. To create a PUBLIC synonym for the EMP table in the schema SCOTT on the remote SALES database, you could issue the following statement:

```
CREATE PUBLIC SYNONYM emp
   FOR scott.emp@sales;
```

Note that a synonym may have the same name as the base table, provided the base table is contained in another schema.

Related Topics

CREATE DATABASE LINK on page 4-225
CREATE TABLE on page 4-306
CREATE VIEW on page 4-363

CREATE TABLE

Purpose

To create a *table*, the basic structure to hold user data, specifying the following information:

- column definitions
- table organization definition
- OBJ column definitions using objects
- integrity constraints
- the table's tablespace
- storage characteristics
- an optional cluster
- data from an arbitrary query
- degree of parallelism used to create the table and the default degree of parallelism for queries on the table
- partitioning definitions
- index-organized or heap-organized

For illustrations of some of these purposes, "Examples" on page 4-321.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

OBJ Use CREATE TABLE to create an object table or a table that uses an object type for a column definition. An *object table* is a table explicitly defined to hold object instances of a particular type.

OBJ You can also create an object type and then use it in a column when creating a relational table. For more information about creating objects, see *Oracle8 Application Developer's Guide* and CREATE TYPE on page 4-345.

Prerequisites

To create a relational table in your own schema, you must have CREATE TABLE system privilege. To create a table in another user's schema, you must have

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 UNLIMITED TABLESPACE system privilege.

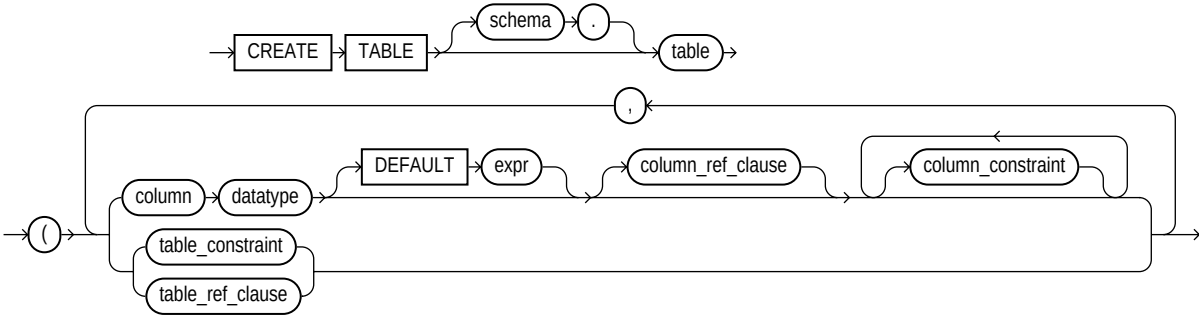
OBJ In addition to the table privileges above, to create a table that uses types, the owner of the table must have the EXECUTE object privilege in order to access all types referenced by the table, or you must have the EXECUTE ANY TYPE system privilege. These privileges must be granted explicitly and not acquired through a role.

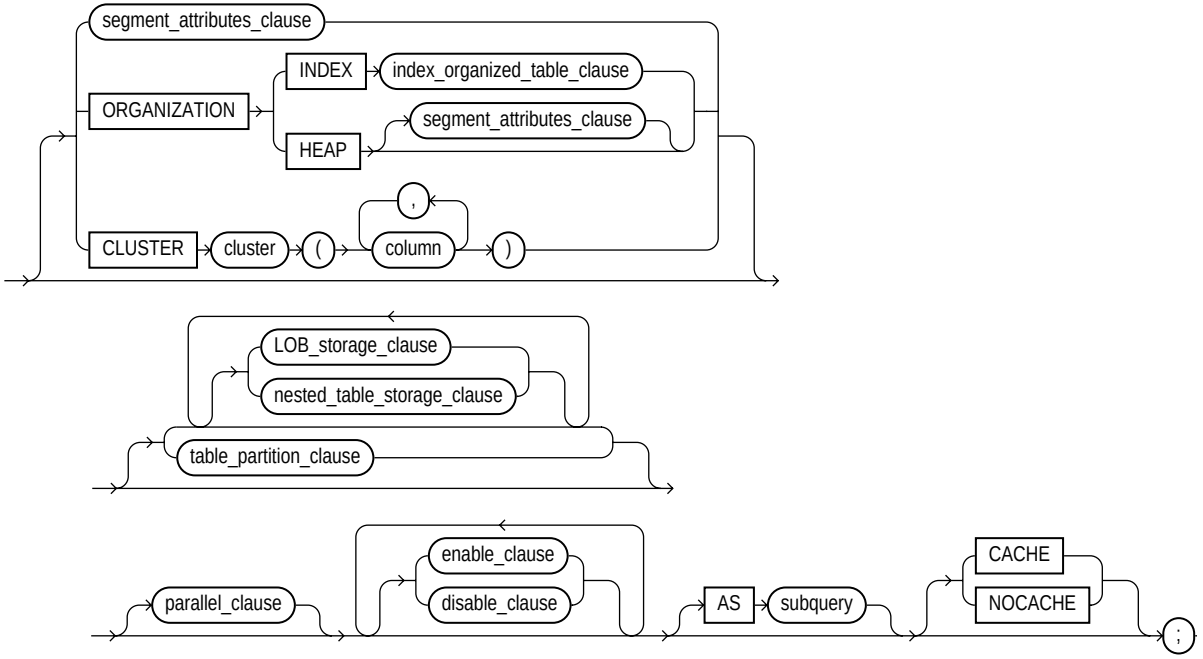
Additionally, if the table owner intends to grant access to the table to other users, the owner must have been granted the EXECUTE privileges to the referenced types with the GRANT OPTION, or have the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, the table owner has insufficient privileges to grant access on the table to other users.

For more information about the privileges required to create tables using types, see *Oracle8 Application Developer's Guide*.

Syntax

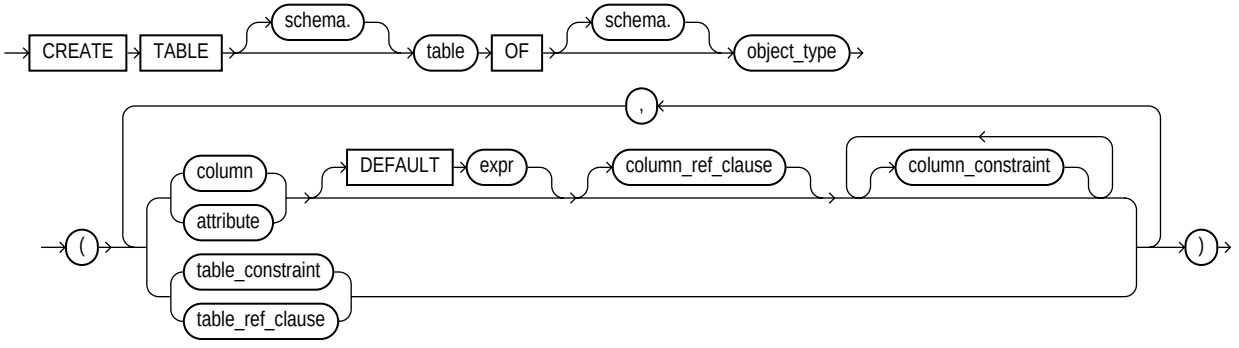
Relational table definition ::=

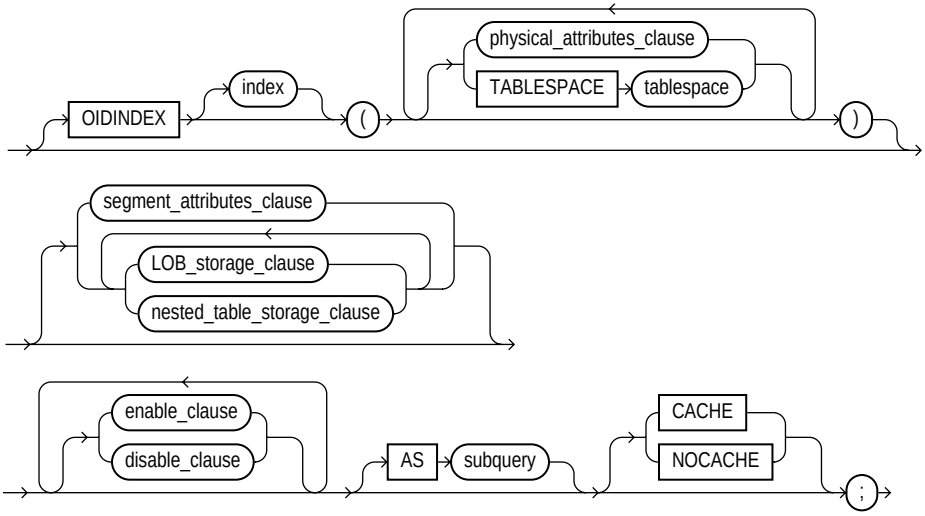




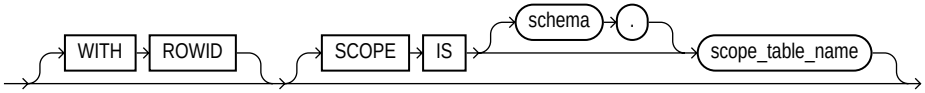
,

OBJ Object table definition ::=

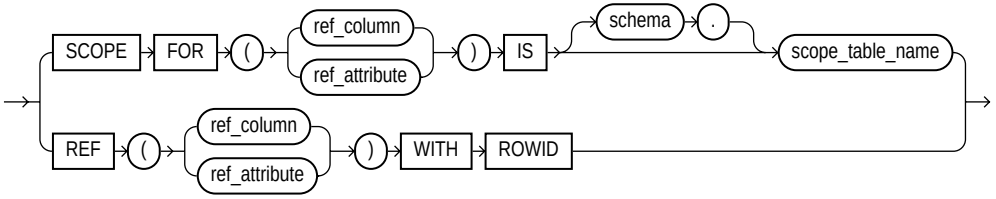




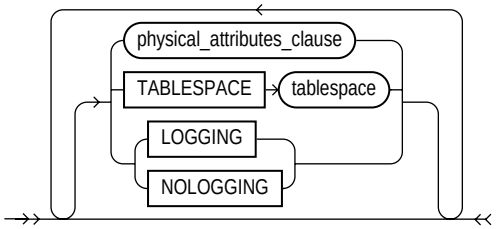
column_ref_clause::=



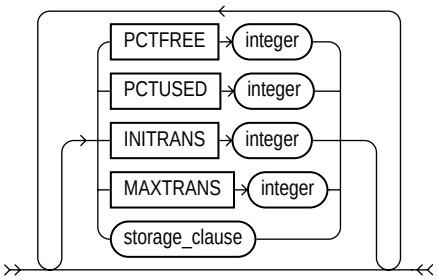
table_ref_clause::=



segment_attributes_clause::=

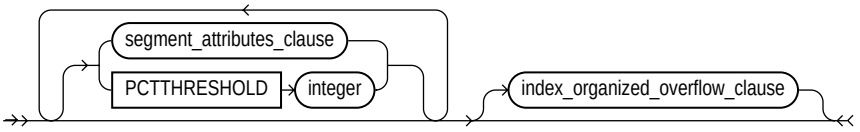


physical_attributes_clause::=

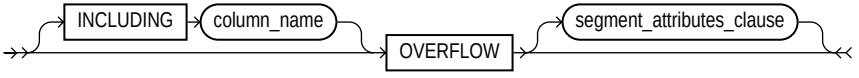


storage_clause: See the STORAGE clause on page 4-523.

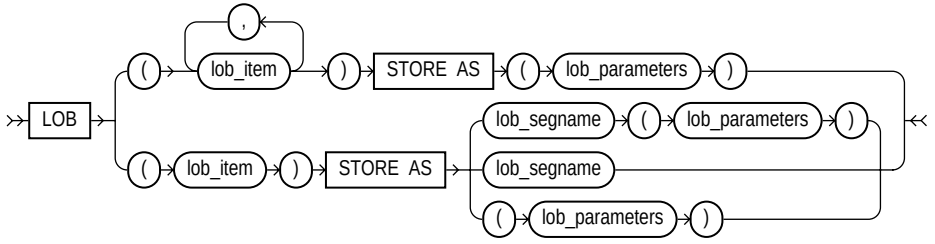
index_organized_table_clause ::=



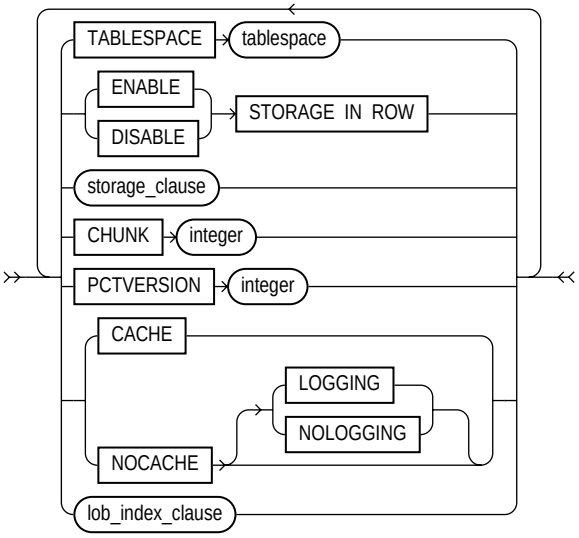
index_organized_overflow_clause ::=



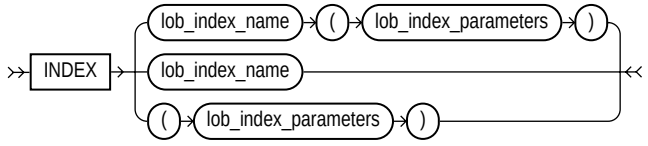
LOB_storage_clause ::=



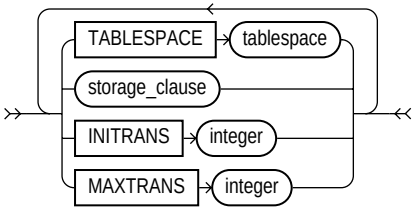
lob_parameters ::=



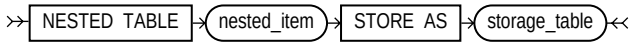
lob_index_clause ::=



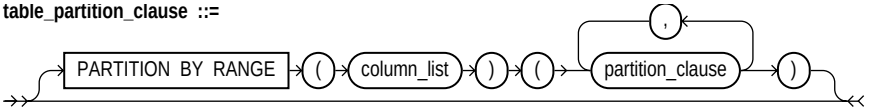
lob_index_parameters ::=



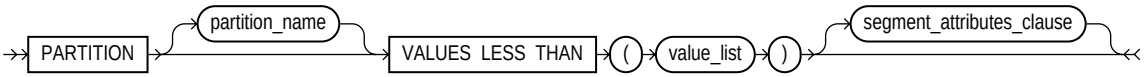
nested_table_storage_clause ::=



table_partition_clause ::=



partition_clause ::=



disable_clause: See the DISABLE clause on page 4-380.

enable_clause: See the ENABLE clause on page 4-417.

parallel_clause: See the PARALLEL clause on page 4-465

storage_clause: see STORAGE clause on page 4-523

subquery: See "Subqueries" on page 4-530

Keywords and Parameters

schema	is the schema to contain the table. If you omit *schema*, Oracle creates the table in your own schema.
table	is the name of the table (or object table) to be created. A partitioned *table* cannot be a clustered table or an object table.
OBJ OF *object_type*	explicitly creates 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 (OID) when a row is inserted. If you omit schema, Oracle creates the object table in your own schema. For more information about creating objects, see CREATE TYPE on page 4-345. See also "Object Tables" on page 4-324.
column	specifies the name of a column of the table. A table can have up to 1000 columns. You may omit column definitions only when using the AS subquery clause. See also "LOB Column Example" on page 4-322.
OBJ attribute	specifies the qualified column name of an item in an object.

datatype is the datatype of a column. Oracle-supplied datatypes are defined in "Datatypes" on page 2-5. You can omit the datatype only if the statement also designates the column as part of a foreign key in a referential integrity constraint. Oracle automatically assigns to the column the datatype of the corresponding column of the referenced key of the referential integrity constraint.

OBJ Object types, REF *object_type*, VARRAYs, and nested tables are valid datatypes. See also "REFs" on page 4-325.

DEFAULT specifies a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The datatype of the expression must match the datatype of the column. The column must also be long enough to hold this expression. For the syntax of *expr*, see "Expressions" on page 3-78. A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.

column_ref_clause lets you further specify a column of type REF:

OBJ WITH ROWID — stores the ROWID and the REF value in *column* or *attribute*. Storing a REF value with a ROWID can improve the performance of dereferencing operations, but will also use more space. Default storage of REF values is without ROWIDs.

OBJ SCOPE IS *scope_table_name* — restricts the scope of the column REF values to *scope_table_name*. The REF values for the column must come from REF values obtained from the object table specified in the clause. You can only specify one scope table per REF column.

The *scope_table_name* is the name of the object table in which object instances (of the same type as the REF column) are stored. The values in the REF column point to objects in the scope table.

You must have SELECT privileges on the table or SELECT ANY TABLE system privileges.

column_constraint defines an integrity constraint as part of the column definition. See the syntax description of column_constraint in the CONSTRAINT clause on page 4-188.

table_constraint defines an integrity constraint as part of the table definition. See the syntax description of *table_constraint* in the CONSTRAINT clause on page 4-188.

table_ref_clause

OBJ SCOPE FOR... IS... — restricts the scope of the REF values in *ref_column* or *ref_attribute* to *scope_table_name*. The REF values for the column or attribute must come from REF values obtained from the object table specified in the clause.

The *ref_column* or *ref_attribute* is the name of a REF column in an object table or an embedded REF attribute within an object column of a relational table. The values in the REF column or attribute point to objects in the scope table.

OBJ REF — is a reference to a row in an object table. You can specify either a REF column name of an object or relational table (*ref_column*) or an embedded REF attribute within an object column (*ref_attribute*).

OBJ OIDINDEX — specifies an index on the hidden object identifier column and / or the storage specification for the index. Either *index* or *storage_specification* must be specified.

OBJ *index* — is the name of the index on the hidden object identifier column. If not specified, Oracle generates a name.

physical_attributes_clause — specifies the value of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics of the table.

Note: For a nonpartitioned table, each parameter and storage characteristic specified is the actual physical attribute of the segment associated with the table. For partitioned tables, the value specified for the parameter or storage characteristic is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and in subsequent ALTER TABLE ADD PARTITION statements), unless you explicitly override that value in the PARTITION clause of this command.

PCTFREE — specifies the percentage of space in each data block of the table, object table OIDINDEX, or partition reserved for future updates to the table's rows. The value of PCTFREE must be a value from 0 to 99. A value of 0 allows the entire block to 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 PARTITION description clause and in the commands that create and alter clusters, indexes, snapshots, and snapshot logs. The combination of PCTFREE and PCTUSED determines whether new rows will be inserted into existing data blocks or into new blocks.

PCTUSED — specifies the minimum percentage of used space that Oracle maintains for each data block of the table, object table OIDINDEX, or index-organized table overflow data segment. A block becomes a candidate for row insertion when its used space falls below PCTUSED. PCTUSED is specified as a positive integer from 1 to 99 and defaults to 40.

PCTUSED has the same function in the PARTITION description clause and in the commands that create and alter clusters, snapshots, and snapshot logs.

PCTUSED is not a valid table storage characteristic for an index-organized table (ORGANIZATION INDEX).

	The sum of PCTFREE and PCTUSED must be less than 100. You can use PCTFREE and PCTUSED together to utilize space within a table more efficiently. For information on the performance effects of different values PCTUSED and PCTFREE, see *Oracle8 Tuning*.
INITRANS	specifies the initial number of transaction entries allocated within each data block allocated to the table, object table OIDINDEX, partition, LOB index segment, or overflow data segment. This value can range from 1 to 255 and defaults to 1. 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. The size of a transaction entry depends on your operating system. 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 PARTITION description clause, clusters, indexes, snapshots, and snapshot logs as in tables. The minimum and default INITRANS value for a cluster or index is 2, rather than 1.
MAXTRANS	specifies the maximum number of concurrent transactions that can update a data block allocated to the table, object table OIDINDEX, partition, LOB index segment, or index-organized overflow data segment. This limit does not apply to queries. This value can range from 1 to 255 and the default is a function of the data block size. You should not change the MAXTRANS value from its default. If the number concurrent transactions updating a block exceeds the INITRANS value, Oracle dynamically allocates transaction entries in the block until either the MAXTRANS value is exceeded or the block has no more free space. The MAXTRANS parameter serves the same purpose in the PARTITION description clause, clusters, snapshots, and snapshot logs as in tables.
storage_clause	specifies the storage characteristics for the table, object table OIDINDEX, partition, LOB storage, LOB index 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. See the STORAGE clause on page 4-523.
TABLESPACE	specifies the tablespace in which Oracle creates the table, object table OIDINDEX, partition, LOB storage, LOB index segment, or index-organized table overflow data segment. If you omit this option, then Oracle creates that item in the default tablespace of the owner of the schema containing the table. 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 clause.

LOGGING / NOLOGGING

specifies whether the creation of the table (and any indexes required because of constraints), partition, or LOB storage characteristics will be logged in the redo log file. It also specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against the table, partition, or LOB storage are logged (LOGGING) or not logged (NOLOGGING).

If you omit the LOGGING / NOLOGGING clause, the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides. For LOBs, if you omit the LOGGING / NOLOGGING clause,

- if you specify CACHE, then LOGGING is used (because you cannot have CACHE NOLOGGING);
- otherwise, the logging attribute defaults to the logging attribute of the tablespace in which it resides.

For nonpartitioned tables, the value specified for LOGGING is the actual physical attribute of the segment associated with the table. For partitioned tables, the logging attribute value specified is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and in subsequent ALTER TABLE ADD PARTITION statements), unless you specify LOGGING / NOLOGGING in the PARTITION description clause.

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 this table, you should take a backup after the NOLOGGING operation.

The size of a redo log generated for an operation in NOLOGGING mode is significantly smaller than the log generated with the LOGGING option set.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation restores the table. However, media recovery from a backup taken before the NOLOGGING operation does not restore the table.

The logging attribute of the table is independent of that of its indexes.

NOLOGGING is not a valid keyword for creating index-organized tables.

For more information about the LOGGING option and Parallel DML, see *Oracle8 Concepts* and *Oracle8 Administrator's Guide*.

Note: In future versions of Oracle, the LOGGING keyword will replace the RECOVERABLE option. RECOVERABLE is still available as a valid keyword in Oracle when creating nonpartitioned tables, however, it is not recommended.

ORGANIZATION INDEX

specifies 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. See also "Index-Organized Tables" on page 4-323.

ORGANIZATION HEAP	specifies that the data rows of *table* are stored in no particular order. This is the default.	
index_organized_table_clause	PCTTHRESHOLD *integer*	specifies the percentage of space reserved in the index block for an index-organized table row. Any portion of the row that exceeds the specified threshold is stored in the area. PCTTHRESHOLD must be a value from 0 to 50.
	OVERFLOW	specifies that index-organized table data rows exceeding the specified threshold are placed in the data segment listed in this clause. If OVERFLOW is not specified, then rows exceeding the PCTTHRESHOLD limit are rejected.
	INCLUDING *column_name*	specifies a column at which to divide an index-organized table row into index and overflow portions. All columns that follow *column_name* are stored in the overflow data segment. A *column_name* is either the name of the last primary-key column or any nonprimary-key column.
RECOVERABLE	is a deprecated option. RECOVERABLE is not a valid keyword for creating partitioned tables or LOB storage characteristics.	
UNRECOVERABLE	is a deprecated option. It specifies that the creation of the table (and any indices required because of constraints) will not be logged in the redo log file. This keyword can only be specified with the AS subquery clause. UNRECOVERABLE is not a valid keyword for creating partitioned or index-organized tables. **Note:** In future versions of Oracle, the LOGGING keyword will replace the RECOVERABLE option. RECOVERABLE is still available as a valid keyword in Oracle when creating nonpartitioned tables, however, it is not recommended.	
LOB_storage_clause	LOB	specifies the LOB storage characteristics. For detailed information about LOBs, see *Oracle8 Application Developer's Guide.*
	lob_item	is 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.
STORE AS	*lob_segname*	specifies the name of the LOB data segment. You cannot use *lob_segname* if you specify more than one *lob_item*.
lob_parameters	ENABLE STORAGE IN ROW	specifies that the LOB value is stored in the row (in-line) if its length is less than approximately 4000 bytes minus system control information. This is the default.
	DISABLE STORAGE IN ROW	specifies that the LOB value is stored outside of the row regardless of the length of the LOB value.

Note that the LOB locator is always stored in the row regardless of where the LOB value is stored. You cannot change the value of STORAGE IN ROW once it is set.

CHUNK *integer* — specifies the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, Oracle allocates 4096 bytes (2 blocks). The maximum value is 32768 (32K), which is the largest Oracle block size allowed.

Note: The value of CHUNK must be less than or equal to the values of both INITIAL and NEXT (either the default values or those specified in the storage clause). If CHUNK exceeds the value of either INITIAL or NEXT, Oracle returns an error.

PCTVERSION *integer* — is the maximum percentage of overall LOB storage space used for creating new versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until 10% of the overall LOB storage space is used.

INDEX *lob_index_name* — is the name of the LOB index segment. You cannot specify *lob_index_name* if you specify more than one *lob_item* in the associated lob_item list. Note that you cannot alter the LOB index through the ALTER INDEX statement. You can alter a LOB index specification only through the ALTER TABLE statement (see ALTER TABLE on page 4-106).

Note also that a user cannot drop the LOB index. It is a system index created and maintained by the system.

OBJ NESTED TABLE ... STORE AS ... — specifies *storage_table* as the name of the storage table in which the rows of all *nested_item* values reside. You must include this clause when creating a table with columns or column attributes whose type is a nested table. See also "Nested Table Storage" on page 4-325

- *nested_item* is the name of a column or a column-qualified attribute whose type is a nested table.
- *storage_table* is the name of the storage table. The storage table is created in the same schema and the same tablespace as the parent table.

CLUSTER — specifies that the table is to be part of the *cluster*. The columns listed in this clause are the table columns that correspond to the cluster's 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.

Specify one column from the table for each column in the cluster key. The columns are matched by position, not by name.

A clustered table uses the cluster's space allocation. Therefore, do not use the PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameters, the TABLESPACE option, or the STORAGE clause with the CLUSTER option.

OBJ Object tables cannot be part of a cluster.

table_partitioning_clause:

PARTITION BY RANGE — specifies that the table is partitioned on ranges of values from *column_list*. See also "Partitioned Tables" on page 4-323.

column_list — is an ordered list of columns used to determine into which partition a row belongs. You cannot specify more than 16 columns in *column_list*. The *column_list* cannot contain the ROWID pseudocolumn or any columns of datatype ROWID or LONG.

PARTITION *partition_name* — specifies the physical partition clause. If *partition_name* is omitted, Oracle generates a name with the form SYS_P*n* for the partition. The *partition_name* must conform to the rules for naming schema objects and their part as described in "Schema Object Naming Rules" on page 2-47.

VALUES LESS THAN — specifies the noninclusive upper bound for the current partition.

value_list — is an ordered list of literal values corresponding to *column_list* in the PARTITION BY RANGE clause. You can substitute the keyword MAXVALUE for any literal in *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. See *Oracle8 Concepts* for more information about partition bounds.

parallel_clause — specifies the degree of parallelism for creating the table and the default degree of parallelism for queries on the table once created.

This is not a valid option when creating index-organized tables. For more information, see PARALLEL clause on page 4-465.

enable_clause — enables an integrity constraint. See the ENABLE clause on page 4-417.

disable_clause — disables an integrity constraint. See the DISABLE clause on page 4-380.

Constraints specified in the ENABLE and DISABLE clauses of a CREATE TABLE statement must be defined in the statement. You can also enable and disable constraints with the ENABLE and DISABLE keywords of the CONSTRAINT clause. If you define a constraint but do not explicitly enable or disable it, Oracle enables it by default.

	You cannot use the ENABLE and DISABLE clauses in a CREATE TABLE statement to enable and disable triggers.
AS *subquery*	inserts the rows returned by the subquery into the table upon its creation. See "Subqueries" on page 4-530. **Note:** This subquery is not supported for index-organized tables with overflow. 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 datatypes. Oracle derives datatypes and lengths from the subquery. Oracle also follows the following rules for integrity constraints: ■ Oracle automatically defines any NOT NULL constraints on columns in the new table that existed on the corresponding columns of the selected table if the subquery selects the column rather than an expression containing the column. ■ A CREATE TABLE statement cannot contain both an AS clause and a referential integrity constraint definition. ■ If a CREATE TABLE statement contains both the AS clause and a CONSTRAINT clause or an ENABLE clause with the EXCEPTIONS option, Oracle ignores the EXCEPTIONS option. If any rows violate the constraint, Oracle does not create the table and returns an error message. If all expressions in the subquery are columns, rather than expressions, 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 the subquery. OBJ 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.
CACHE	for data that will be accessed frequently, 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 option is useful for small lookup tables. As a parameter in the LOB storage clause, CACHE specifies that Oracle preallocates and retains LOB data values in memory for faster access. This is not a valid keyword when creating index-organized tables.
NOCACHE	for data that will not be accessed frequently, specifies 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. For LOBs, the LOB value either is not brought into the buffer cache or is brought into the buffer cache and placed at the least recently used end of the LRU list. This is the default behavior except when creating index-organized tables. This is not a valid keyword when creating index-organized tables. As a parameter in the LOB storage clause, NOCACHE specifies that LOB values are not preallocated in memory. This is the LOB storage default.

Examples

Tables are created with no data unless a query is specified. You can add rows to a table with the INSERT command.

After creating a table, you can define additional columns, partitions, and integrity constraints with the ADD clause of the ALTER TABLE command. You can change the definition of an existing column or partition with the MODIFY clause of the ALTER TABLE command. To modify an integrity constraint, you must drop the constraint and redefine it.

Example I. To define the EMP table owned by SCOTT, you could issue the following statement:

```
CREATE TABLE scott.emp
   (empno     NUMBER        CONSTRAINT pk_emp PRIMARY KEY,
    ename     VARCHAR2(10)  CONSTRAINT nn_ename NOT NULL
                            CONSTRAINT upper_ename
CHECK (ename = UPPER(ename)),
    job        VARCHAR2(9),
    mgr      NUMBER         CONSTRAINT fk_mgr
                            REFERENCES scott.emp(empno),
    hiredate  DATE          DEFAULT SYSDATE,
    sal       NUMBER(10,2)  CONSTRAINT ck_sal
CHECK (sal > 500),
       comm      NUMBER(9,0)   DEFAULT NULL,
       deptno    NUMBER(2)     CONSTRAINT nn_deptno NOT NULL
                               CONSTRAINT fk_deptno
                               REFERENCES scott.dept(deptno) )
PCTFREE 5 PCTUSED 75;
```

This table contains 8 columns. The EMPNO column is of datatype NUMBER and has an associated integrity constraint named PK_EMP. The HIRDEDATE column is of datatype DATE and has a default value of SYSDATE, and so on.

This table definition specifies a PCTFREE of 5 and a PCTUSED of 75, which is appropriate for a relatively static table. The definition also defines integrity constraints on some columns of the EMP table.

Example II. To define the sample table SALGRADE in the HUMAN_RESOURCE tablespace with a small storage capacity and limited allocation potential, issue the following statement:

```
CREATE TABLE salgrade
   ( grade  NUMBER  CONSTRAINT pk_salgrade
```

```
                    PRIMARY KEY
                    USING INDEX TABLESPACE users_a,
     losal  NUMBER,
     hisal  NUMBER )
TABLESPACE human_resource
STORAGE (INITIAL     6144
         NEXT        6144
         MINEXTENTS     1
         MAXEXTENTS     5
         PCTINCREASE    5);
```

The above statement also defines a PRIMARY KEY constraint on the GRADE column and specifies that the index Oracle creates to enforce this constraint is created in the USERS_A tablespace.

For more examples of defining integrity constraints, see the CONSTRAINT clause on page 4-188. For examples of enabling and disabling integrity constraints, see the ENABLE clause on page 4-417 and the DISABLE clause on page 4-380.

Example III. When using parallel query, the fastest way to create a table that has the same columns as the EMP table, but only for those employees in department 10, is to issue a command similar to the following:

```
CREATE TABLE emp_tmp
   NOLOGGING
   PARALLEL (DEGREE 3)
   AS SELECT * FROM emp WHERE deptno = 10;
```

Using parallelism speeds up the creation of the table because Oracle uses three processes 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.

LOB Column Example

The following example creates table LOB_TAB with two LOB columns and specifies the LOB storage characteristics:

```
CREATE TABLE lob_tab (col1 BLOB, col2 CLOB)
STORAGE (INITIAL 256 NEXT 256)
LOB (col1, col2) STORE AS
   (TABLESPACE lob_seg_ts
    STORAGE (INITIAL 6144 NEXT 6144)
    CHUNK 4
    NOCACHE LOGGING
       INDEX (TABLESPACE lob_index_ts
```

```
        STORAGE (INITIAL 256 NEXT 256)
    )
);
```

Index-Organized Tables

Index-organized tables are special kinds of tables that keep data sorted on the primary key and are therefore best suited for primary key-based access and manipulation.

An index-organized table is an alternative to

- a nonclustered table indexed on the primary key by using the CREATE INDEX command
- a clustered table stored in an indexed cluster that has been created using the CREATE CLUSTER command that maps the primary key for the table to the cluster key

Index-organized tables differ from other kinds of tables in that Oracle maintains the table rows in a B*-tree index built on the primary key. However, the index row contains both the primary key column values and the associated non-key column values for the corresponding row.

You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row. Use the primary key instead of the ROWID for directly accessing index-organized rows.

Example. The following statement creates an index-organized table:

```
CREATE TABLE docindex
  ( token CHAR(20),
    doc_oid INTEGER,
    token_frequency SMALLINT,
    token_occurrence_data VARCHAR(512),
       CONSTRAINT pk_docindex PRIMARY KEY (token, doc_oid) )
  ORGANIZATION INDEX TABLESPACE text_collection
  PCTTHRESHOLD 20 INCLUDING token_frequency
  OVERFLOW TABLESPACE text_collection_overflow;
```

Partitioned Tables

A partitioned table consists of a number of pieces all of which have the same logical attributes. For example, all partitions share the same column and constraint definitions.

You can create a partitioned table with just one partition. Note, however, that a partitioned table with one partition is different from a nonpartitioned table. For instance, you cannot add a partition to a nonpartitioned table.

Example. The following example creates a table with three partitions:

```
CREATE TABLE stock_xactions
   (stock_symbol CHAR(5),
    stock_series CHAR(1),
    num_shares NUMBER(10),
    price NUMBER(5,2),
    trade_date DATE)
STORAGE (INITIAL 100K NEXT 50K) LOGGING
PARTITION BY RANGE (trade_date)
  (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
     TABLESPACE ts0 NOLOGGING,
   PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
     TABLESPACE ts1,
   PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
     TABLESPACE ts2);
```

For information about partitioned table maintenance operations, see the *Oracle8 Administrator's Guide*.

OBJ Object Tables

In order to have Oracle assign an object identifier to an object, the object must reside in a special kind of table called an object table. Objects residing in an object table are referenceable. For more information about using REFs, see "User-Defined Types" on page 2-22, "User Functions" on page 3-60, "Expressions" on page 3-78, CREATE TYPE on page 4-345, and *Oracle8 Administrator's Guide*

The columns of an object table correspond to the top-level attributes of the corresponding type. Each row will contain an object instance, and each instance will be assigned a unique, system-generated object identifier (OID) when a row is inserted.

Example I. For example, consider object type DEPT_T:

```
CREATE TYPE dept_t AS OBJECT
( dname VARCHAR2(100),
  address VARCHAR2(200) );
```

Object table DEPT holds department objects of type DEPT_T:

```
CREATE TABLE dept OF dept_t;
```

Example II. OBJ The following example creates object table SALESREPS with a user-defined object type, SALESREP_T:

```
CREATE OR REPLACE TYPE salesrep_t AS OBJECT
  ( repId NUMBER,
    repName VARCHAR2(64));
CREATE TABLE salesreps OF salesrep_t;
```

OBJ Nested Table Storage

Creating a table with columns of type TABLE implicitly creates a storage table for each nested table column. The storage table is created in the same tablespace as its parent table (using the default storage characteristics) and stores the nested table values of the column for which it was created.

You *cannot* query or perform DML statements on the storage table directly, but you can modify the nested table column storage characteristics by using the name of storage table in an ALTER TABLE statement. For information about modifying nested table column storage characteristics, see ALTER TABLE on page 4-106.

Example. The following example creates relational table EMPLOYEE with a nested table column PROJECTS:

```
CREATE TABLE employee (empno NUMBER, name CHAR(31),
projects PROJ_TABLE_TYPE)
NESTED TABLE projects STORE AS nested_proj_table;
```

OBJ REFs

A REF value is a reference to a row in an object table. A table can have top-level REF columns or REF attributes embedded within an object type column. In general, if a table has a REF column, each REF value in the column could reference a row in a different object table. A SCOPE clause restricts the scope of references to a single table.

For example, if you create an object table DEPT which stores all the departments in an organization, you could then create table EMP that contains a REF column (E_DEPT) to point to the department in which each employee works. Because all employees work in some department stored in the DEPT table, a scope clause can be specified on the E_DEPT column of EMP to restrict the scope of references to the DEPT table.

You can increase the performance of queries with dereference operations and decrease the amount of storage needed for REF values by using the scope clause. Note that a SCOPE clause does not have the same semantics as referential constraints. Referential constraints do not allow dangling references. Also, referential constraints do not necessarily restrict the scope of references to a single table (one can specify multiple referential constraints on the same foreign key, with each one of them pointing to a different table).

You can also store REF values with or without ROWIDs. Storing REF values WITH ROWID can enhance the performance of dereference operations, but takes up more space. The default behavior is to store REF values without the ROWID.

You cannot specify REF clauses on REF columns in nested tables using the CREATE TABLE statement. To specify REF clauses on REF columns in nested tables, use the ALTER TABLE to modify the nested table 's storage table.

Example. The following example creates object type DEPT_T and object table DEPT to store instances of all departments. A table with a scoped REF is then created.

```
CREATE TYPE dept_t AS OBJECT
( dname VARCHAR2(100),
  address VARCHAR2(200) );

CREATE TABLE dept OF dept_t;

CREATE TABLE emp
( ename VARCHAR2(100),
  enumber NUMBER,
  edept REF dept_t SCOPE IS dept );
```

OBJ Constraints on Object Type Columns

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.

Example.

```
CREATE TYPE address 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,
    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') );
```

Related Topics

CREATE TYPE on page 4-345
CREATE CLUSTER on page 4-207
CREATE CLUSTER on page 4-207
CREATE TABLESPACE on page 4-328
CREATE TYPE on page 4-345
DROP TABLE on page 4-405
CONSTRAINT clause on page 4-188
DISABLE clause on page 4-380
ENABLE clause on page 4-417
PARALLEL clause on page 4-465
STORAGE clause on page 4-523

CREATE TABLESPACE

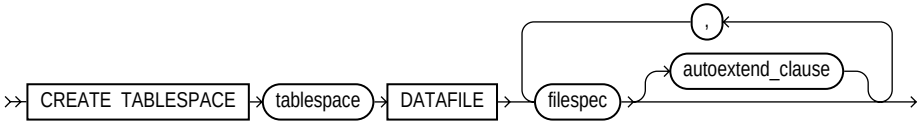
Purpose

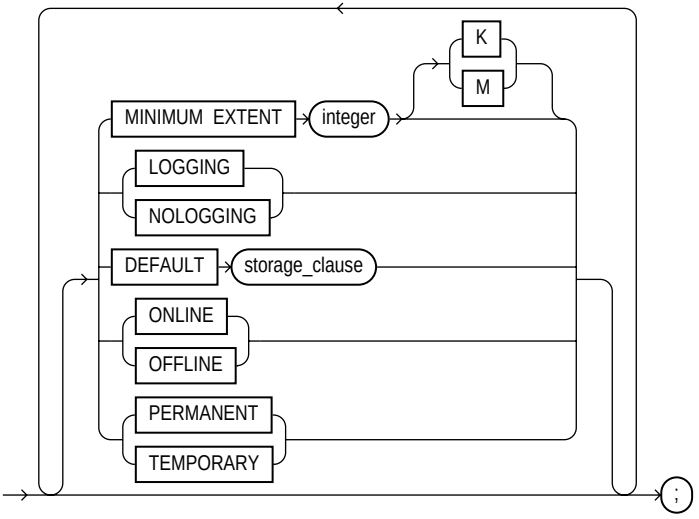
To create a tablespace. A *tablespace* is an allocation of space in the database that can contain schema objects. See also "About Tablespaces" on page 4-330.

Prerequisites

You must have CREATE TABLESPACE system privilege. Also, the SYSTEM tablespace must contain at least two rollback segments including the SYSTEM rollback segment.

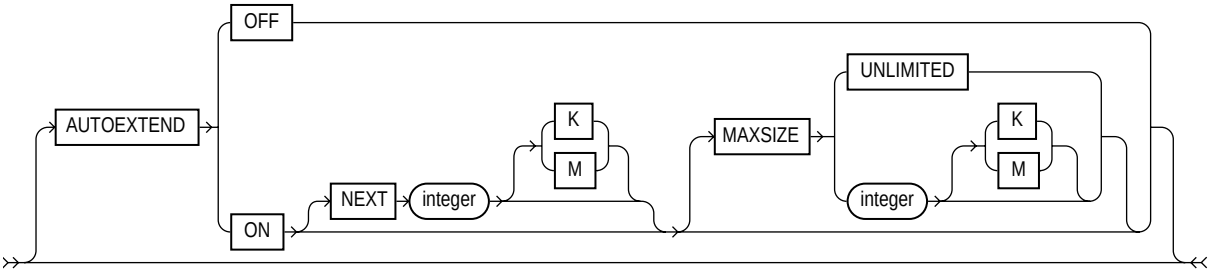
Syntax





filespec: See "Filespec" on page 4-431.

autoextend_clause::=



storage_clause: See STORAGE clause on page 4-523.

Keywords and Parameters

tablespace is the name of the tablespace to be created.

DATAFILE *filespec* specifies the datafile or files to make up the tablespace. See "Filespec" on page 4-431.

autoextend_clause enables or disables the automatic extension of the datafile.

OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND commands.
ON	enables autoextend.
NEXT	specifies the disk space to allocate to the datafile when more extents are required.
MAXSIZE	specifies the maximum disk space allowed for allocation to the datafile.
UNLIMITED	sets no limit on allocating disk space to the datafile.

MINIMUM EXTENT *integer* controls free space fragmentation in the tablespace by ensuring that every used and/or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*. For more information about using MINIMUM EXTENT to control space fragmentation, see *Oracle8 Administrator's Guide*.

LOGGING/ NOLOGGING specifies the default logging attributes of all tables, index, and partitions within the tablespace. LOGGING is the default.

The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Only the following operations support the NOLOGGING mode:

DML:

- direct-load INSERT (serial or parallel)
- Direct Loader (SQL*Loader)

DDL:

- CREATE TABLE ... AS SELECT
- CREATE INDEX
- ALTER INDEX ... REBUILD
- ALTER INDEX ... REBUILD PARTITION
- ALTER INDEX ... SPLIT PARTITION
- ALTER TABLE ... SPLIT PARTITION
- ALTER TABLE ... MOVE PARTITION

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 logged. Therefore, if you cannot afford to lose the object, you should take a backup after the NOLOGGING operation.

DEFAULT *storage_clause*	specifies the default storage parameters for all objects created in the tablespace. For information on storage parameters, see the STORAGE clause on page 4-523.
ONLINE	makes the tablespace available immediately after creation to users who have been granted access to the tablespace.
OFFLINE	makes the tablespace unavailable immediately after creation. If you omit both the ONLINE and OFFLINE options, Oracle creates the tablespace online by default. The data dictionary view DBA_TABLESPACES indicates whether each tablespace is online or offline.
PERMANENT	specifies that the tablespace will be used to hold permanent objects. This is the default.
TEMPORARY	specifies that the tablespace will be used only to hold temporary objects—for example, segments used by implicit sorts to handle ORDER BY clauses.

About Tablespaces

A *tablespace* is an allocation of space in the database that can contain any of the following segments:

- data segments
- index segments
- rollback segments
- temporary segments

All databases have at least one tablespace, SYSTEM, which Oracle creates automatically when you create the database.

When you create a tablespace, it is initially a read-write tablespace. After creating the tablespace, you can subsequently use the ALTER TABLESPACE command to take it offline or online, add datafiles to it, or make it a read-only tablespace.

Many schema objects have associated segments that occupy space in the database. These objects are located in tablespaces. The user creating such an object can optionally specify the tablespace to contain the object. The owner of the schema containing the object must have space quota on the object's tablespace. You can assign space quota on a tablespace to a user with the QUOTA clause of the CREATE USER or ALTER USER commands.

WARNING: For operating systems that support raw devices, be aware that the STORAGE clause REUSE keyword has no meaning when specifying a raw device as a datafile in a CREATE TABLESPACE command; such a command will always succeed even if REUSE is **not** specified.

Example I. This command creates a tablespace named TABSPACE_2 with one data file:

```
CREATE TABLESPACE tabspace_2
   DATAFILE 'diska:tabspace_file2.dat' SIZE 20M
   DEFAULT STORAGE (INITIAL 10K NEXT 50K
                    MINEXTENTS 1 MAXEXTENTS 999
                    PCTINCREASE 10)
   ONLINE;
```

Example II. This command creates a tablespace named TABSPACE_3 with one data file; when more space is required, 50 kilobyte extents will be added up to a maximum size of 10 megabytes:

```
CREATE TABLESPACE tabspace_5
   DATAFILE 'diskb:tabspace_file3.dat' SIZE 500K REUSE
   AUTOEXTEND ON NEXT 500K MAXSIZE 10M;
```

Example III. This command creates tablespace TABSPACE_5 with one data file and allocates every extent as a multiple of 64K:

```
CREATE TABLESPACE tabspace_3
   DATAFILE 'tabspace_file5.dbf' SIZE 2M
   MINIMUM EXTENT 64K
   DEFAULT STORAGE (INITIAL 128K NEXT 128K)
   LOGGING;
```

Related Topics

ALTER TABLESPACE on page 4-133
DROP TABLESPACE on page 4-407
"Filespec" on page 4-431

CREATE TRIGGER

Purpose

To create and enable a database trigger. A *database trigger* is a stored PL/SQL block associated with a table. Oracle automatically executes a trigger when a specified SQL statement is issued against the table. See also "Using Triggers" on page 4-336.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

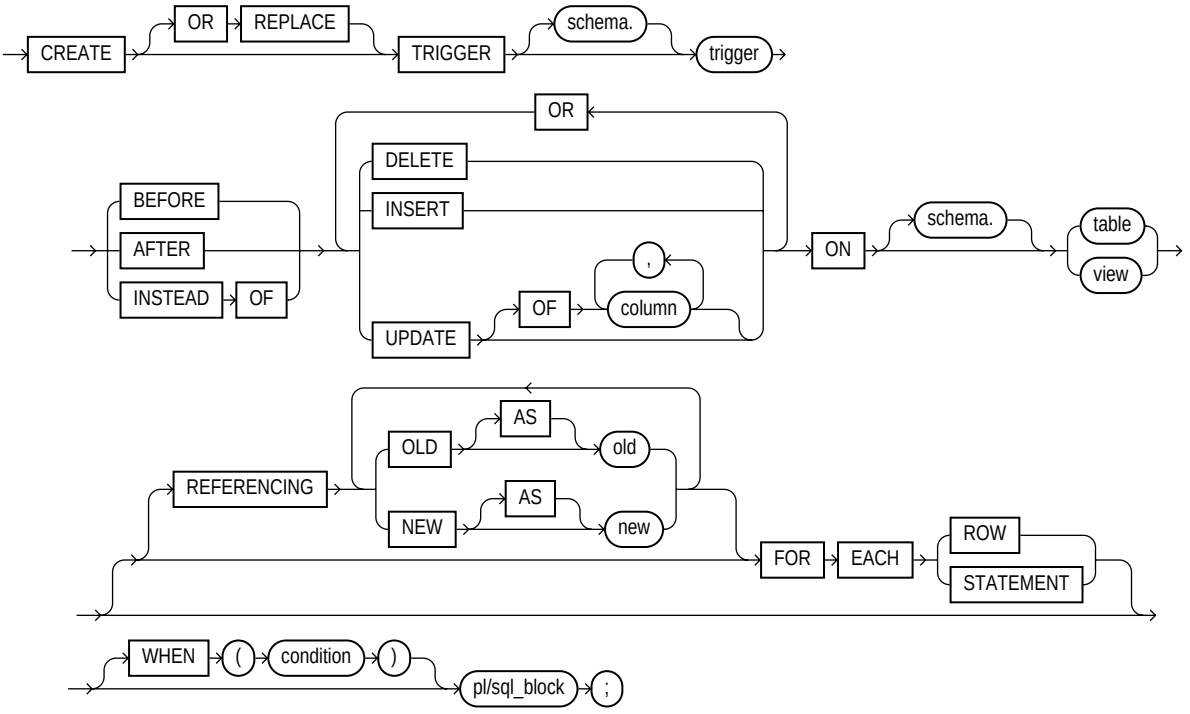
Before a trigger can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To issue this statement, you must have one of the following system privileges:

CREATE TRIGGER	lets you create a trigger in your own schema on a table in your own schema.
CREATE ANY TRIGGER	lets you create a trigger in any user's schema on a table in any schema.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the schema to contain 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



Keywords and Parameters

OR REPLACE — re-creates the trigger if it already exists. Use this option to change the definition of an existing trigger without first dropping it.

schema — is the schema to contain the trigger. If you omit schema, Oracle creates the trigger in your own schema.

trigger — is the name of the trigger to be created. See also“Conditional Predicates” on page 4-337, “Parts of a Trigger” on page 4-338, and “Types of Triggers” on page 4-339.

BEFORE — causes Oracle to fire the trigger before executing the triggering statement. For row triggers, this is a separate firing before each affected row is changed.

You cannot specify a BEFORE trigger on a view or an object view.

AFTER	causes Oracle to fire the trigger after executing the triggering statement. For row triggers, this is a separate firing after each affected row is changed. See also "Snapshot Log Triggers" on page 4-340. You cannot specify an AFTER trigger on a view or an object view.
INSTEAD OF	causes Oracle to fire the trigger instead of executing the triggering statement. By default, INSTEAD OF triggers are activated for each row. See also "INSTEAD OF Triggers" on page 4-342. INSTEAD OF is a valid option only for views. You cannot specify an INSTEAD OF trigger on a table.
DELETE	causes Oracle to fire the trigger whenever a DELETE statement removes a row from the table.
INSERT	causes Oracle to fire the trigger whenever an INSERT statement adds a row to table.
UPDATE	causes Oracle to fire the trigger whenever an UPDATE statement changes a value in one of the columns specified in the OF clause. If you omit the OF clause, Oracle fires the trigger whenever an UPDATE statement changes a value in any column of the table. You cannot specify an OF clause with an INSTEAD OF trigger. Oracle fires INSTEAD OF triggers whenever an UPDATE changes a value in any column of the view. You cannot specify nested table or LOB columns in the OF clause. See also "User-Defined Types, LOB, and REF Columns" on page 4-343.
ON	specifies the *schema* and *table* or *view* name of the of one of the following on which the trigger is to be created: ■ table ■ OBJ object table ■ view ■ OBJ object view If you omit *schema*, Oracle assumes the table is in your own schema. You can create triggers on index-organized tables. You cannot create a trigger on a table in the schema SYS. See also "User-Defined Types, LOB, and REF Columns" on page 4-343.
table	is the name of a table or an object table.
view	is the name of a view or an object view.
REFERENCING	specifies correlation names. You can use correlation names in the PL/SQL block and WHEN clause of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, use this clause to specify different correlation names to avoid confusion between the table name and the correlation name.

	OBJ If the trigger is defined on an object table or view, OLD and NEW refer to object instances.
FOR EACH ROW	designates the trigger to be a row trigger. Oracle fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the WHEN clause. Except for INSTEAD OF triggers, if you omit this clause, the trigger is a statement trigger. Oracle fires a statement trigger only once when the triggering statement is issued if the optional trigger constraint is met. INSTEAD OF trigger statements are implicitly activated for each row.
WHEN (*condition*)	specifies the trigger restriction—a SQL condition that must be satisfied for Oracle to fire the trigger. See the syntax description of condition in "Conditions" on page 3-90. This condition must contain correlation names and cannot contain a query. You can specify a trigger restriction only for a row trigger. Oracle evaluates this condition for each row affected by the triggering statement. You cannot specify trigger restrictions for INSTEAD OF trigger statements. OBJ You can reference object columns or their attributes, VARRAY, nested table, or LOB columns. You cannot invoke PL/SQL functions or methods in the trigger restriction.
pl/sql_block	is the PL/SQL block that Oracle executes to fire the trigger. For information on PL/SQL, including how to write PL/SQL blocks, see *PL/SQL User's Guide and Reference*. **Note:** The PL/SQL block of a trigger cannot contain transaction control SQL statements (COMMIT, ROLLBACK, SAVEPOINT, and SET CONSTRAINT).

Using Triggers

Oracle automatically *fires*, or executes, a trigger when a triggering statement is issued. You can use triggers for the following purposes:

- to provide sophisticated auditing and transparent event logging
- to automatically generate derived column values
- to enforce complex security authorizations and business constraints
- to maintain replicate asynchronous tables

For more information on how to design triggers for the above purposes, see *Oracle8 Application Developer's Guide*.

An existing trigger must be in one of the following states:

- If a trigger is *enabled*, Oracle fires the trigger whenever a triggering statement is issued and the condition of the trigger restriction is met.

- If a trigger is *disabled*, Oracle does not fire the trigger when a triggering statement is issued and the condition of the trigger restriction is met.

When you create a trigger, Oracle enables it automatically. You can subsequently disable and enable a trigger with the DISABLE and ENABLE options of the ALTER TRIGGER command or the ALTER TABLE command.

For information on how to enable and disable triggers, see ALTER TRIGGER on page 4-141, ALTER TABLE on page 4-106, the ENABLE clause on page 4-417, and the DISABLE clause on page 4-380.

Before Release 7.3, Oracle parsed and compiled a trigger whenever it was fired. From Release 7.3 onward, Oracle stores a compiled version of a trigger in the data dictionary and calls this compiled version when the trigger is fired. This feature provides a significant performance improvement for applications that use many triggers.

If a trigger produces compilation errors, it is still created, but it fails on execution. This means it effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped.

To embed a CREATE TRIGGER 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.

Conditional Predicates

When you create a trigger for more than one DML operation, you can use conditional predicates within the trigger body to execute specific blocks of code, depending on the type of statement that fires the trigger. Conditional predicates are evaluated as follows:

INSERTING	returns true if the trigger fires for an INSERT statement.
DELETING	returns true if the trigger fires for a DELETE statement.
UPDATING	returns true if the trigger fires for an UPDATE statement.
UPDATING (*column_name*)	returns true if the trigger fires for an UPDATE statement and *column_name* is updated. **Note:** You cannot specify an object attribute as *column_name*.

For more information about creating and using conditional predicates in trigger bodies, see *Oracle8 Application Developer's Guide.*

Example. The following example uses conditional predicates to provide information about which DML statement fires trigger AUDIT_TRIGGER:

```
CREATE TRIGGER audit_trigger BEFORE INSERT OR DELETE OR UPDATE
  ON classified_table FOR EACH ROW
  BEGIN
    IF INSERTING THEN
       INSERT INTO audit_table
          VALUES (USER || ' is inserting' ||
                    ' new key: ' || :new.key);
    ELSIF DELETING THEN
       INSERT INTO audit_table
          VALUES (USER || ' is deleting' ||
                    ' old key: ' || :old.key);
    ELSIF UPDATING('FORMULA') THEN
       INSERT INTO audit_table
          VALUES (USER || ' is updating' ||
                    ' old formula: ' || :old.formula ||
                    ' new formula: ' || :new.formula);
    ELSIF UPDATING THEN
       INSERT INTO audit_table
          VALUES (USER || ' is updating' ||
                    ' old key: ' || :old.key ||
                    ' new key: ' || :new.key);
    END IF;
  END;
```

Parts of a Trigger

The syntax of the CREATE TRIGGER statement includes the following parts of the trigger:

Triggering statement

The definition of the triggering statement specifies what SQL statements cause Oracle to fire the trigger.

DELETE INSERT UPDATE	You must specify at least one of these commands that causes Oracle to fire the trigger. You can specify as many as three.
ON	You must also specify the table with which the trigger is associated. The triggering statement is one that modifies this table. You can define a trigger on an index-organized table.

Trigger restriction

The trigger restriction specifies an additional condition that must be satisfied for a row trigger to be fired. You specify this condition with the WHEN clause. This condition must be a SQL condition, rather than a PL/SQL condition.

Trigger action

The trigger action specifies the PL/SQL block Oracle executes to fire the trigger.

Oracle evaluates the condition of the trigger restriction whenever a triggering statement is issued. If this condition is satisfied, then Oracle fires the trigger using the trigger action.

Types of Triggers

You can create different types of triggers. The type of a trigger determines:

- when Oracle fires the trigger in relation to executing the triggering statement
- how many times Oracle fires the trigger

The type of a trigger depends on the BEFORE, AFTER, and FOR EACH ROW options of the CREATE TRIGGER command. Using all combinations of these options for the above parts, you can create four types of triggers. Table 4–9 describes each type of trigger, its properties, and the options used to create it.

Table 4–9 Types of Triggers

	FOR EACH option	
	STATEMENT (or omitted)	**ROW**
BEFORE Option	**BEFORE statement trigger:** Oracle fires the trigger once before executing the triggering statement.	**BEFORE row trigger:** Oracle fires the trigger before modifying each row affected by the triggering statement.
AFTER Option	**AFTER statement trigger:** Oracle fires the trigger once after executing the triggering statement.	**AFTER row trigger:** Oracle fires the trigger after modifying each row affected by the triggering statement.

For a single table, you can create each type of trigger for each of the following commands:

- DELETE
- INSERT

- UPDATE

You can also create triggers that fire for more than one command.

If you create multiple triggers of the same type that fire for the same command on the same table, the order in which Oracle fires these triggers is indeterminate. If your application requires that one trigger be fired before another of the same type for the same command, combine these triggers into a single trigger whose trigger action performs the trigger actions of the original triggers in the appropriate order.

Snapshot Log Triggers

When you create a snapshot log for a table, Oracle implicitly creates an AFTER ROW trigger on the table. This trigger inserts a row into the snapshot log whenever an INSERT, UPDATE, or DELETE statement modifies the table's data. You cannot control the order in which multiple row triggers fire; therefore, you should not write triggers intended to affect the content of the snapshot. For more information on snapshot logs, see CREATE SNAPSHOT LOG on page 4-297.

Example I. This example creates a BEFORE statement trigger named EMP_PERMIT_CHANGES in the schema SCOTT. This trigger ensures that changes to employee records are made only during business hours on working days:

```
CREATE TRIGGER scott.emp_permit_changes
    BEFORE
    DELETE OR INSERT OR UPDATE
    ON scott.emp
    DECLARE
        dummy  INTEGER;
    BEGIN
        /* If today is a Saturday or Sunday,
           then return an error.*/
        IF (TO_CHAR(SYSDATE, 'DY') = 'SAT' OR
         TO_CHAR(SYSDATE, 'DY') = 'SUN')
         THEN raise_application_error( -20501,
       'May not change employee table during the weekend');
        END IF;
       /* Compare today's date with the dates of all
          company holidays. If today is a company holiday,
           then return an error.*/
        SELECT COUNT(*)
         INTO dummy
          FROM company_holidays
          WHERE day = TRUNC(SYSDATE);
        IF dummy > 0
```

```
      THEN raise_application_error( -20501,
       'May not change employee table during a holiday');
     END IF;
     /*If the current time is before 8:00AM or after
       6:00PM, then return an error.
     */
     IF (TO_CHAR(SYSDATE, 'HH24') < 8 OR
       TO_CHAR(SYSDATE, 'HH24') >= 18)
       THEN raise_application_error( -20502,
   'May only change employee table during working hours');
     END IF;
  END;
```

Oracle fires this trigger whenever a DELETE, INSERT, or UPDATE statement affects the EMP table in the schema SCOTT. The trigger EMP_PERMIT_CHANGES is a BEFORE statement trigger, so Oracle fires it once before executing the triggering statement.

The trigger performs the following operations:

1. If the current day is a Saturday or Sunday, the trigger raises an application error with a message that the employee table cannot be changed during weekends.
2. The trigger compares the current date with the dates listed in the table of company holidays.
3. If the current date is a company holiday, the trigger raises an application error with a message that the employee table cannot be changed during holidays.
4. If the current time is not between 8:00AM and 6:00PM, the trigger raises an application error with a message that the employee table can be changed only during business hours.

Example II. This example creates a BEFORE row trigger named SALARY_CHECK in the schema SCOTT. Whenever a new employee is added to the employee table or an existing employee's salary or job is changed, this trigger guarantees that the employee's salary falls within the established salary range for the employee's job:

```
CREATE TRIGGER scott.salary_check
    BEFORE
    INSERT OR UPDATE OF sal, job ON scott.emp
    FOR EACH ROW
    WHEN (new.job <> 'PRESIDENT')
    DECLARE
       minsal NUMBER;
```

```
   maxsal NUMBER;
BEGIN
    /* Get the minimum and maximum salaries for the
      employee's job from the SAL_GUIDE table.  */
    SELECT minsal, maxsal
     INTO minsal, maxsal
     FROM sal_guide
     WHERE job = :new.job;
     /* If the employee's salary is below the minimum or  */
     /* above the maximum for the job, then generate an   */
     /* error.*/
    IF (:new.sal < minsal OR :new.sal > maxsal)
    THEN raise_application_error( -20601,
      'Salary ' || :new.sal || ' out of range for job '
      || :new.job || ' for employee ' || :new.ename );
    END IF;
END;
```

Oracle fires this trigger whenever one of the following statements is issued:

- an INSERT statement that adds rows to the EMP table
- an UPDATE statement that changes values of the SAL or JOB columns of the EMP table

SALARY_CHECK is a BEFORE row trigger, so Oracle fires it before changing each row that is updated by the UPDATE statement or before adding each row that is inserted by the INSERT statement.

SALARY_CHECK has a trigger restriction that prevents it from checking the salary of the company president. For each new or modified employee row that meets this condition, the trigger performs the following steps:

1. The trigger queries the salary guide table for the minimum and maximum salaries for the employee's job.
2. The trigger compares the employee's salary with these minimum and maximum values.
3. If the employee's salary does not fall within the acceptable range, the trigger raises an application error with a message that the employee's salary is not within the established range for the employee's job.

INSTEAD OF Triggers

Use INSTEAD OF triggers to perform DELETE, UPDATE, or INSERT operations on views, which are not inherently modifiable. "The View Query" on page 4-366 for a

list of constructs that prevent inserts, updates, or deletes on a view. In the following example, customer data is stored in two tables. The object view ALL_CUSTOMERS is created as a UNION of the two tables, CUSTOMERS_SJ and CUSTOMERS_PA. An INSTEAD OF trigger is used to insert values:

```
CREATE TABLE customers_sj
  ( cust    NUMBER(6),
    address VARCHAR2(50),
    credit   NUMBER(9,2)  );

CREATE TABLE customers_pa
  ( cust    NUMBER(6),
    address VARCHAR2(50),
    credit   NUMBER(9,2) );

CREATE TYPE customer_t AS OBJECT
  ( cust    NUMBER(6),
    address   VARCHAR2(50),
    credit    NUMBER(9,2),
    location   VARCHAR2(20)  );

CREATE VIEW all_customers (cust)
AS SELECT customer_t (cust, address, credit, 'SAN_JOSE')
FROM   customers_sj
UNION ALL
SELECT customer_t(cust, address, credit, 'PALO_ALTO')
FROM   customers_pa;

CREATE TRIGGER instrig INSTEAD OF INSERT ON all_customers
   FOR EACH ROW
      BEGIN
        IF (:new.location = 'SAN_JOSE') THEN
           INSERT INTO customers_sj
            VALUES (:new.cust, :new.address, :new.credit);
        ELSE
           INSERT INTO customers_pa
            VALUES (:new.cust, :new.address, :new.credit);
        END IF;
       END;
```

OBJ User-Defined Types, LOB, and REF Columns

You can reference and use object, VARRAY, nested table, LOB, and REF columns in the trigger action inside the PL/SQL block, but you cannot modify their values within the trigger action. For an UPDATE trigger, object type, VARRAY type, and

REF type, you can specify columns in the OF clause to indicate that the trigger should be fired whenever an UPDATE statement changes a value in one of the columns.

When defining INSTEAD OF TRIGGERS for LOB columns, you can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW values. When defining any other triggers for LOB columns, you can read the :OLD value but not the :NEW value; you cannot write either the :OLD or the :NEW value.

Using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns does not cause Oracle to fire triggers defined on the table containing the columns or the attributes. Likewise, performing DML operations directly on nested table columns does not cause Oracle to fire triggers defined on the table containing the nested table column.

Related Topics

CREATE TRIGGER on page 4-333
DROP TRIGGER on page 4-409
ENABLE clause on page 4-417
DISABLE clause on page 4-380
CREATE VIEW on page 4-363
ALTER VIEW on page 4-154

OBJ CREATE TYPE

Purpose

To create an object type, named varying array (VARRAY), nested table type, or an incomplete object type.

> **Note:** This command is available only if the Oracle objects option is installed on your database server.

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. However, it can be referenced by other types, and so can be used to define types that refer to each other. See also "Incomplete Object Types" on page 4-351.

For more information about objects, incomplete types, VARRAYs, and nested tables see the *PL/SQL User's Guide and Reference*, *Oracle8 Application Developer's Guide*, and *Oracle8 Concepts*.

Prerequisites

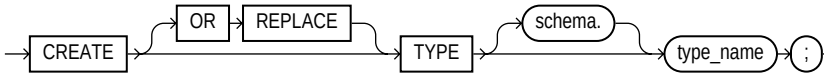
To create a type in your own schema, you must have the CREATE TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. You can acquire these privileges explicitly or be granted them through a role.

The owner of the type must either 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.

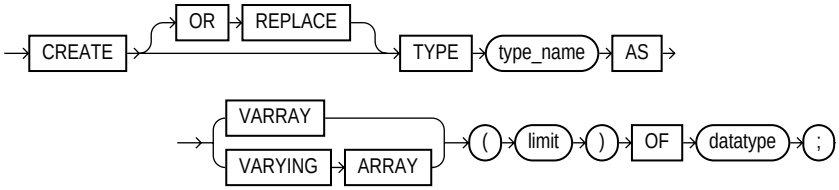
If the type owner intends to grant other users access to the type, the owner must be granted the EXECUTE object privilege to the referenced types with the GRANT OPTION, or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

Syntax

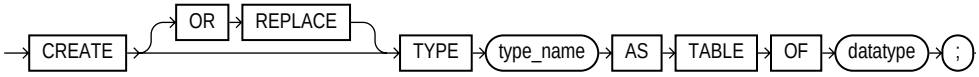
create_incomplete_type::=



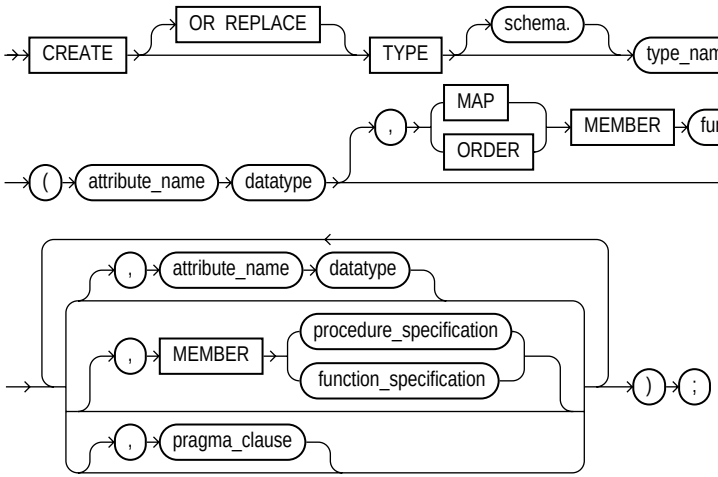
create_varray_type::=



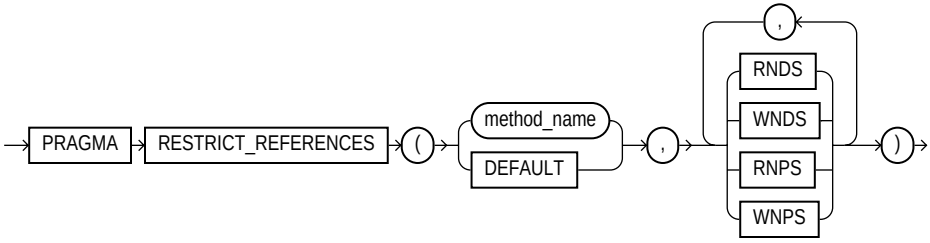
create_nested_table_type::=



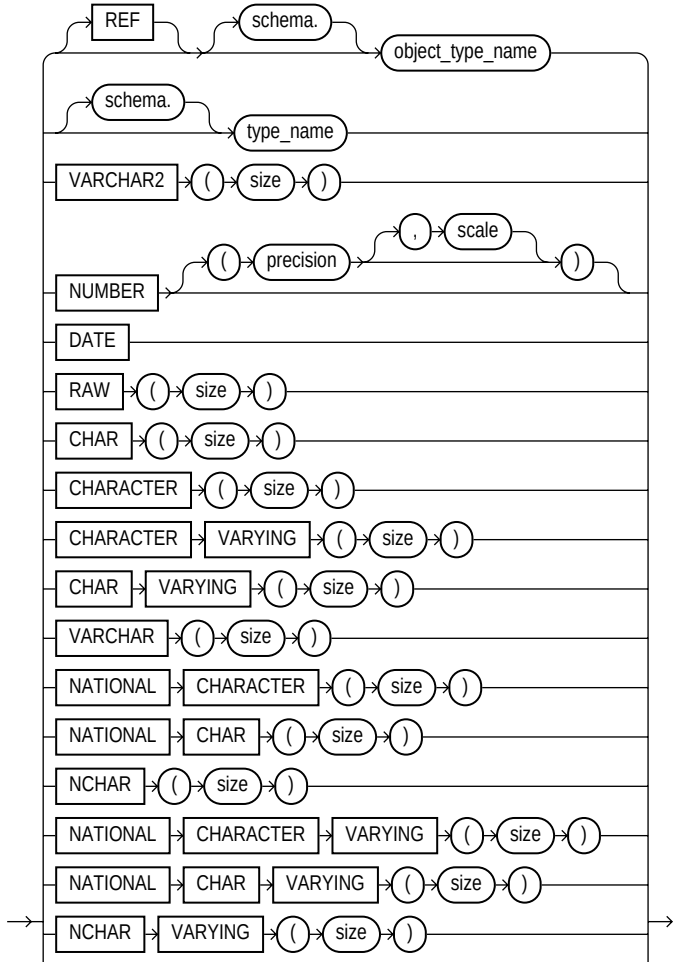
create_object_type::=

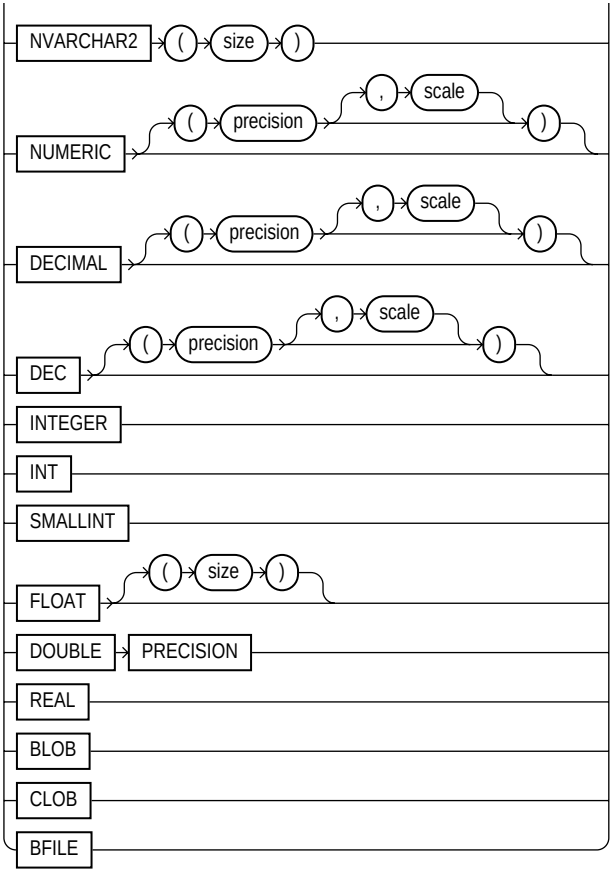


pragma_clause::=



datatype::=





Keywords and Parameters

OR REPLACE	re-creates the type if it already exists. Use this option 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.
schema	is the schema to contain the type. If you omit *schema*, Oracle creates the type in your current schema.
type_name	is the name of an object type, a nested table type, or a VARRAY type.

AS OBJECT	creates the type as a user-defined object type. The variables that form the data structure are called *attributes*. The member subprograms that define the object's behavior are called *methods*. AS OBJECT is required when creating an object type. See also "Constructors" on page 4-352.
AS TABLE OF	creates a named nested table of type *datatype*. When *datatype* is an object type, the nested table type describes a table whose columns match the name and attributes of the object type. When *datatype* is a scalar type, then the nested table type describes a table with a single, scalar type column called "column_value". Note that a collection type cannot contain any other collection type, either directly or indirectly.
AS VARRAY(*limit*)	creates the type as an ordered set of elements, each of which has the same datatype. You must specify a name and a maximum *limit* of zero or more. The array limit must be an integer literal. Only variable-length arrays are supported. Oracle does not support anonymous VARRAYs. The type name for the objects contained in the VARRAY must be one of the following: ■ a scalar datatype (see description below). The available datatypes are listed in the syntax for this command. ■ a REF, or ■ an object type, including an object with VARRAY attributes. The type name for the objects contained in the VARRAY cannot be ■ an object type with a nested table attribute, ■ a VARRAY type, or ■ a TABLE type. Note that a collection type cannot contain any other collection type, either directly or indirectly.
OF *datatype*	is the name of any Oracle built-in datatype or library type. ROWID, LONG, and LONG RAW are not valid datatypes. For a list of possible datatypes, see the syntax definition for CREATE TYPE on page 4-345.
REF *object_type_name*	associates an instance of a source type with an instance of the target object. A REF logically identifies and locates the target object. The target object must have an object identifier.
attribute_name	is an object attribute name. Attributes are data items with a name and a type specifier that form the structure of the object.

MEMBER	specifies a function or procedure subprogram associated with the object type which is referenced as an attribute. For information about overloading subprogram names within a package, see the *PL/SQL User's Guide and Reference*. You must specify a corresponding method body in the object type body for each procedure or function specification. See CREATE TYPE BODY on page 4-353.
procedure_specification	is the specification of a procedure subprogram.
function_specification	is the specification of a function subprogram.
MAP MEMBER *function_specification*	specifies a member function (map method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined *scalar* type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons. A *scalar* value is always manipulated as a single unit. Scalars are mapped directly to the underlying hardware. An integer, for example, occupies 4 or 8 contiguous bytes of storage, in memory or on disk. An object specification can contain only one map method, which must be a function. The result type must be a predefined SQL scalar type, and the map function can have no arguments other than the implicit SELF argument.
ORDER MEMBER *function_specification*	specifies a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument. When instances of the same object type definition are compared in an ORDER BY clause, the order method *function_specification* is invoked. An object specification can contain only one ORDER method, which must be a function having the return type INTEGER.

You can define either a MAP method or an ORDER method in a type specification, but not both. If you declare either method, you can compare object instances in SQL.

If neither a MAP nor an ORDER method is specified, only comparisons for equality or inequality can be performed; therefore and thus object instances cannot be ordered. Note that instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types. For more information about object value comparisons, "Object Values" on page 2-27 and *Oracle8 Application Developer's Guide*.

pragma_clause:

PRAGMA RESTRICT_ REFERENCES — is a compiler directive that denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects. For more information, see the *PL/SQL User's Guide and Reference*.

method_name	is the name of the MEMBER function or procedure to which the pragma is being applied.
WNDS	specifies the constraint *writes no database state* (does not modify database tables).
WNPS	specifies the constraint *writes no package state* (does not modify packaged variables).
RNDS	specifies the constraint *reads no database state* (does not query database tables).
RNPS	specifies the constraint *reads no package state* (does not reference packages variables).

Incomplete Object Types

You must fully specify an incomplete object type before you can use it to create a table or an object column or a column of a nested table type.

You cannot create nested VARRAY or nested table types. That is, VARRAY and nested table types cannot contain any elements that are VARRAYs or nested tables. You cannot create VARRAY types of LOB datatypes.

Example I. The following example creates object type PERSON_T with LOB attributes:

```
CREATE TYPE person_t AS OBJECT
  (name CHAR(20),
   resume CLOB,
   picture BLOB);
```

Example II. The following statement creates MEMBERS_TYPE as a VARRAY type with 100 elements:

```
CREATE TYPE members_type AS VARRAY(100) OF CHAR(5);
```

Example III. The following example creates a named table type PROJECT_TABLE of object type PROJECT_T:

```
CREATE TYPE project_t AS OBJECT
  (pno CHAR(5),
```

```
   pname CHAR(20),
   budgets DEC(7,2));

CREATE TYPE project_table AS TABLE OF project_t;
```

Example IV. The following example invokes method constructor COL.GETBAR():

```
CREATE TYPE foo AS OBJECT (a1 NUMBER,
                  MEMBER FUNCTION getbar RETURN NUMBER,
                  pragma RESTRICT_REFERENCES(getbar, WNDS, WNPS));
CREATE TABLE footab(col foo);

SELECT col.getbar() FROM footab;
```

Constructors

Oracle implicitly defines a constructor method for each user-defined type that you create. A *constructor* is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the same as the name of the user-defined type.

The parameters of the object type constructor method are the data attributes of the object type; they occur in the same order as the attribute definition order for the object type. The parameters of a nested table or VARRAY constructor are the elements of the nested table or the VARRAY.

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

Example. This example invokes the system-defined constructor to construct the FOO_T object and insert it into the FOO_TAB table:

```
CREATE TYPE foo_t AS OBJECT (a1 NUMBER, a2 NUMBER);
CREATE TABLE foo_tab (b1 NUMBER, b2 foo_t);
INSERT INTO foo_tab VALUES (1, foo_t(2,3));
```

For more information about constructors, see *Oracle8 Application Developer's Guide* and *PL/SQL User's Guide and Reference*.

Related Topics

ALTER TYPE on page 4-144
CREATE TYPE BODY on page 4-353
Oracle8 Application Developer's Guide
PL/SQL User's Guide and Reference

OBJ CREATE TYPE BODY

Purpose

To define or implement the member methods defined in the object type specification. See also “TYPE and TYPE BODY” on page 4-355.

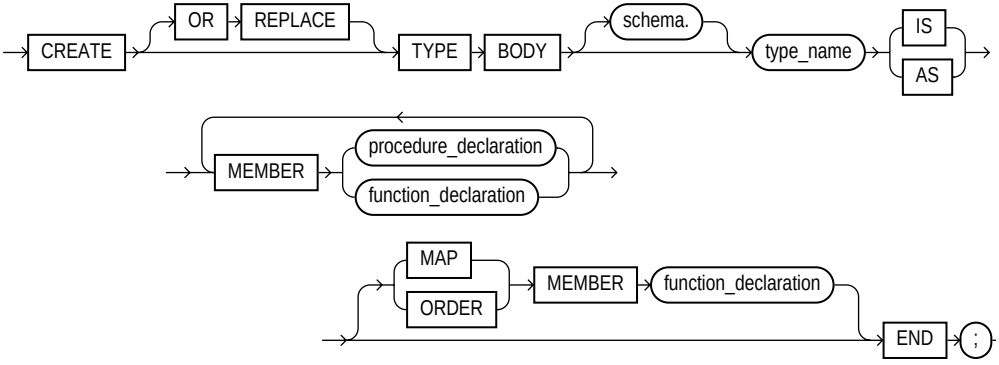
> **Note:** This command is available only if the Oracle objects option is installed on your database server.

Prerequisites

Every member declaration in the CREATE TYPE specification for object types must have a corresponding construct in the CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have CREATE TYPE or CREATE ANY TYPE system privilege. To create an object type in another user’s schema, you must have CREATE ANY TYPE system privileges. To replace an object type in another user’s schema, you must have DROP ANY TYPE system privileges.

Syntax



Keywords and Parameters

OR REPLACE	re-creates the type body if it already exists. Use this option 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 option to add new member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE command.
schema	is the schema to contain the type body. If you omit *schema*, Oracle creates the type body in your current schema.
type_name	is the name of an object type.
MEMBER	declares or implements a method function or procedure subprogram associated with the object type specification. For information about overloading subprogram names within a package, see *PL/SQL User's Guide and Reference*. You must define a corresponding method name, optional parameter list, and (for functions) a return type in the object type specification for each procedure or function declaration. See CREATE TYPE BODY on page 4-353.
procedure_ declaration	is the declaration of a procedure subprogram. For more information about writing type bodies, see *PL/SQL User's Guide and Reference*.
function_ declaration	is the declaration of a function subprogram. For more information about writing type bodies, see *PL/SQL User's Guide and Reference*.
MAP MEMBER *function_ declaration*	declares or implements a member function (map method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and specifies an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons. An object type body can contain only one map method, which must be a function. The map function can have no arguments other than the implicit SELF argument.
ORDER MEMBER *function_ specification*	specifies a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument. When instances of the same object type definition are compared in an ORDER BY clause, Oracle invokes the order method *function_specification*. An object specification can contain only one ORDER method, which must be a function having the return type INTEGER.

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare only object instances for equality or inequality. Note that instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

TYPE and TYPE BODY

You create object types with the CREATE TYPE and the CREATE TYPE BODY commands. The CREATE TYPE command specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY command contains the code for the methods in the type.

For each method specified in an object type specification, there must be a corresponding method body in the object type body.

Example. The following object type body implements member subprograms for RATIONAL:

```
CREATE TYPE BODY rational
  IS
   MAP MEMBER FUNCTION rat_to_real RETURN REAL IS
   BEGIN
     RETURN numerator/denominator;
   END;

   MEMBER PROCEDURE normalize IS
     gcd INTEGER := integer_operations.greatest_common_divisor
                    (numerator, denominator);
   BEGIN
     numerator := numerator/gcd;
     denominator := denominator/gcd;
   END;

   MEMBER FUNCTION plus(x rational) RETURN rational IS
      r rational := rational_operations.make_rational
                     (numerator*x.denominator +
                      x.numerator*denominator,
                      denominator*x.denominator);
   BEGIN
     RETURN r;
   END;
  END;
```

Related Topics

CREATE TYPE on page 4-345
ALTER TYPE on page 4-144
PL/SQL User's Guide and Reference

CREATE USER

Purpose

To create a database *user*, or an account through which you can log in to the database and establish the means by which Oracle permits access by the user. You can assign the following optional properties to the user:

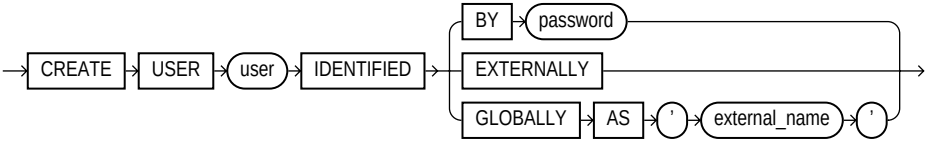
- default tablespace
- temporary tablespace
- quotas for allocating space in tablespaces
- profile containing resource limits

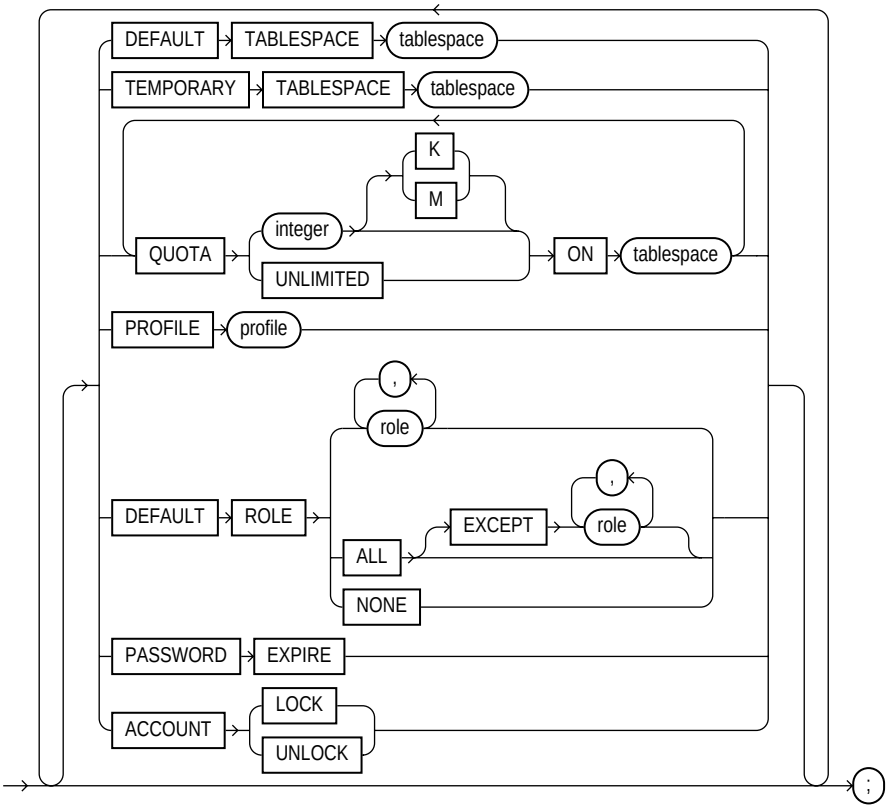
For a detailed description and explanation of how to use password management and protection, see *Oracle8 Administrator's Guide*

Prerequisites

You must have CREATE USER system privilege.

Syntax





Keywords and Parameters

user is 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 "Schema Object Naming Rules" on page 2-47. Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multi-byte characters.

IDENTIFIED indicates how Oracle authenticates the user. See *Oracle8 Application Developer's Guide* and your operating system specific documentation for more information.

	BY *password*	requires the user to specify this password to log on. *Password* must follow the rules described in the section "Schema Object Naming Rules" on page 2-47 and can only contain single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.
	EXTERNALLY	indicates that a user must be authenticated by an external service (such as an operating system or a third-party service). See also "Verifying Users Through Your Operating System" on page 4-359.
	GLOBALLY AS *'external_name'*	indicates that a user must be authenticated by the Oracle Security Service. The *'external_name'* string is the X.509 name at the Oracle Security Service that identifies this user. See also "Verifying Users Through the Network" on page 4-360.
DEFAULT TABLESPACE	identifies the default tablespace for objects that the user creates. If you omit this clause, objects default to the SYSTEM tablespace.	
TEMPORARY TABLESPACE	identifies the tablespace for the user's temporary segments. If you omit this clause, temporary segments default to the SYSTEM tablespace.	
QUOTA	allows the user to allocate space in the tablespace and optionally establishes a quota of *integer* bytes. This quota is the maximum space in the tablespace the user can allocate. You can also use K or M to specify the quota in kilobytes or megabytes. See also "Establishing Tablespace Quotas for Users" on page 4-360. Note that a CREATE USER command can have multiple QUOTA clauses for multiple tablespaces.	
	UNLIMITED	allows the user to allocate space in the tablespace without bound.
PROFILE	reassigns the profile named to the user. The profile limits the amount of database resources the user can use. If you omit this clause, Oracle assigns the DEFAULT profile to the user. See also "Granting Privileges to a User" on page 4-360.	
PASSWORD EXPIRE	causes the user's *password* to expire. Change the password before attempting to log in to the database.	
ACCOUNT LOCK	locks the user's account and disables access.	
ACCOUNT UNLOCK	unlocks the user's account and enables access to the account.	

Verifying Users Through Your Operating System

Using CREATE USER ... IDENTIFIED EXTERNALLY enables a database administrator to create a database user that can only be accessed from a specific operating system account. Effectively, you are relying on the login authentication of

the operating system to ensure that a specific operating system user has access to a specific database user. Thus, the effective security of such database accounts is entirely dependent on the strength of that security mechanism. Oracle strongly recommends that you do not use IDENTIFIED EXTERNALLY with operating systems that have inherently weak login security. For more information, see *Oracle8 Administrator's Guide*.

Verifying Users Through the Network

Using CREATE USER ... IDENTIFIED GLOBALLY enables a database administrator to create a database user that can only be authorized by an external authentication service, such as Oracle Security Server (OSS), or any external authentication system. For more information about OSS, see *The Oracle Security Server Guide* and *Oracle8 Distributed Database Systems*.

Establishing Tablespace Quotas for Users

To create an object or a temporary segment, the user must allocate space in some tablespace. To allow the user to allocate space, use the QUOTA clause. A CREATE USER statement can have multiple QUOTA clauses, each for a different tablespace. Other clauses can appear only once.

Note that you yourself need not have a quota on a tablespace to establish a quota for another user on that tablespace.

Granting Privileges to a User

For a user to perform any database operation, the user's privilege domain must contain a privilege that authorizes that operation. A user's privilege domain contains all privileges granted to the user and all privileges in the privilege domains of the user's enabled roles.

Notes:

- When you create a user with the CREATE USER command, the user's privilege domain is empty.
- To log on to Oracle, a user must have CREATE SESSION system privilege. After creating a user, you should grant the user this privilege.

Example I. If you create a new user with PASSWORD EXPIRE, the user's password must be changed before attempting to log in to the database. You can create the user SIDNEY by issuing the following statement:

```
CREATE USER sidney
```

```
    IDENTIFIED BY carton
    DEFAULT TABLESPACE cases_ts
    QUOTA 10M ON cases_ts
    QUOTA 5M ON temp_ts
    QUOTA 5M ON system
    PROFILE engineer
    PASSWORD EXPIRE;
```

The user SIDNEY has the following characteristics:

- the password CARTON
- default tablespace CASES_TS, with a quota of 10 megabytes
- temporary tablespace TEMP_TS, with a quota of 5 megabytes
- access to the tablespace SYSTEM, with a quota of 5 megabytes
- limits on database resources defined by the profile ENGINEER
- an expired password, which must be changed before attempting to log in to the database

Example II. To create a user accessible only by the operating system account GEORGE, prefix GEORGE by the value of the initialization parameter OS_AUTHENT_PREFIX. For example, if this value is “OPS$”, you can create the user OPS$GEORGE with the following statement:

```
CREATE USER ops$george
    IDENTIFIED EXTERNALLY
    DEFAULT TABLESPACE accs_ts
    TEMPORARY TABLESPACE temp_ts
    QUOTA UNLIMITED ON accs_ts
    QUOTA UNLIMITED ON temp_ts;
```

The user OPS$GEORGE has the following additional characteristics:

- default tablespace ACCS_TS
- default temporary tablespace TEMP_TS
- unlimited space on the tablespaces ACCS_TS and TEMP_TS
- limits on database resources defined by the DEFAULT profile

Example III. The following example creates user CINDY as a global user:

```
CREATE USER cindy IDENTIFIED GLOBALLY AS 'CN=cindyuser'
    DEFAULT TABLESPACE legal_ts
    QUOTA 20M ON legal_ts
```

```
    PROFILE lawyer;
```

Related Topics

ALTER USER on page 4-150
CREATE PROFILE on page 4-265
CREATE TABLESPACE on page 4-328
GRANT (System Privileges and Roles) on page 4-434

CREATE VIEW

Purpose

To define a *view*, a logical table based on one or more tables or views.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

OBJ Use CREATE VIEW to create an object view or a relational view that supports LOB and object datatypes (object types, REFs, 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.

For more information about creating and using object views, see "Using Views" on page 4-366 and *Oracle8 Application Developer's Guide*. For examples of creating views, see "Examples" on page 4-370.

Prerequisites

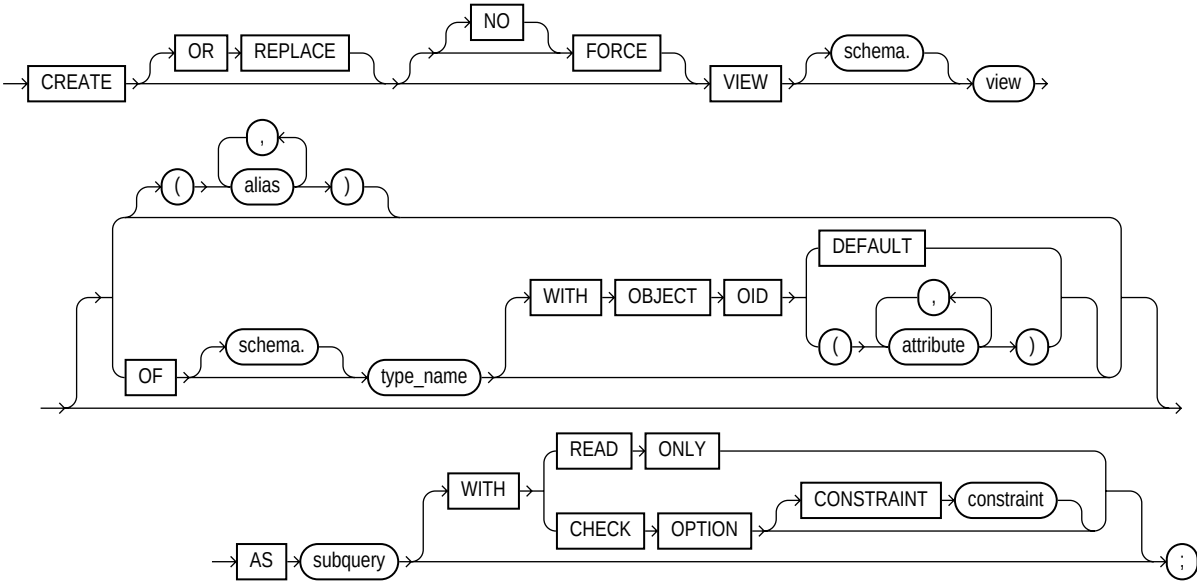
To create a view in your own schema, you must have CREATE VIEW system privilege. To create a view in another user's schema, you must have CREATE ANY VIEW system privilege.

The owner of the schema containing the view must have the privileges necessary to either select, insert, update, or delete rows from all the tables or views on which the view is based. For information on these privileges, see SELECT on page 4-489, INSERT on page 4-451, UPDATE on page 4-542, and DELETE on page 4-374. The owner must be granted these privileges directly, rather than through a role.

OBJ 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 EXECUTE ANY TYPE system privileges.
- You must EXECUTE object privileges on that object type.

Syntax



subquery: See "Subqueries" on page 4-530

Keywords and Parameters

OR REPLACE	re-creates the view if it already exists. You can use this option to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it. Note that INSTEAD OF triggers defined in the view are dropped when a view is re-created. See CREATE TRIGGER on page 4-333 for more information about the INSTEAD OF option.
FORCE	creates the view regardless of whether the view's base tables or the referenced object types exist or the owner of the schema containing the view has privileges on them. Note that these conditions must be true before any SELECT, INSERT, UPDATE, or DELETE statements can be issued against the view.
NO FORCE	creates 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.
schema	is the schema to contain the view. If you omit *schema*, Oracle creates the view in your own schema.

view	is the name of the view or the object view.
alias	specifies names for the expressions selected by the view's query. The number of aliases must match the number of expressions selected by the view. Aliases must follow the rules for naming schema objects in the section, "Referring to Schema Objects and Parts" on page 2-51. Aliases must be unique within the view. If you omit the aliases, Oracle derives them from the columns or column aliases in the view's query. For this reason, you must use aliases if the view's query contains expressions rather than only column names. OBJ You cannot specify an alias when creating an object view.
OBJ OF *type_name*	explicitly creates 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 (OID) as specified in the WITH OBJECT OID clause. See also "Object Views" on page 4-370. If you omit *schema*, *Oracle* creates the object view in your own schema. For more information about creating objects, see CREATE TYPE on page 4-345.
[OBJ WITH OBJECT OID	specifies the attributes of the object type that will be used as a key to uniquely identify each row in the object view. In most cases these attributes correspond to the primary-key columns of the base table. If the object view is defined on an object table or an object view, you can omit this clause or specify DEFAULT.
OBJ DEFAULT	specifies that the intrinsic object identifier of the underlying object table or object view will be used to uniquely identify each row.
OBJ *attribute*	is an attribute of the object type from which the object identifier for the object view is to be created.
AS *subquery*	identifies columns and rows of the table(s) that the view is based on. A view's query can be any SELECT statement without the ORDER BY or FOR UPDATE clauses. Its select list can contain up to 1000 expressions. See "The View Query" on page 4-366, "Join Views" on page 4-368, and "Subqueries" on page 4-530. OBJ For object views, the number of elements in the view subquery select list must be the same as the number of top-level attributes for the object type. The datatype of each of the selecting elements must be the same as the corresponding top-level attribute. OBJ Object types, REF *object_type*, LOBs, VARRAYs, and nested tables are valid column types.
WITH READ ONLY	specifies that no delete, inserts, or updates can be performed through the view.

WITH CHECK OPTION	specifies that inserts and updates performed through the view must result in rows that the view query can select. The CHECK OPTION cannot make this guarantee if: ■ there is a subquery in the 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.
CONSTRAINT *constraint*	assigns the name of the CHECK OPTION constraint. If you omit this identifier, 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.

Using Views

A *view* is a logical table that allows you to access data from other tables and views. A view contains no data itself. The tables upon which a view is based are called *base tables*.

Views are used for the following purposes:

- To provide an additional level of table security, by restricting access to a predetermined set of rows and/or columns of a base table.
- To hide data complexity. For example, a view may be used to act as one table when actually several tables are used to construct the result.
- To present data from another perspective. For example, views provide a means of renaming columns without actually changing the base table's definition.
- To cause Oracle to perform some operations, such as joins, on the database containing the view, rather than another database referenced in the same SQL statement. (See also "Join Views" on page 4-368.)

You can use a view anywhere you can use a table in these SQL statements:

- COMMENT
- DELETE
- INSERT
- LOCK TABLE
- UPDATE
- SELECT

The View Query

See "Subqueries" on page 4-530 for the syntax of the view's query in the description of the *subquery* syntax. Note the following caveats:

- A view's query cannot select the CURRVAL or NEXTVAL pseudocolumns.
- If a view's query selects the ROWID, ROWNUM, or LEVEL pseudocolumns, they must have aliases in the view's query.
- You can define a view with a query that uses an asterisk (*) to select all the columns of a table:

  ```
  CREATE VIEW emp_vu
    AS SELECT * FROM emp;
  ```

  Oracle translates the asterisk into a list of all the columns in the table at the time the CREATE VIEW statement is issued. If you subsequently add new columns to the table, the view will not contain these columns unless you recreate the view by issuing another CREATE VIEW statement with the OR REPLACE option. Oracle recommends that you explicitly specify all columns in the select list of a view query, rather than use the asterisk.

- You can create views that refer to remote tables and views by using database links in the view query. You should qualify any remote table or view referenced in the view query with the name of the schema containing it. Further, you should define any database links used in the view query using the CONNECT TO clause of the CREATE DATABASE LINK command.

The above caveats also apply to the query for a snapshot.

A view is *inherently updatable* if it can be inserted, updated, or deleted without using INSTEAD OF triggers and if it conforms to the restrictions listed below. However, if the view query contains any of the following constructs, it is not *inherently updatable*, and therefore you cannot perform inserts, updates, or deletes on the view except through INSTEAD OF triggers:

- set operators
- group functions
- GROUP BY, CONNECT BY, or START WITH clauses
- the DISTINCT operator
- joins (a subset of join views are updatable)

Note that if a view contains pseudocolumns or expressions, you can update the view only with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.

OBJ For more information about updating object views or relational views that support object types, see *Oracle8 Application Developer's Guide*.

Join Views

A join view is a view with a subquery containing a join. The restrictions discussed in "The View Query" on page 4-366 also apply to join views.

If at least one column in the subquery 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 ed AS
  SELECT e.empno, e.ename, d.deptno, d.loc
     FROM emp e, dept d
  WHERE e.deptno = d.deptno

View created.

SELECT column_name, updatable
  FROM user_updatable_columns
 WHERE table_name = 'ED';

COLUMN_NAME     UPD
--------------- ---
ENAME           YES
DEPTNO          NO
EMPNO           YES
LOC             NO
```

In the above example, note the unique index on the DEPTNO column of the DEPT table. You can insert, update or delete a row from the EMP base table, because all the columns in the view mapping to the emp table are marked as updatable and because the primary key of emp is included in the view. For more information on updating join views, see the *Oracle8 Application Developer's Guide*.

Note: If there were NOT NULL columns in the base EMP table that were not specified in the view subquery, then you could not insert into the table using the view.

ROWID Selection

You can select a ROWID from a join view, provided that there is one and only one key-preserved table in the join. The ROWID of that table becomes the ROWID of the view.

Updatable Join Views

A join view is a view that contains a join. Join views are updatable under the conditions discussed in this section.

A key-preserved table is a table in a join view, all of whose key columns are present as keys in the join view. This means the keys must not only be in the join view, but must still be unique and not null in the join view. This implies that a key-preserved table generally cannot be an outer-joined table. A key-preserved table could be an outer-joined table only if the outer join did not in fact generate any nulls. This, however, is a function of the data and therefore inadmissible as a basis for operations.

Therefore, you can execute the DML statements INSERT, UPDATE, and DELETE on a join view only provided that *all* of the following are true:

- The DML statement affects only one of the tables underlying the join.
- If the statement is UPDATE, then all columns updated are extracted from a key-preserved table. In addition, if the view has the CHECK OPTION, join columns and columns taken from tables that are referenced more than once in the view are shielded from UPDATE.
- If the statement is DELETE, then there is one and only one key-preserved table in the join. This table may be present more than once in the join, unless the view has the CHECK OPTION.
- If the statement is INSERT, then all columns into which values are inserted come from a key-preserved table, and the view does not have the CHECK OPTION.

Partition Views

Partition views were introduced in Release 7.3 to provide partitioning capabilities for applications requiring them. Partition views are supported in Oracle8 so that you can upgrade applications from Release 7.3 without any modification. In most cases, subsequent to migration to Oracle8 you will want to migrate partition views into partitions (see *Oracle8 Migration* and *Oracle8 Application Developer's Guide*).

With Oracle8, you can use the CREATE TABLE command to create partitioned tables easily. Partitioned tables offer the same advantages as partition views, while also addressing their shortcomings. Oracle recommends that you use partitioned tables rather than partition views in most operational environments. For more information about partitioned tables, see CREATE TABLE on page 4-306.

Examples

Example I. The following statement creates a view of the EMP table named DEPT20. The view shows the employees in Department 20 and their annual salary:

```
CREATE VIEW dept20
    AS SELECT ename, sal*12 annual_salary
        FROM emp
        WHERE deptno = 20;
```

Note that the view declaration need not define a name for the column based on the expression SAL*12, because the subquery uses a column alias (ANNUAL_SALARY) for this expression.

Example II. The following statement creates an updatable view named CLERKS of all clerks in the EMP table; only the employees' IDs, names, and department numbers are visible in this view and only these columns can be updated in rows identified as clerks:

```
CREATE VIEW clerk (id_number, person, department, position)
    AS SELECT empno, ename, deptno, job
       FROM emp
       WHERE job = 'CLERK'
    WITH CHECK OPTION CONSTRAINT wco;
```

Because of the CHECK OPTION, you cannot subsequently insert a new row into CLERK if the new employee is not a clerk.

Example III. The following statement creates a read-only view named CLERKS of all clerks in the EMP table; only the employee's IDs, names, and department numbers are visible in this view:

```
CREATE VIEW clerk (id_number, person, department, position)
     AS SELECT empno, ename, deptno, job
       FROM emp
       WHERE job = 'CLERK'
    WITH READ ONLY;
```

OBJ Object Views

An *object view* synthesizes objects based on queries of relational or object tables. The number of elements in the view subquery's select list must be the same as the number of top-level attributes of the object type. Each select element's datatype must be the same as (or convertible to) the corresponding top-level attribute.

The set of attributes in the WITH OBJECT OID clause must yield a unique key for the objects in the object view. If you try to dereference or pin a primary key REF that resolves to more than one instance in the object view, Oracle raises an error.

If a view is inherently updatable and has INSTEAD OF triggers, the triggers take preference. In other words, Oracle fires the triggers instead of performing DML on the view.

If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.

For more information about object views, refer to *Oracle8 Concepts* and the *Oracle8 Application Developer's Guide*.

Example. The following example creates object view EMP_OBJECT_VIEW of EMPLOYEE_TYPE:

```
CREATE TYPE employee_type AS OBJECT
  ( empno       NUMBER(4),
    ename       VARCHAR2(20),
    job         VARCHAR2(9),
    mgr         NUMBER(4),
    hiredate    DATE,
    sal         NUMBER(7,2),
    comm        NUMBER(7,2)  );

CREATE OR REPLACE VIEW emp_object_view OF employee_type
  WITH OBJECT OID (empno)
  AS SELECT empno, ename, job, mgr, hiredate, sal, comm
     FROM emp;
```

Related Topics

CREATE TABLE on page 4-306
CREATE SYNONYM on page 4-302
CREATE TYPE on page 4-345
DROP VIEW on page 4-416
RENAME on page 4-473
SELECT on page 4-489

DEALLOCATE UNUSED clause

Purpose

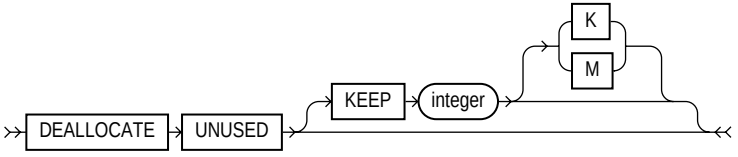
To specify the amount of unused space to deallocate from extents. See also "Deallocating Unused Space" on page 4-372.

Prerequisites

This clause can be used only in the following commands:

- ALTER CLUSTER
- ALTER TABLE
- ALTER INDEX

Syntax



Keywords and Parameters

KEEP	specifies the amount of unused space to keep.	
	integer	the number of bytes to keep. You can use K or M to specify the size in kilobytes or megabytes.

Deallocating Unused Space

For more information on the administration of schema objects, see *Oracle8 Administrator's Guide*.

Use the DEALLOCATE clause to reclaim unused space in extents in a cluster, table or index for reuse by other objects in the tablespace. Oracle credits the amount of the released space to the user quota for the tablespace in which the deallocation occurs.

Unused space is deallocated from the end of the object toward the high-water mark at the beginning of the object. 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.

The exact amount of space freed depends on the values of the INITIAL, MINEXTENTS, and NEXT parameters (are described in STORAGE clause on page 4-523).

- If you omit the KEEP option 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 use the KEEP option, 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 MINEXTENTS is adjusted to the new number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is adjusted to the new size.
- In either case, NEXT is set to the size of the last extent that was deallocated.

Example. The following command frees all unused space for reuse in table EMP, where the high-water mark is above MINEXTENTS:

```
ALTER TABLE emp
    DEALLOCATE UNUSED
```

Related Topics

ALTER TABLE on page 4-106
Oracle8 Administrator's Guide
Oracle8 Concepts

DELETE

Purpose

To remove rows from a table, a partitioned table, a view's base table, or a view's partitioned base table. See also "Using DELETE" on page 4-377.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

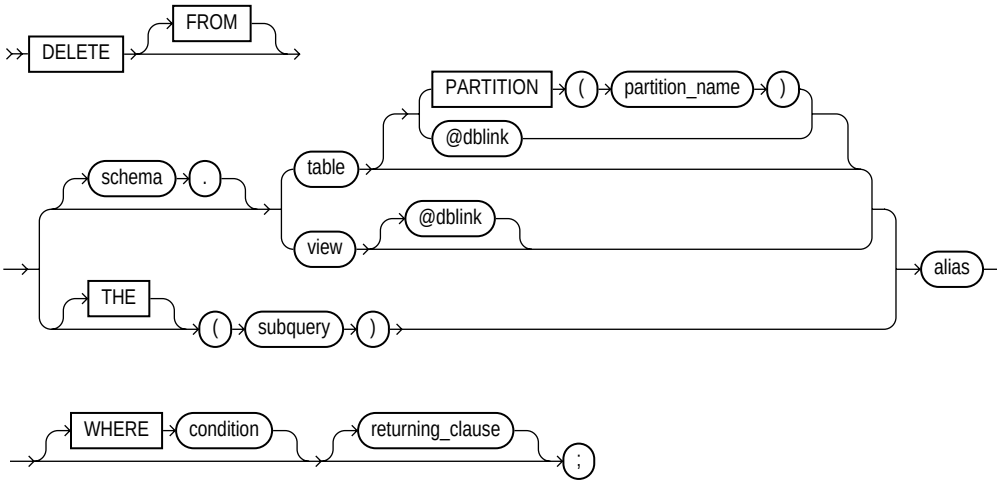
For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also allows you to delete rows from any table or any view's base table.

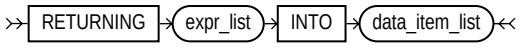
If the SQ92_SECURITY initialization parameter is set to TRUE, then you must have SELECT privilege on the table to perform a DELETE that references table columns (such as the columns in a WHERE clause).

Syntax

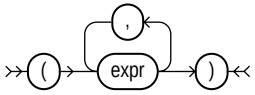


subquery: See "Subqueries" on page 4-530.

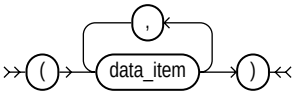
returning_clause ::=



expr_list ::=



data_item_list ::=



Keywords and Parameters

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table or *view*	is the name of a table from which the rows are to be deleted. If you specify *view*, Oracle deletes rows from the view's base table.
dblink	is the complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-54. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes that the table or view is located on the local database.
PARTITION *(partition_name)*	specifies partition-level row deletes for *table*. The *partition_name* is the name of the partition within table targeted for deletes. See also "Deleting from a Single Partition" on page 4-378.
OBJ THE	informs Oracle that the column value returned by the subquery is a nested table, not a scalar value. A subquery prefixed by THE is called a flattened subquery. "Using Flattened Subqueries" on page 4-533. *subquery* — specifies which data is selected for deletion. Oracle executes the subquery and then uses the resulting rows as a table in the FROM clause. The subquery cannot query a table that appears in the same FROM clause as the subquery. See "Subqueries" on page 4-530.
alias	is an alias assigned to the table, view or subquery. Aliases are generally used in DELETE statements with correlated queries.
WHERE *condition*	deletes only rows that satisfy the condition. The condition can reference the table and can contain a subquery. See the syntax description in "Conditions" on page 3-90. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes that the table or view is located on the local database. If you omit the WHERE clause, Oracle deletes all rows of the table or view.
returning_clause	retrieves the rows affected by the DELETE statement. You can retrieve only scalar, LOB, ROWID, and REF types. See also "The RETURNING Clause" on page 4-378.
expr	is any of the syntax descriptions in "Expressions" on page 3-78. You must specify a column expression in the RETURNING clause for each variable in the *data_item_list*.
INTO	indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item_list*.
data_item	is a PL/SQL variable or bind variable that stores the retrieved *expr* value.

You cannot use the RETURNING clause with parallel DML or with remote objects.

Using DELETE

You can use comments in a DELETE statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle8 Tuning*.

You can place a parallel hint immediately after the DELETE keyword to parallelize both the underlying scan and DELETE operations. For detailed information about parallel DML, see *Oracle8 Tuning*, *Oracle8 Parallel Server Concepts and Administration*, and *Oracle8 Concepts*.

All table and index space released by the deleted rows is retained by the table and index. You cannot delete from a view if the view's defining query contains one of the following constructs:

- set operator
- GROUP BY clause
- group function
- DISTINCT operator

Issuing a DELETE statement against a table fires any DELETE triggers defined on the table.

Example I. The following statement deletes all rows from a table named TEMP_ASSIGN.

```
DELETE FROM temp_assign;
```

Example II. The following statement deletes from the EMP table all sales staff who made less than $100 commission last month:

```
DELETE FROM emp
    WHERE JOB = 'SALESMAN'
    AND COMM < 100;
```

Example III. The following statement has the same effect as Example II:

```
DELETE FROM (select * from emp)
    WHERE JOB = 'SALESMAN'
    AND COMM < 100;
```

Example IV. The following statement deletes all rows from the bank account table owned by the user BLAKE on a database accessible by the database link DALLAS:

```
DELETE FROM blake.accounts@dallas;
```

Example V. OBJ The following example deletes rows of nested table PROJS where the department number is either 123 or 456, or the department's budget is greater than 456.78:

```
DELETE THE(SELECT projs
           FROM dept d WHERE d.dno = 123) AS p
WHERE p.pno IN (123, 456) OR p.budgets > 456.78;
```

Deleting from a Single Partition

You do not need to specify the partition name when deleting values from a partitioned table. However, in some cases specifying the partition name is more efficient than a complicated WHERE clause. To target a single partition of a partitioned table whose values you want to change, specify the PARTITION clause. This syntax is less cumbersome than using a WHERE clause in some cases.

Example. The following example removes rows from partition NOV96 of the SALES table:

```
DELETE FROM sales PARTITION (nov96)
  WHERE amount_of_sale != 0;
```

The RETURNING Clause

You can use a RETURNING clause to return values from deleted columns, and thereby eliminate the need to perform a SELECT following the DELETE statement.

- When deleting a single row, a DELETE statement with a RETURNING clause can retrieve column expressions using the deleted row, ROWID, and REFs to the deleted row and store them in PL/SQL variables or bind variables.
- When using a DELETE statement with the RETURNING clause to remove multiple rows, values from expressions, ROWID, and REFs involving the deleted rows are stored in bind arrays.

You can also use DELETE with a RETURNING clause to delete from views with single base tables.

For host binds, the datatype and size of the expression must be compatible with the bind variable.

Example. The following example returns column SAL from the deleted rows and stores the result in bind array :1:

```
DELETE FROM emp
  WHERE job = 'SALESMAN' AND COMM < 100
RETURNING sal INTO :1;
```

Related Topics

UPDATE on page 4-542

DISABLE clause

Purpose

To disable an integrity constraint or all triggers associated with a table:

- If you disable an integrity constraint, Oracle does not enforce it. However, disabled integrity constraints appear in the data dictionary along with enabled integrity constraints.
- If you disable a trigger, Oracle does not fire it if its triggering condition is satisfied.

See also “Using the DISABLE Clause” on page 4-381.

Prerequisites

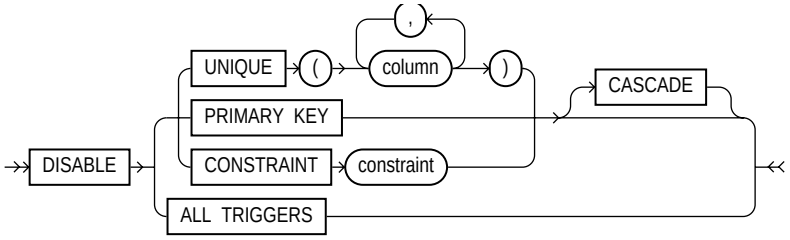
A DISABLE clause that disables an integrity constraint can appear in either a CREATE TABLE or ALTER TABLE command. To disable an integrity constraint, you must have the privileges necessary to issue one of these commands. For information on these privileges, see CREATE TABLE on page 4-306 and ALTER TABLE on page 4-106.

For an integrity constraint to appear in a DISABLE clause, either

- the integrity constraint must be defined in the containing statement, or
- the integrity constraint must already have been defined and enabled in previously issued statements.

A DISABLE clause that disables triggers can appear only in an ALTER TABLE statement. To disable triggers with a DISABLE clause, you must have the privileges necessary to issue the ALTER TABLE statement. For information on these privileges, see ALTER TABLE on page 4-106. Also, the triggers must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

Syntax



Keywords and Parameters

UNIQUE	disables the UNIQUE constraint defined on the specified column or combination of columns.
PRIMARY KEY	disables the table's PRIMARY KEY constraint.
CONSTRAINT	disables the integrity constraint with the name constraint.
CASCADE	disables 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 option.
ALL TRIGGERS	disables all triggers associated with the table. This option can appear only in a DISABLE clause in an ALTER TABLE statement, not in a CREATE TABLE statement.

Using the DISABLE Clause

Use the DISABLE clause to disable either:

- a single integrity constraint or
- all triggers associated with a table.

To disable a single trigger, use the DISABLE option of the ALTER TRIGGER command.

Disabling Integrity Constraints

You disable an integrity constraint by naming it in a DISABLE clause of either a CREATE TABLE or ALTER TABLE statement. You can define an integrity constraint with a CONSTRAINT clause and disable it with a DISABLE clause together in the same statement. You can also define an integrity constraint in one statement and subsequently disable it in another.

You can also disable an integrity constraint with the DISABLE keyword in the CONSTRAINT clause that defines the integrity constraint. For information on this keyword, see the CONSTRAINT clause on page 4-188.

How Oracle Disables Integrity Constraints

If you disable an integrity constraint, Oracle does not enforce it. If you define an integrity constraint and disable it, Oracle does not apply it to existing rows of the table, although Oracle does store it in the data dictionary along with enabled integrity constraints. Also, Oracle can execute data manipulation language (DML) statements that change table data and violate a disabled integrity constraint.

If you disable a UNIQUE or PRIMARY KEY constraint that was previously enabled, Oracle drops the index that enforces the constraint.

You can enable a disabled integrity constraint with the ENABLE clause.

Disabling Referenced Keys in Referential Integrity Constraints

To disable a UNIQUE or PRIMARY KEY constraint that identifies the referenced key of a referential integrity constraint (foreign key), you must also disable the foreign key. To do so, use the CASCADE option of the DISABLE clause.

You cannot enable a foreign key that references a unique or primary key that is disabled.

Example I. The following statement creates the DEPT table and defines a disabled PRIMARY KEY constraint:

```
CREATE TABLE dept
    (deptno  NUMBER(2)  PRIMARY KEY,
     dname    VARCHAR2(10),
     loc      VARCHAR2(9) )
     DISABLE  PRIMARY KEY;
```

Since the primary key is disabled, you can add to the table rows that violate the primary key. For example, you can add departments with null department numbers or multiple departments with the same department number.

Example II. The following statement defines and disables a CHECK constraint on the EMP table:

```
ALTER TABLE emp
    ADD (CONSTRAINT check_comp  CHECK (sal + comm <= 5000) )
    DISABLE CONSTRAINT check_comp;
```

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.

Example III. Consider a referential integrity constraint involving a foreign key on the combination of the AREACO and PHONENO columns of the PHONE_CALLS table. The foreign key references a unique key on the combination of the AREACO and PHONENO columns of the CUSTOMERS table. The following statement disables the unique key on the combination of the AREACO and PHONENO columns of the CUSTOMERS table:

```
ALTER TABLE customers
    DISABLE UNIQUE (areaco, phoneno) CASCADE;
```

The unique key in the CUSTOMERS table is referenced by the foreign key in the PHONE_CALLS table, so you must use the CASCADE option to disable the unique key. This option disables the foreign key as well.

How to Disable Triggers

You can disable all triggers associated with a table by using the ALL TRIGGERS option in a DISABLE clause of an ALTER TABLE statement. After you disable a trigger, Oracle does not fire the trigger when a triggering statement meets the condition of the trigger restriction.

Example . The following statement disables all triggers associated with the EMP table:

```
ALTER TABLE emp
    DISABLE ALL TRIGGERS;
```

Related Topics

ALTER TABLE on page 4-106
ALTER TRIGGER on page 4-141
CONSTRAINT clause on page 4-188
CREATE TABLE on page 4-306
CREATE TRIGGER on page 4-333
DISABLE clause on page 4-380

DROP clause

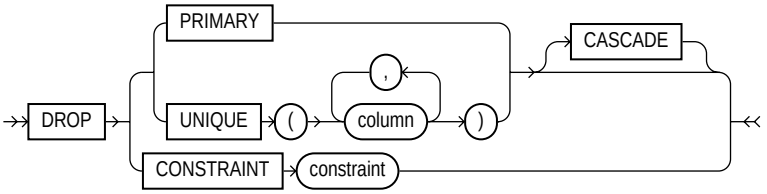
Purpose

To remove an integrity constraint from the database. See also “Removing Integrity Constraints” on page 4-384.

Prerequisites

The DROP clause can appear only in an ALTER TABLE statement. To drop an integrity constraint, you must have the privileges necessary to issue an ALTER TABLE statement. For information on these privileges, see ALTER TABLE on page 4-106.

Syntax



Keywords and Parameters

PRIMARY KEY	drops the table’s PRIMARY KEY constraint.
UNIQUE	drops the UNIQUE constraint on the specified columns.
CONSTRAINT	drops the integrity constraint named *constraint*.
CASCADE	drops all other integrity constraints that depend on the dropped integrity constraint.

Removing Integrity Constraints

You can drop an integrity constraint by naming it in a DROP clause of an ALTER TABLE statement. When you drop an integrity constraint, Oracle stops enforcing the integrity constraint and removes it from the data dictionary.

You cannot drop a unique or primary key that is part of a referential integrity constraint without also dropping the foreign key. You can drop the referenced key and the foreign key together by specifying the referenced key with the CASCADE

option in the DROP clause. If you omit CASCADE, Oracle does not drop the unique or primary key constraint if any foreign key references it.

Example I. The following statement drops the primary key of the DEPT table:

```
ALTER TABLE dept
    DROP PRIMARY KEY CASCADE;
```

If you know that the name of the PRIMARY KEY constraint is PK_DEPT, you could also drop it with the following statement:

```
ALTER TABLE dept
    DROP CONSTRAINT pk_dept CASCADE;
```

The CASCADE option drops any foreign keys that reference the primary key.

Example II. The following statement drops the unique key on the DNAME column of the DEPT table:

```
ALTER TABLE dept
    DROP UNIQUE (dname);
```

Note that the DROP clause in this example omits the CASCADE option. Because of this omission, Oracle does not drop the unique key if any foreign key references it.

Related Topics

ALTER TABLE on page 4-106
CONSTRAINT clause on page 4-188

DROP CLUSTER

Purpose

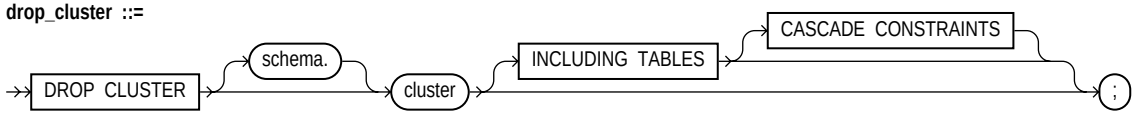
To remove a cluster from the database. See “Restrictions” on page 4-386.

Prerequisites

The cluster must be in your own schema or you must have DROP ANY CLUSTER system privilege.

Syntax

drop_cluster ::=



Keywords and Parameters

schema	is the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.
cluster	is the name of the cluster to be dropped.
INCLUDING TABLES	drops all tables that belong to the cluster.
CASCADE CONSTRAINTS	drops 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 option and such referential integrity constraints exist, Oracle returns an error message and does not drop the cluster.

Restrictions

Dropping a cluster also drops the cluster index and returns all cluster space, including data blocks for the index, to the appropriate tablespace(s).

You cannot uncluster an individual table. To create an unclustered table identical to an existing clustered table, follow the following steps:

1. Create a new table with the same structure and contents as the old one but with no CLUSTER option.

2. Drop the old table.
3. Use the RENAME command to give the new table the name of the old one.

Grants on the old clustered table do not apply to the new unclustered table and must be regranted.

Example. This command drops a cluster named GEOGRAPHY, all its tables, and any referential integrity constraints that refer to primary or unique keys in those tables:

```
DROP CLUSTER geography
    INCLUDING TABLES
        CASCADE CONSTRAINTS;
```

Related Topic

DROP TABLE on page 4-405

DROP DATABASE LINK

Purpose

To remove a database link from the database.

Prerequisites

To drop a private database link, the database link must be in your own schema. To drop a PUBLIC database link, you must have DROP PUBLIC DATABASE LINK system privilege. See also "Example" below.

Syntax

Keywords and Parameters

PUBLIC	must be specified to drop a PUBLIC database link.
dblink	specifies the database link to be dropped.

Restrictions

You cannot drop 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; therefore, Oracle 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.

Example

The following statement drops a private database link named BOSTON:

```
DROP DATABASE LINK boston;
```

Related Topics

CREATE DATABASE LINK on page 4-225

DROP DIRECTORY

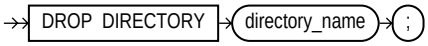
Purpose

Use DROP DIRECTORY to remove a directory object from the database. See also "Dropping a Directory" below.

Prerequisites

To drop a directory you must have DROP ANY DIRECTORY system privilege.

Syntax



Keywords and Parameters

directory_name	is the name of the directory database object to be dropped.

Dropping a Directory

Dropping a directory removes the database object, but does not delete the associated operating system directory on the server's file system.

You should not DROP a directory when files in the associated file system are being accessed by PL/SQL or OCI programs.

Example. The following statement drops the directory object BFILE_DIR:

```
DROP DIRECTORY bfile_dir;
```

Related Topics

CREATE DIRECTORY on page 4-230
GRANT (System Privileges and Roles) on page 4-434
"Large Object (LOB) Datatypes" on page 2-15

DROP FUNCTION

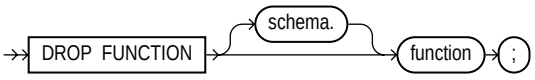
Purpose

To remove a standalone stored function from the database. See also "Dropping Functions" below.

Prerequisites

The function must be in your own schema or you must have DROP ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.
function	is the name of the function to be dropped.

Dropping Functions

When you drop a function, Oracle invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error message if you have not re-created the dropped function. For more information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8 Concepts*.

You can use this command to drop only a standalone function. To remove a function that is part of a package, use one of the following methods:

- Drop the entire package using the DROP PACKAGE command.
- Redefine the package without the function using the CREATE PACKAGE command with the OR REPLACE option.

Example. The following statement drops the function NEW_ACCT in the schema RIDDLEY:

```
DROP FUNCTION riddley.new_acct;
```

When you drop the NEW_ACCT function, Oracle invalidates all objects that depend upon NEW_ACCT.

Related Topics

CREATE FUNCTION on page 4-232

DROP INDEX

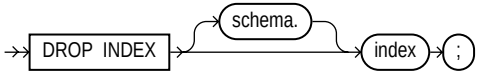
Purpose

To remove an index from the database. See also "Dropping an Index" below.

Prerequisites

The index must be in your own schema or you must have DROP ANY INDEX system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.
index	is the name of the index to be dropped.

Dropping an Index

When the index is dropped, all data blocks allocated to the index are returned to the index's tablespace.

Example. This command drops an index named MONOLITH:

```
DROP INDEX monolith;
```

Related Topics

ALTER INDEX on page 4-28

DROP LIBRARY

Purpose

To remove an external procedure library from the database.

Prerequisites

You must have the DROP LIBRARY system privilege.

Syntax

Keywords and Parameters

libname	is the name of the external procedure library being dropped.

Example

The following statement drops the EXT_PROCS library:

```
DROP LIBRARY ext_procs;
```

Related Topics

CREATE LIBRARY on page 4-248

DROP PACKAGE

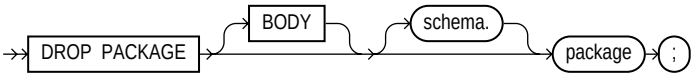
Purpose

To remove a stored package from the database. See also "Dropping a Package" below.

Prerequisites

The package must be in your own schema or you must have DROP ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

BODY	drops only the body of the package. If you omit this option, Oracle drops both the body and specification of the package.
schema	is the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.
package	is the name of the package to be dropped.

Dropping a Package

When you drop the body and specification of a package, Oracle invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped package. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8 Concepts*.

When you drop only the body of a package but not its specification, Oracle 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.

The DROP PACKAGE command drops the package and all its objects together. To remove a single object from a package, re-create the package without the object using the CREATE PACKAGE and CREATE PACKAGE BODY commands with the OR REPLACE option.

Example. The following statement drops the specification and body of the BANKING package, invalidating all objects that depend on the specification:

```
DROP PACKAGE banking;
```

Related Topics

CREATE PACKAGE on page 4-250

DROP PROCEDURE

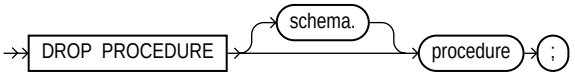
Purpose

To remove a standalone stored procedure from the database. See also "Dropping a Procedure" below.

Prerequisites

The procedure must be in your own schema or you must have DROP ANY PROCEDURE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.
procedure	is the name of the procedure to be dropped.

Dropping a Procedure

When you drop a procedure, Oracle invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error message if you have not re-created the dropped procedure. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8 Concepts*.

You can use this command only to drop a standalone procedure. To remove a procedure that is part of a package, use one of the following methods:

- Drop the entire package using the DROP PACKAGE command.
- Redefine the package without the procedure using the CREATE PACKAGE command with the OR REPLACE option.

Example. The following statement drops the procedure TRANSFER owned by the user KERNER:

```
DROP PROCEDURE kerner.transfer
```

When you drop the TRANSFER procedure, Oracle invalidates all objects that depend upon TRANSFER.

Related Topics

CREATE PROCEDURE on page 4-259

DROP PROFILE

Purpose

To remove a profile from the database. See also "Dropping a Profile" below.

Prerequisites

You must have DROP PROFILE system privilege.

Syntax

Keywords and Parameters

profile	is the name of the profile to be dropped.
CASCADE	deassigns the profile from any users to whom it is assigned. Oracle automatically assigns the DEFAULT profile to such users. You must specify this option to drop a profile that is currently assigned to users.

Dropping a Profile

You cannot drop the DEFAULT profile.

Example. The following statement drops the profile ENGINEER:

```
DROP PROFILE engineer CASCADE;
```

Oracle assigns the DEFAULT profile to any users currently assigned the ENGINEER profile.

Related Topics

CREATE PROFILE on page 4-265

DROP ROLE

Purpose

To remove a role from the database. See also "Dropping a Role" below.

Prerequisites

You must have been granted the role with the ADMIN OPTION or you must have DROP ANY ROLE system privilege.

Syntax

Keywords and Parameters

role	is the role to be dropped.

Dropping a Role

When you drop a role, Oracle revokes it from all users and roles to whom it has been granted and removes it from the database.

Example. To drop the role FLORIST, issue the following statement:

```
DROP ROLE florist;
```

Related Topics

CREATE ROLE on page 4-272
SET ROLE on page 4-516

DROP ROLLBACK SEGMENT

Purpose

To remove a rollback segment from the database. See also "Dropping Rollback Segments" below.

Prerequisites

You must have DROP ROLLBACK SEGMENT system privilege.

Syntax

Keywords and Parameters

rollback_segment	is the name the rollback segment to be dropped.

Dropping Rollback Segments

When you drop a rollback segment, all space allocated to the rollback segment returns to the tablespace.

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 option of the ALTER ROLLBACK SEGMENT command.

You cannot drop the SYSTEM rollback segment.

Example. The following statement drops the rollback segment ACCOUNTING:

```
DROP ROLLBACK SEGMENT accounting;
```

Related Topics

ALTER ROLLBACK SEGMENT on page 4-53
CREATE ROLLBACK SEGMENT on page 4-275
CREATE TABLESPACE on page 4-328

DROP SEQUENCE

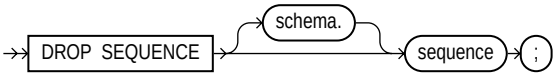
Purpose

To remove a sequence from the database. See also "Dropping Sequences" below.

Prerequisites

The sequence must be in your own schema or you must have DROP ANY SEQUENCE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the sequence. If you omit *schema*, Oracle assumes the sequence is in your own schema.
sequence	is the name of the sequence to be dropped.

Dropping Sequences

One method for restarting a sequence is to drop and re-create 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, you would:

1. Drop the sequence.
2. Create it with the same name and a START WITH value of 27.

Example. The following statement drops the sequence ESEQ owned by the user ELLY. To issue this statement, you must either be connected as user ELLY or have DROP ANY SEQUENCE system privilege:

```
DROP SEQUENCE elly.eseq;
```

Related Topics

ALTER SEQUENCE on page 4-56
CREATE SEQUENCE on page 4-281

DROP SNAPSHOT

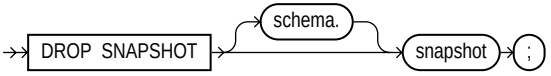
Purpose

To remove a snapshot from the database. See "Dropping Snapshots" below.

Prerequisites

The snapshot must be in your own schema or you must have DROP ANY SNAPSHOT system privilege. You must also have the privileges to drop the internal table, views, and index that Oracle uses to maintain the snapshot's data. For information on these privileges, see DROP TABLE on page 4-405, DROP VIEW on page 4-416, and DROP INDEX on page 4-392.

Syntax



Keywords and Parameters

schema	is the schema containing the snapshot. If you omit *schema*, Oracle assumes the snapshot is in your own schema.
snapshot	is the name of the snapshot to be dropped.

Dropping Snapshots

When you drop a simple snapshot that is the least recently refreshed snapshot of a master table, Oracle automatically purges from master table's snapshot log only the rows needed to refresh the dropped snapshot.

When you drop a master table, Oracle does not automatically drop snapshots based on the table. However, Oracle returns an error message when it tries to refresh a snapshot based on a master table that has been dropped.

The following statement drops the snapshot PARTS owned by the user HQ:

```
DROP SNAPSHOT hq.parts;
```

Related Topics

CREATE SNAPSHOT on page 4-286

DROP SNAPSHOT LOG

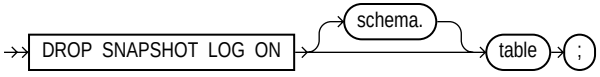
Purpose

To remove a snapshot log from the database. See also "Dropping Snapshot Logs" below.

Prerequisites

A snapshot log consists of a table and a trigger. To drop a snapshot log, you must have the privileges listed for DROP TABLE on page 4-405. You must also have the privileges to drop a trigger from the snapshot log's master table. For information on these privileges, see DROP TRIGGER on page 4-409.

Syntax



Keywords and Parameters

schema	is the schema containing the snapshot log and its master table. If you omit *schema*, Oracle assumes the snapshot log and master table are in your own schema.
table	is the name of the master table associated with the snapshot log to be dropped.

Dropping Snapshot Logs

After you drop a snapshot log, snapshots based on the snapshot log's master table can no longer be refreshed fast. They must be refreshed completely. For more information on refreshing snapshots, see CREATE SNAPSHOT on page 4-286.

The following statement drops the snapshot log on the PARTS master table:

```
DROP SNAPSHOT LOG ON parts;
```

Related Topics

CREATE SNAPSHOT LOG on page 4-297

DROP SYNONYM

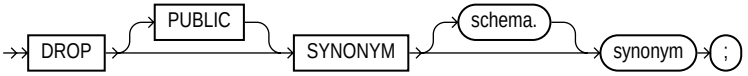
Purpose

To remove a synonym from the database. See also "Dropping Synonyms" below.

Prerequisites

If you want to drop a private synonym, either the synonym must be in your own schema or you must have DROP ANY SYNONYM system privilege. If you want to drop a PUBLIC synonym, either the synonym must be in your own schema or you must have DROP ANY PUBLIC SYNONYM system privilege.

Syntax



Keywords and Parameters

PUBLIC	must be specified to drop a public synonym. You cannot specify *schema* if you have specified PUBLIC.
schema	is the schema containing the synonym. If you omit *schema*, Oracle assumes the synonym is in your own schema.
synonym	is the name of the synonym to be dropped.

Dropping Synonyms

You can change the definition of a synonym by dropping and re-creating it.

To drop a synonym named MARKET, issue the following statement:

```
DROP SYNONYM market;
```

Related Topic

CREATE SYNONYM on page 4-302

DROP TABLE

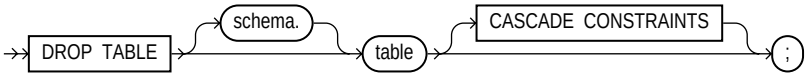
Purpose

To remove a table or an object table and all its data from the database. See also "Dropping Tables" below.

Prerequisites

The table must be in your own schema or you must have DROP ANY TABLE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the table. If you omit *schema*, Oracle assumes the table is in your own schema.
table	is the name of the table, object table, or index-organized table to be dropped.
CASCADE CONSTRAINTS	drops all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this option, and such referential integrity constraints exist, Oracle returns an error message and does not drop the table.

Dropping Tables

When you drop a table, Oracle also automatically performs the following operations:

- Oracle removes all rows from the table (as if the rows were deleted).
- Oracle drops all the table's indexes, regardless of who created them or whose schema contains them.
- If the table is not part of a cluster, Oracle returns all data blocks allocated to the table and its indexes to the tablespaces containing the table and indexes.
- If the table is a base table for views or if it is referenced in stored procedures, functions, or packages, Oracle invalidates these 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, it must contain all the columns selected by the queries originally used to define the 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 snapshots, Oracle does not drop the snapshots. Such a snapshot 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 snapshot's query.

  If you choose to re-create the table, it must contain all the columns selected by the queries originally used to define the snapshots.

- If the table has a snapshot log, Oracle drops the snapshot log.

You can drop a cluster and all of its tables using the DROP CLUSTER command with the INCLUDING TABLES clause to avoid dropping each table individually.

Example. The following statement drops the TEST_DATA table:

```
DROP TABLE test_data;
```

Related Topics

DROP CLUSTER on page 4-386
ALTER TABLE on page 4-106
CREATE INDEX on page 4-237
CREATE TABLE on page 4-306

DROP TABLESPACE

Purpose

To remove a tablespace from the database. See also "Dropping Tablespaces" below.

Prerequisites

You must have DROP TABLESPACE system privilege. You cannot drop a tablespace if it contains any rollback segments holding active transactions.

Syntax

Keywords and parameters

tablespace	is the name of the tablespace to be dropped.
INCLUDING CONTENTS	drops 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, Oracle returns error messages and does not drop the tablespace. **Note:** If the tablespace contains some, but not all, partitions of a partitioned table, DROP TABLESPACE will fail even if you specify INCLUDING CONTENTS.
CASCADE CONSTRAINTS	drops all referential integrity constraints from tables outside this clause to drop a tablespace that contains any database objects. If you omit this option and such referential integrity constraints exist, Oracle returns an error message and does not drop the tablespace.

Dropping Tablespaces

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 may want to alter 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 command.

You cannot drop the SYSTEM tablespace.

Example. The following statement drops the MFRG tablespace and all its contents:

```
DROP TABLESPACE mfrg
    INCLUDING CONTENTS
        CASCADE CONSTRAINTS;
```

Related Topics

ALTER TABLESPACE on page 4-133
CREATE DATABASE on page 4-219
CREATE TABLESPACE on page 4-328

DROP TRIGGER

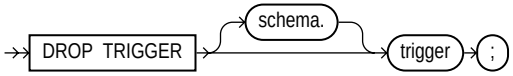
Purpose

To remove a database trigger from the database. See also "Dropping Triggers" below.

Prerequisites

The trigger must be in your own schema or you must have DROP ANY TRIGGER system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.
trigger	is the name of the trigger to be dropped.

Dropping Triggers

When you drop a database trigger, Oracle removes it from the database and does not fire it again.

The following statement drops the REORDER trigger in the schema RUTH:

```
DROP TRIGGER ruth.reorder;
```

Related Topics

CREATE TRIGGER on page 4-333

OBJ DROP TYPE

Purpose

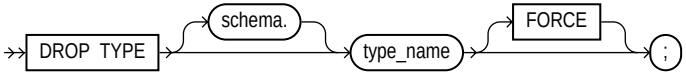
To drop the specification and body of an object, a VARRAY, or nested table type. To drop just the body of an object, use the DROP TYPE BODY on page 4-412. See also "Dropping Types" below.

> **Note:** This command is available only if the Oracle objects option is installed on your database server.

Prerequisites

The object, VARRAY, or nested table type must be in your own schema or you must have DROP ANY TYPE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.
type_name	is the name of the object, VARRAY, or nested table type to be dropped. You can drop only types with no type or table dependencies.
FORCE	forces the type to be dropped even if it has table or type dependencies.

Dropping Types

Unless you specify FORCE, you can drop only object, nested table, or VARRAY types that are standalone schema objects with no dependencies. This is the default behavior.

WARNING: Oracle does not recommend using the FORCE option to drop types with dependencies. This operation is not recoverable and could cause the data in the dependent tables to become inaccessible. For information about type dependencies, see *Oracle8 Application Developer's Guide*.

Example. The following statement removes object type PERSON_T:

```
DROP TYPE person_t;
```

Related Topics

CREATE TYPE on page 4-345

OBJ DROP TYPE BODY

Purpose

To drop the body of an object, a VARRAY, or nested table type. See also "Dropping Type Bodies" below.

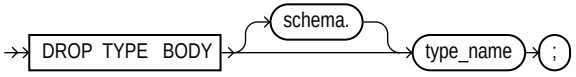
To drop the specification of an object, see DROP TYPE on page 4-410.

> **Note:** This command is available only if the Oracle objects option is installed on your database server.

Prerequisites

The object type body must be in your own schema, and you must have CREATE TYPE or CREATE ANY TYPE system privilege, or you must have DROP ANY TYPE system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.
type_name	is the name of the object type body to be dropped. You can only drop type bodies with no ty6pe or table dependencies.

Dropping Type Bodies

When you drop a type body, the object type specification still exists, and you can re-create the type body. You can still use the object type, although you cannot call the member functions.

The following statement removes object type body RATIONAL:

```
DROP TYPE BODY rational;
```

Related Topics

CREATE TYPE BODY on page 4-353

DROP USER

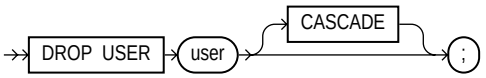
Purpose

To remove a database user and optionally remove the user's objects. See also "Dropping Users and Their Objects" below.

Prerequisites

You must have the DROP USER system privilege.

Syntax



Keywords and Parameters

user	is the user to be dropped.
CASCADE	drops all objects in the user's schema before dropping the user. You must specify this option to drop a user whose schema contains any objects.

Dropping Users and Their Objects

Oracle does not drop users whose schemas contain objects. To drop such a user, you must either

- explicitly drop the user's objects before dropping the user or
- drop the user and objects together using the CASCADE option.

If you specify the CASCADE option and drop tables in the user's schema, Oracle automatically drops any referential integrity constraints on tables in other schemas that refer to primary and unique keys on these tables. The CASCADE option also causes Oracle to invalidate, but 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 does not drop snapshots on tables or views in the dropped user's schema, but if you specify CASCADE, the snapshots can no longer be refreshed.

Oracle does not drop roles created by the user.

Example I. If BRADLEY's schema contains no objects, you can drop BRADLEY by issuing the statement:

```
DROP USER bradley;
```

Example II. If BRADLEY's schema contains objects, you must use the CASCADE option to drop BRADLEY and the objects:

```
DROP USER bradley CASCADE;
```

Related Topics

CREATE USER on page 4-357
DROP TABLE on page 4-405
CREATE TABLESPACE on page 4-328
DROP TRIGGER on page 4-409
DROP VIEW on page 4-416

DROP VIEW

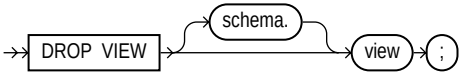
Purpose

To remove a view or an object view from the database. See also "Dropping Views" below.

Prerequisites

The view must be in your own schema or you must have DROP ANY VIEW system privilege.

Syntax



Keywords and Parameters

schema	is the schema containing the view. If you omit *schema*, Oracle assumes the view is in your own schema.
view	is the name of the view to be dropped.

Dropping Views

When you drop a view, views and synonyms that refer to the view are not dropped, but become invalid. Drop them or redefine them, or define other views in such a way that the invalid views and synonyms become valid again.

You can change the definition of a view by dropping and re-creating it.

Example. The following statement drops the VIEW_DATA view:

```
DROP VIEW view_data;
```

Related Topics

CREATE TABLE on page 4-306
CREATE VIEW on page 4-363
CREATE SYNONYM on page 4-302

ENABLE clause

Purpose

To enable an integrity constraint or all triggers associated with a table:

- If you enable a constraint, Oracle enforces it by applying it to all data in the table. All table data must satisfy an enabled constraint.
- If you enable a trigger, Oracle fires the trigger whenever its triggering condition is satisfied.

To enable a single trigger, use the ENABLE option of ALTER TRIGGER on page 4-141.

See also “Enabling and Disabling Constraints” on page 4-419.

Prerequisites

An ENABLE clause that enables an integrity constraint can appear in either a CREATE TABLE or ALTER TABLE statement. To enable a constraint in this manner, you must have the privileges necessary to issue one of these statements. For information on these privileges, see CREATE TABLE on page 4-306 or ALTER TABLE on page 4-106.

If you enable a UNIQUE or PRIMARY KEY constraint, Oracle creates an index on the columns of the unique or primary key in the schema containing the table. To enable such a constraint, you must have the privileges necessary to create the index. For information on these privileges, see CREATE INDEX on page 4-237.

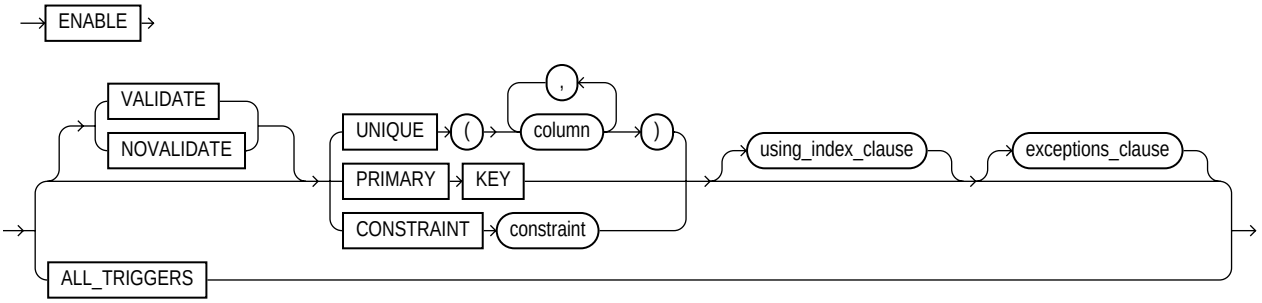
If you enable a referential integrity constraint, the referenced UNIQUE or PRIMARY KEY constraint must already be enabled.

For an integrity constraint to appear in an ENABLE clause, either

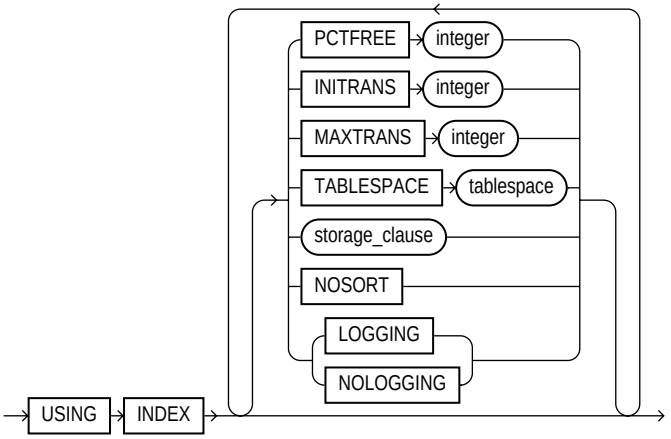
- the integrity constraint must be defined in the containing statement, or
- the integrity constraint must already have been defined and disabled in a previously issued statement.

An ENABLE clause that enables triggers can appear only in an ALTER TABLE statement. To enable triggers with the ENABLE clause, you must have the privileges necessary to issue the ALTER TABLE statement. For information on these privileges, see ALTER TABLE on page 4-106. Also, the triggers must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

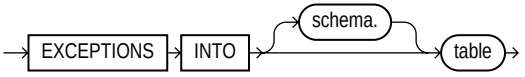
Syntax



using_index_clause::=



exceptions_clause::=



storage_clause: See the STORAGE clause on page 4-523.

Keywords and Parameters

VALIDATE ensures that all new insert, delete, and update operations on the constrained data comply with the constraint, and that all old data also obeys the constraint. An enabled and validated constraint guarantees that all data is and will continue to be valid. This is the default. See also "How Oracle Validates Integrity Constraints" on page 4-421.

NOVALIDATE	ensures that all new insert, update, delete operations on the constrained data comply with the constraint. Oracle does not verify that existing data in the table complies with the constraint.
UNIQUE	enables the UNIQUE constraint defined on the specified column or combination of columns.
PRIMARY KEY	enables the table's PRIMARY KEY constraint.
CONSTRAINT	enables the integrity constraint named *constraint*.
using_index_clause	specifies parameters for the index Oracle creates to enforce a UNIQUE or PRIMARY KEY constraint. Oracle gives the index the same name as the constraint. You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, STORAGE, and PCTFREE parameters for the index. For information on these parameters, see CREATE TABLE on page 4-306. For a description of NOSORT and LOGGING / NOLOGGING, see CREATE INDEX on page 4-237. Use these parameters only when enabling UNIQUE and PRIMARY KEY constraints.
exceptions_clause:	
EXCEPTIONS INTO	specifies a table into which Oracle places information about rows that violate the integrity constraint. The table must exist on your local database before you use this option. If you omit *schema*, Oracle assumes the exception table is in your own schema. See also "How to Identify Exceptions" on page 4-421.
ALL TRIGGERS	enables all triggers associated with the table. You can use this option only in an ENABLE clause in an ALTER TABLE statement, not in a CREATE TABLE statement. See also "Enabling Triggers" on page 4-424.

Enabling and Disabling Constraints

Constraints can have one of three states:

- DISABLE
- ENABLE NOVALIDATE
- ENABLE VALIDATE

Taking a constraint from a DISABLE to ENABLE VALIDATE state requires an exclusive lock on the table, because while Oracle is checking all old data for validity, no new data can be entered into the table. Due to this behavior, only one constraint can be enabled at a time and each new constraint must check all existing rows by serial scan. (Placing constraints concurrently in the ENABLE VALIDATE state requires that you issue multiple ALTER TABLE commands from separate sessions.)

To avoid locking the table, place the constraint in the ENABLE NOVALIDATE state. This state ensures that all new DML statements on the table are validated; therefore, Oracle does not need prevent concurrent access to the table.

ENABLE NOVALIDATE also allows you to place several of the table's constraints in the ENABLE VALIDATE state concurrently. Each scan that Oracle performs to validate existing data can also be performed in parallel when possible.

Enabling Primary Key and Unique Key Constraints

Enabling a primary key or unique key constraint automatically creates a unique index to enforce the constraint. This index is dropped if the constraint is subsequently disabled, thus causing Oracle to rebuild the index every time the constraint is enabled.

To avoid this behavior, create new primary key and unique key constraints initially disabled. Then create nonunique indexes or use existing nonunique indexes to enforce the constraint. Oracle does not drop the nonunique index when the constraint is disabled, so any ENABLE operation on a primary key or unique key constraint occurs almost instantly because the index already exists. This technique also eliminates redundant indexes.

Enabling Integrity Constraints

You can enable a constraint when you create it (see CREATE TABLE on page 4-306 and ALTER TABLE on page 4-106), or you can enable a disabled constraint with the ENABLE clause. To ensure maximum concurrency and performance, constraints should be created or subsequently enabled as follows:

1. Create the constraint in the DISABLEd state.
2. For primary and unique key constraints, create nonunique indexes for maintaining uniqueness.
3. Place all constraints for a table in the ENABLE NOVALIDATE state. This ensures that any new data entered into the table conforms to the constraint.
4. Place all constraints for a table in the ENABLE VALIDATE state.

To enable disabled constraints, you need only perform steps 3 and 4.

Note that constraints are placed in the ENABLE VALIDATE state by default upon creation. Use the procedure outlined above to avoid the default behavior and thereby ensure maximum performance.

How Oracle Validates Integrity Constraints

When you attempt to place an integrity constraint in ENABLE VALIDATE state, Oracle scans the table and applies the integrity constraint to any existing rows in the table:

- If all rows in the table satisfy the integrity constraint, then Oracle places the integrity constraint in ENABLE VALIDATE state.
- If any row in the table violates the integrity constraint, the integrity constraint remains disabled. Oracle returns an error message indicating the integrity constraint is still disabled.

Once an integrity constraint is in ENABLE VALIDATE state, Oracle applies the integrity constraint whenever an INSERT, UPDATE, or DELETE statement tries to change table data:

- If the new data satisfies the integrity constraint, then Oracle executes the statement.
- If the new data violates the integrity constraint, then Oracle does not execute the statement, but instead generates an error message indicating the integrity constraint violation.

How to Identify Exceptions

An *exception* is a row in a table that violates an integrity constraint. You can request that Oracle identify exceptions to an integrity constraint when you attempt to place it in ENABLE VALIDATE state. If you specify an exception table in your ENABLE clause, Oracle inserts a row into the exception table for each exception. A row of the exception table contains the following information:

- the ROWID of the exception
- the name of the integrity constraint
- the schema and name of the table

A definition of a sample exception table named EXCEPTIONS appears in a SQL script available on your distribution medium. Your exception table must have the same column datatypes and lengths as the sample. The common name of this script is UTLEXCPT.SQL; its exact name and location depend on your operating system. You can request that Oracle send exceptions from multiple enabled integrity constraints to the same exception table.

For index-organized tables, rows that violate a constraint are identified by primary key and not by ROWID. This means that the exception table created for index-organized tables uses a different format. Use the BUILD_EXCEPTIONS_TABLE

procedure in the DBMS_IOT package to create the EXCEPTIONS table for inserting rows from index-organized tables that violate integrity constraints.

Example I. The following example creates the ORDER_EXCEPTIONS table to hold rows from an index-organized table ORDERS that violate integrity constraint CHECK_ORDERS:

```
CREATE TABLE orders
  (ord_num NUMBER PRIMARY KEY,
   ord_quantity NUMBER) ORGANIZATION INDEX;

EXECUTE DBMS_IOT.BUILD_EXCEPTIONS_TABLE
  ('SCOTT', 'ORDERS', 'ORDER_EXCEPTIONS');

ALTER TABLE orders
  ADD CONSTRAINT CHECK_ORDERS CHECK(ord_quantity > 0)
  EXCEPTIONS INTO ORDER_EXCEPIONS;
```

To specify an exception table in an ENABLE VALIDATE clause, you must have the privileges necessary to insert rows into the table. For information on these privileges, see ALTER TABLE on page 4-106. To examine the identified exceptions, you must have the privileges necessary to query the exceptions table. For information on these privileges, see SELECT on page 4-489.

If a CREATE TABLE statement contains both the AS clause and an ENABLE VALIDATE clause with the EXCEPTIONS option, Oracle ignores the EXCEPTIONS option. If there are any exceptions, Oracle does not create the table and returns an error message.

Example II. The following statement creates the DEPT table, defines a PRIMARY KEY constraint, and places it in ENABLE VALIDATE state:

```
CREATE TABLE dept
    (deptno  NUMBER(2) PRIMARY KEY,
     dname   VARCHAR2(10),
     loc     VARCHAR2(9) )
     TABLESPACE user_a
     ENABLE VALIDATE PRIMARY KEY;
```

Example III. The following statement places in ENABLE VALIDATE state an integrity constraint named FK_DEPTNO in the EMP table:

```
ALTER TABLE emp
    ENABLE VALIDATE CONSTRAINT fk_deptno
        EXCEPTIONS INTO except_table;
```

Each row of the EMP table must satisfy the constraint for Oracle to enable the constraint. If any row violates the constraint, the constraint remains disabled. Oracle lists any exceptions in the table EXCEPT_TABLE. You can query this table with the following statement:

```
SELECT *
    FROM except_table;
```

The output of this query might look like this:

```
ROW_ID              OWNER   TABLE_NAME  CONSTRAINT
------------------ -----  ----------  ----------
AAAAZzAABAAABrXAAA  SCOTT   EMP         FK_DEPTNO
```

You can also identify the exceptions in the EMP table with the following statement:

```
SELECT emp.*
    FROM emp, except_table
    WHERE emp.row_id except_table.row_id
      AND except_table.table_name = 'EMP'
      AND except_table.constraint = 'FK_DEPTNO';
```

If there are exceptions to the FK_DEPTNO constraint, the output of this query might look like this:

```
EMPNo  ENAME    JOB       MGR   HIREDATE     SAL     COMM   DEPTNO
----  --------  -----  ------  ---------  ------  --------  ------
8001    JACK    CLERK     778  25-AUG-92    1100                70
```

Example IV. The following statement tries to place in ENABLE NOVALIDATE state two constraints on the EMP table:

```
ALTER TABLE emp
    ENABLE NOVALIDATE UNIQUE (ename)
    ENABLE NOVALIDATE CONSTRAINT nn_ename;
```

The preceding statement has two ENABLE clauses:

- The first places a UNIQUE constraint on the ENAME column in ENABLE NOVALIDATE state.
- The second places the constraint named NN_ENAME in ENABLE NOVALIDATE state.

In this case, Oracle only enables the constraints if both are satisfied by each row in the table. If any row violates either constraint, Oracle returns an error message and both constraints remain disabled.

To place the constraints in Example IV in ENABLE VALIDATE state, issue the following:

```
ALTER TABLE emp
  ENABLE VALIDATE UNIQUE (ename)
  ENABLE VALIDATE CONSTRAINT nn_ename;
```

This method of enabling constraints allows both constraints to be enabled concurrently, because they were both previously in the ENABLE NOVALIDATE state. This method also allows each constraint to be enabled in parallel.

Enabling Triggers

You can enable all triggers associated with the table by including the ALL TRIGGERS option in an ENABLE clause of an ALTER TABLE statement. After you enable a trigger, Oracle fires the trigger whenever a triggering statement is issued that meets the condition of the trigger restriction. When you create a trigger, Oracle enables it automatically.

Example. The following statement enables all triggers associated with the EMP table:

```
ALTER TABLE emp
    ENABLE ALL TRIGGERS;
```

Related Topics

ALTER TABLE on page 4-106
ALTER TRIGGER on page 4-141
CONSTRAINT clause on page 4-188
CREATE TABLE on page 4-306
CREATE TRIGGER on page 4-333
DISABLE clause on page 4-380
SET CONSTRAINT(S) on page 4-514
STORAGE clause on page 4-523

EXPLAIN PLAN

Purpose

To determine the execution plan Oracle follows to execute a specified SQL statement. This command inserts a row describing each step of the execution plan into a specified table. If you are using cost-based optimization, this command also determines the cost of executing the statement. See also "Using EXPLAIN PLAN" on page 4-426, "EXPLAIN PLAN and Partitioned Tables" on page 4-427, and "EXPLAIN PLAN and Parallel DML" on page 4-430.

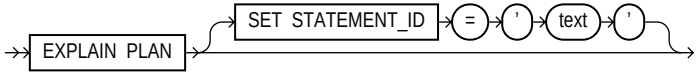
Prerequisites

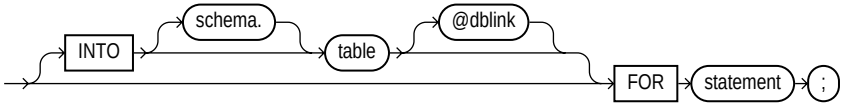
To issue an EXPLAIN PLAN statement, you must have the privileges necessary to insert rows into an existing output table that you specify to hold the execution plan. For information on these privileges, see INSERT on page 4-451.

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, 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, 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. For more information on these privileges, see SELECT on page 4-489.

Syntax





Keywords and Parameters

SET STATEMENT_ID	specifies the value of the STATEMENT_ID column for the rows of the execution plan in the output table. If you omit this clause, the STATEMENT_ID value defaults to null.
INTO	specifies the schema, name, and database containing the output table. This table must exist before you use the EXPLAIN PLAN command. If you omit *schema*, Oracle 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. For information on referring to database links, see the section, "Referring to Objects in Remote Databases" on page 2-54. You can specify a remote output table only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes the table is on your local database. If you omit the INTO clause altogether, Oracle assumes an output table named PLAN_TABLE in your own schema on your local database.
FOR *statement*	specifies a SELECT, INSERT, UPDATE, or DELETE statement for which the execution plan is generated.

Using EXPLAIN PLAN

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 datatypes as this table. The common name of this script is UTLXPLAN.SQL the exact name and location depend on your operating system.

The value you specify in the SET STATEMENT_ID clause appears in the STATEMENT_ID column in the rows of the execution plan. 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.

The EXPLAIN PLAN command is a data manipulation language (DML) command, rather than a data definition language (DDL) command. Therefore, Oracle 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, you must commit the transaction containing the statement.

Do not use the EXPLAIN PLAN command to determine the execution plans of SQL statements that access data dictionary views or dynamic performance tables.

You can also issue the EXPLAIN PLAN command as part of the SQL trace facility. For information on how to use the SQL trace facility and how to interpret execution plans, see *Oracle8 Tuning*.

Example. This EXPLAIN PLAN statement determines the execution plan and cost for an UPDATE statement and inserts rows describing the execution plan into the specified OUTPUT table with the STATEMENT_ID value of 'Raise in Chicago':

```
EXPLAIN PLAN
    SET STATEMENT_ID = 'Raise in Chicago'
    INTO output
    FOR UPDATE emp
        SET sal = sal * 1.10
        WHERE deptno =  (SELECT deptno
                  FROM dept
                  WHERE loc = 'CHICAGO');
```

This SELECT statement queries the OUTPUT table and returns the execution plan and the cost:

```
SELECT LPAD(' ',2*(LEVEL-1))||operation operation, options,
object_name, position
    FROM output
    START WITH id = 0 AND statement_id = 'Raise in Chicago'
    CONNECT BY PRIOR id = parent_id AND
    statement_id = 'Raise in Chicago';
```

The query returns this execution plan:

```
OPERATION             OPTIONS     OBJECT_NAME     POSITION
-------------------------------------------------------------
UPDATE STATEMENT                                         1
    FILTER                                               0
      TABLE ACCESS    FULL        EMP                    1
      TABLE ACCESS    FULL        DEPT                   2
```

The value in the POSITION column of the first row shows that the statement has a cost of 1.

EXPLAIN PLAN and Partitioned Tables

Information for partitioning is provided in the steps (rows of the Explain table) of the Explain plan for a SQL statement. The information consists of:

- three columns of the Explain table: partition_start, partition_stop, partition_id
- a step (row) of the EXPLAIN table labeled PARTIITON
- enhancements to the TABLE ACCESS and INDEX steps when such steps refer to partitioned objects

Partition Columns of the Explain Table

The **partition_start** and **partition_stop** columns describe how the partitions being accessed are computed by Oracle and provide the range of accessible partitions (if known).

The **partition_start** column describes the start partition of a range of accessed partitions. It can take one of these values:

- NUMBER(*n*) indicates that the start partition has been identified by the SQL compiler and its partition number is given by *n*.
- KEY indicates that the start partition will be identified at execution time from partitioning key values.
- ROW LOCATION indicates that the start partition (same as the stop partition) will be computed at execution time from the location of each record being retrieved. The record location is obtained by a user or from a global index.
- INVALID indicates that the range of accessed partitions is empty.

The **partition_stop** column describes the stop partition of a range of accessed partitions. It can take these values:

- NUMBER(*n*) indicates that the stop partition has been identified by the SQL compiler and its partition number is given by *n*.
- KEY indicates that the stop partition will be identified at execution time from partitioning key values.
- ROW LOCATION indicates that the stop partition (same as the start partition) will be computed at execution time from the location of each record being retrieved. The record location is obtained by a user or from a global index.
- INVALID indicates that the range of accessed partitions is empty.

The **partition_id** column identifies the step that has computed a pair of values of the partition_start and partition_stop columns.

Partition Step of Explain Table

The PARTITION step describes partition boundaries applicable to a single partitioned object (table or index) or to a set of equi-partitioned objects (a partitioned table and its local indexes). The partition boundaries are provided by the values of partition_start and partition_stop of the PARTITION step. Possible values for partition_start and partition_stop are NUMBER(*n*), KEY, and INVALID.

The **options** column of a PARTITION step can take these values:

- CONCATENATED indicates that the PARTITION step concatenates the result sets returned from accessed partitions.
- SINGLE indicates that the set of partitions to be accessed consists of a single partition to be determined at execution time.
- EMPTY indicates that the set of partitions to be accessed is empty.

Modified Steps (rows) of Explain Table

The TABLE ACCESS and INDEX steps describing access to a partitioned table or index are enhanced to provide partition boundary information in the partition_ start, partition_stop, and partition_id columns.

The partition boundaries may have been computed by:

- a previous PARTITION step, in which case the partition_start and partition_ stop column values replicate the values present in the PARTITION step, and the partition_id contains the ID of the PARTITION step. Possible values for partition_start and partition_stop are NUMBER(n), KEY, INVALID.
- the TABLE ACCESS or INDEX step itself, in which case the partition_id contains the ID of the step. Possible values for partition_start and partition_ stop are NUMBER(n), KEY, ROW LOCATION (TABLE ACCESS only), and INVALID.

The **options** column of a TABLE ACCESS step describing access by ROWID to a table may contain the following values:

- “BY USER ROWID” if the table rows are located using user-supplied ROWIDs.
- “BY INDEX ROWID” if the table is nonpartitioned and rows are located using index(es).
- “BY GLOBAL INDEX ROWID” if the table is partitioned and rows are located using only global indexes.
- “BY LOCAL INDEX ROWID” if the table is partitioned and rows are located using one or more local indexes and possibly some global indexes.

Example. Assume that STOCKS is a table that is 8-way partitioned according to a STOCK_NUM column, and that a local prefixed index STOCK_IX on column STOCK_NUM exists. The partition HIGHVALUES are 1000, 2000, 3000, 4000, 5000, 6000, 7000, and 8000.

Consider the query:

```
SELECT * FROM stocks WHERE stock_num BETWEEN 3800 AND: h;
```

EXPLAIN PLAN executes this query with PLAN_TABLE as the output table. The basic execution plan, including partitioning information, is obtained with the query:

```
SELECT id, operation, options, object_name,
  partition_start, partition_stop, partition_id FROM plan_table;
```

EXPLAIN PLAN and Parallel DML

When you use EXPLAIN PLAN to determine the execution of a statement that includes the PARALLEL option, the resulting execution plan will indicate parallel execution. Note, however, that 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 restriction of one parallel DML statement per transaction, and the statement will be executed serially.

To maintain parallel execution of the statements, you must commit or roll back the EXPLAIN PLAN command, and then submit the parallel DML statement.

Related Topics

Oracle8 Tuning

Filespec

Purpose

- To specify a file as a datafile
- To specify a group of one or more files as a redo log file group.

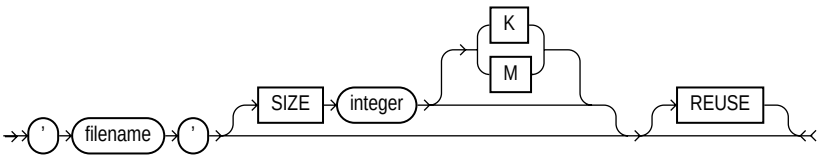
For illustrations, see “Examples” on page 4-432.

Prerequisites

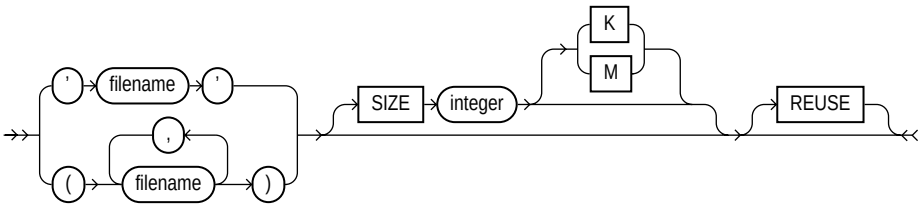
A *filespec* can appear in CREATE DATABASE, ALTER DATABASE, CREATE TABLESPACE, or ALTER TABLESPACE commands. You must have the privileges necessary to issue one of these commands. For information on these privileges, see CREATE DATABASE on page 4-219, ALTER DATABASE on page 4-15, CREATE TABLESPACE on page 4-328, and ALTER TABLESPACE on page 4-133.

Syntax

filespec_data_files ::=



filespec_redo_log_file_groups ::=



Keywords and Parameters

'filename'	is the name of either a datafile or a redo log file member. A '*filename*' can only contain single-byte such as characters from 7-bit ASCII or EBCDIC character sets. Multibyte characters are not valid. A redo log file group can have one or more members, or copies. Each '*filename*' must be fully specified according to the conventions for your operating system.
SIZE *integer*	specifies the size of the file. If you omit this parameter, the file must already exist. Note that the tablespace size must be one block greater than the sum of the sizes of the objects contained in it. You can use K or M to specify the size in kilobytes or megabytes.
REUSE	allows Oracle to reuse an existing file. If the file already exists, Oracle verifies that its size matches the value of the SIZE parameter. If the file does not exist, Oracle creates it. If you omit this option, the file must not already exist and Oracle creates the file. The REUSE option is significant only when used with the SIZE option. If you omit the SIZE option, Oracle expects the file to exist already. **Note**: Whenever Oracle uses an existing file, the file's previous contents are lost.

Examples

Example I. The following statement creates a database named PAYABLE that has two redo log file groups, each with two members, and one datafile:

```
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.dat' SIZE 30M;
```

The first *filespec* 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 with size 50 kilobytes.

The second *filespec* 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 with sizes of 50 kilobytes.

The *filespec* in the DATAFILE clause specifies a datafile named 'DISKC:DBONE.DAT' of size 30 megabytes.

All of these *filespecs* specify a value for the SIZE parameter and omit the REUSE option, so none of these files can already exist. Oracle must create them.

Example II. 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 *filespec* 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' with sizes of 50 kilobytes each. Since the *filespec* specifies the REUSE option, each member can already exist. If a member exists, it must have a size of 50 kilobytes. If it does not exist, Oracle creates it with that size.

Example III. The following statement creates a tablespace named STOCKS that has three datafiles:

```
CREATE TABLESPACE stocks
   DATAFILE 'diskc:stock1.dat',
            'diskc:stock2.dat',
            'diskc:stock3.dat';
```

The *filespecs* for the datafiles specifies files named 'DISKC:STOCK1.DAT', 'DISKC:STOCK2.DAT', and 'DISKC:STOCK3.DAT'. Since each *filespec* omits the SIZE parameter, each file must already exist.

Example IV. The following statement alters the STOCKS tablespace and adds a new datafile:

```
ALTER TABLESPACE stocks
   ADD DATAFILE 'diskc:stock4.dat' REUSE;
```

The *filespec* specifies a datafile named 'DISKC:STOCK4.DAT'. Since the *filespec* omits the SIZE parameter, the file must already exist and the REUSE option is not significant.

Related Topics

CREATE DATABASE on page 4-219
ALTER DATABASE on page 4-15
CREATE TABLESPACE on page 4-328
ALTER TABLESPACE on page 4-133

GRANT (System Privileges and Roles)

Purpose

To grant system privileges and roles to users and roles. To grant object privileges, use the GRANT command (Object Privileges) described in the next section of this chapter. For more information, see "Granting System Privileges and Roles" on page 4-435. For illustrations, see "Examples" on page 4-442.

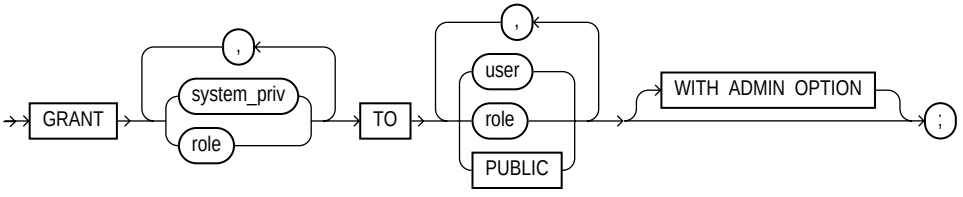
> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

To grant a system privilege, you must either have been granted the system privilege with the ADMIN OPTION or have been granted GRANT ANY PRIVILEGE system privilege.

To grant a role, you must either have been granted the role with the ADMIN OPTION or have been granted GRANT ANY ROLE system privilege, or you must have created the role. See also "Other Authorization Methods" on page 4-442.

Syntax



Keywords and Parameters

system_priv	is a system privilege to be granted.
role	is a role to be granted.
TO	identifies users or roles to which system privileges and roles are granted.

PUBLIC	grants system privileges or roles to all users.
WITH ADMIN OPTION	grant the system privilege or role to other users or roles. If you grant a role with ADMIN OPTION, the grantee can also alter or drop the role. See also "Granting the ADMIN OPTION" on page 4-441.

Granting System Privileges and Roles

Use this form of the GRANT command to grant both system privileges and roles to users, roles, and PUBLIC. Table 4–10 indicates which user or role can be given which authorizations:

Table 4–10 DB GRANTS Allowed

User Type -> / Type of Role	Grant to User Identified by Password	Grant to User Identified Externally	Grant to User Identified Globally	Grant to Local Role	Grant to External Role	Grant to Global Role
Privileges	Yes	Yes	Yes	Yes	Yes	Yes
Local Role	Yes	Yes	Yes	Yes	Yes	Yes
External Role	Yes	Yes	No	Yes	Yes	No
Global Role	No	No	No	No	No	No

If you grant **a privilege to a user**, Oracle adds the privilege to the user's privilege domain. The user can immediately exercise the privilege.

If you grant **a privilege to a role**, Oracle adds the privilege to the role's privilege domain. 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.

If you grant **a privilege to PUBLIC**, Oracle adds the privilege to the privilege domains of each user. All users can immediately perform operations authorized by the privilege.

If you grant **a role to a user**, Oracle makes the role available to the user. The user can immediately enable the role and exercise the privileges in the role's privilege domain.

If you grant **a role to another role**, Oracle adds the granted role's privilege domain to the grantee role's privilege domain. 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**, Oracle makes the role available to all users. All users can immediately enable the role and exercise the privileges in the roles privilege domain.

In addition, the following restrictions apply:

- A privilege or role cannot appear more than once in the list of privileges and roles to be granted.
- A user, role, or PUBLIC cannot appear more than once in the TO clause.
- You cannot grant roles circularly. For example, if you grant the role BANKER to the role TELLER, you cannot subsequently grant TELLER to BANKER.
- 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 role.

Table 4–11 lists system privileges and the operations that they authorize. You can grant any of these system privileges with the GRANT command.

Table 4–11 System Privileges

System Privilege	Allows grantee to . . .
ALTER ANY CLUSTER	alter any cluster in any schema
ALTER ANY INDEX	alter any index in any schema
ALTER ANY PROCEDURE	alter any stored procedure, function, or package in any schema
ALTER ANY ROLE	alter any role in the database
ALTER ANY SEQUENCE	alter any sequence in the database
ALTER ANY SNAPSHOT	alter any snapshot in the database
ALTER ANY TABLE	alter any table or view in the schema
OBJ ALTER ANY TYPE	alter any type in any schema
ALTER ANY TRIGGER	enable, disable, or compile any database trigger in any schema
ALTER DATABASE	alter the database
ALTER PROFILE	alter profiles
ALTER RESOURCE COST	set costs for session resources
ALTER ROLLBACK SEGMENT	alter rollback segments

Table 4–11 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
ALTER SESSION	issue ALTER SESSION statements
ALTER SYSTEM	issue ALTER SYSTEM statements
ALTER TABLESPACE	alter tablespaces
ALTER USER	alter any user. This privilege authorizes the grantee to ■ change another user's password or authentication method, ■ assign quotas on **any** tablespace, ■ set default and temporary tablespaces, and ■ assign a profile and default roles
ANALYZE ANY	analyze any table, cluster, or index in any schema
AUDIT ANY	audit any object in any schema using AUDIT (Schema Objects) statements
AUDIT SYSTEM	issue AUDIT (SQL Statements) statements
BACKUP ANY TABLE	use the Export utility to incrementally export objects from the schema of other users
BECOME USER	become another user. (Required by any user performing a full database import.)
COMMENT ANY TABLE	Comment on any table, view, or column in any schema
CREATE ANY CLUSTER	create a cluster in any schema. Behaves similarly to CREATE ANY TABLE.
CREATE ANY DIRECTORY	create a directory database object in any schema
CREATE ANY INDEX	create an index in any schema on any table in any schema
CREATE ANY LIBRARY	create external procedure / function libraries in any schema
CREATE ANY PROCEDURE	create stored procedures, functions, and packages in any schema
CREATE ANY SEQUENCE	create a sequence in any schema

Table 4–11 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
CREATE ANY SNAPSHOT	create snapshots in any schema
CREATE ANY SYNONYM	create private synonyms in any schema
CREATE ANY TABLE	create tables in any schema. The owner of the schema containing the table must have space quota on the tablespace to contain the table.
CREATE ANY TRIGGER	create a database trigger in any schema associated with a table in any schema
OBJ CREATE ANY TYPE	create types and type bodies in any schema
CREATE ANY VIEW	create views in any schema
CREATE CLUSTER	create clusters in grantee's schema
CREATE DATABASE LINK	create private database links in grantee's schema
CREATE ANY LIBRARY	create external procedure/function libraries in grantee's schema
CREATE PROCEDURE	create stored procedures, functions, and packages in grantee's schema
CREATE PROFILE	create profiles
CREATE PUBLIC DATABASE LINK	create public database links
CREATE PUBLIC SYNONYM	create public synonyms
CREATE ROLE	create roles
CREATE ROLLBACK SEGMENT	create rollback segments
CREATE SEQUENCE	create sequences in grantee's schema
CREATE SESSION	connect to the database
CREATE SNAPSHOT	create snapshots in grantee's schema
CREATE SYNONYM	create synonyms in grantee's schema
CREATE TABLE	create tables in grantee's schema. To create a table, the grantee must also have space quota on the tablespace to contain the table.
CREATE TABLESPACE	create tablespaces

Table 4–11 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
CREATE TRIGGER	create a database trigger in grantee's schema
OBJ CREATE TYPE	create types and type bodies in grantee's schema
CREATE USER	create users. This privilege also allows the creator to ■ assign quotas on **any** tablespace, ■ set default and temporary tablespaces, and ■ assign a profile as part of a CREATE USER statement.
CREATE VIEW	create views in grantee's schema
DELETE ANY TABLE	■ delete rows from tables or views in any schema ■ truncate tables in any schema
DROP ANY CLUSTER	drop clusters in any schema
DROP ANY DIRECTORY	drop directory database objects
DROP ANY INDEX	drop indexes in any schema
DROP ANY LIBRARY	drop external procedure / function libraries in any schema
DROP ANY PROCEDURE	drop stored procedures, functions, or packages in any schema
DROP ANY ROLE	drop roles
DROP ANY SEQUENCE	drop sequences in any schema
DROP ANY SNAPSHOT	drop snapshots in any schema
DROP ANY SYNONYM	drop private synonyms in any schema
DROP ANY TABLE	drop tables in any schema
DROP ANY TRIGGER	drop database triggers in any schema
OBJ DROP ANY TYPE	drop object types and object type bodies in any schema
DROP ANY VIEW	drop views in any schema
DROP LIBRARY	drop external procedure / function libraries

Table 4–11 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
DROP PROFILE	drop profiles
DROP PUBLIC DATABASE LINK	drop public database links
DROP PUBLIC SYNONYM	drop public synonyms
DROP ROLLBACK SEGMENT	drop rollback segments
DROP TABLESPACE	drop tablespaces
DROP USER	drop users
EXECUTE ANY PROCEDURE	■ execute procedures or functions (standalone or packaged) ■ reference public package variables in any schema
OBJ EXECUTE ANY TYPE	use and reference object types, and invoke methods of any type in any schema. You must grant EXECUTE ANY TYPE to a specific user. You cannot grant EXECUTE ANY TYPE to a role.
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 grantee's in-doubt distributed transactions in the local database
GRANT ANY PRIVILEGE	grant any system privilege.
GRANT ANY ROLE	grant any role in the database
INSERT ANY TABLE	insert rows into tables and views in any schema
LOCK ANY TABLE	lock tables and views in any schema
MANAGE TABLESPACE	take tablespaces offline and online and begin and end tablespace backups
RESTRICTED SESSION	logon after the instance is started using the Server Manager STARTUP RESTRICT command
SELECT ANY SEQUENCE	reference sequences in any schema

Table 4–11 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
SELECT ANY TABLE	query tables, views, or snapshots in any schema
SYSDBA	■ perform Server Manager STARTUP and SHUTDOWN commands, ■ ALTER DATABASE OPEN / MOUNT / BACKUP, ■ CREATE DATABASE, ■ ARCHIVELOG and RECOVERY and ■ includes the RESTRICTED SESSION privilege.
SYSOPER	■ perform Server Manager STARTUP and SHUTDOWN commands, ■ ALTER DATABASE OPEN / MOUNT / BACKUP, ■ ARCHIVELOG and RECOVERY ■ includes the RESTRICTED SESSION privilege.
UNLIMITED TABLESPACE	use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, the grantee's schema objects remain but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
UPDATE ANY TABLE	update rows in tables and views in any schema

Granting the ADMIN OPTION

A grant with the ADMIN OPTION supersedes a previous identical grant without the ADMIN OPTION. If you grant a system privilege or role to a user without the ADMIN OPTION, and then subsequently grant the privilege or role to the user with the ADMIN OPTION, the user has the ADMIN OPTION on the privilege or role.

A grant without the ADMIN OPTION does not supersede a previous grant with the ADMIN OPTION. 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.

Other Authorization Methods

You can authorize database users to use roles through means other than the database and the GRANT statement. For example, some operating systems have facilities that grant operating system privileges to operating system users. You can use such facilities to grant roles to Oracle users with the initialization parameter OS_ROLES. If you choose to grant roles to users through operating system facilities, you cannot also grant roles to users with the GRANT command, although you can use the GRANT command to grant system privileges to users and system privileges and roles to other roles.

For information about other authorization methods, see *Oracle8 Administrator's Guide*

Examples

Example I. To grant the CREATE SESSION system privilege to RICHARD, allowing RICHARD to log on to Oracle, issue the following statement:

```
GRANT CREATE SESSION
TO richard;
```

Example II. To grant the CREATE TABLE system privilege to the role TRAVEL_AGENT, issue the following statement:

```
GRANT CREATE TABLE
TO travel_agent;
```

TRAVEL_AGENT's privilege domain now contains the CREATE TABLE system privilege.

The following statement grants the TRAVEL_AGENT role to the EXECUTIVE role:

```
GRANT travel_agent
TO executive;
```

TRAVEL_AGENT is now granted to EXECUTIVE. EXECUTIVE's privilege domain contains the CREATE TABLE system privilege.

To grant the EXECUTIVE role with the ADMIN OPTION to THOMAS, issue the following statement:

```
GRANT executive
TO thomas
WITH ADMIN OPTION;
```

THOMAS can now perform the following operations with the EXECUTIVE role:

- enable the role and exercise any privileges in the role's privilege domain, including the CREATE TABLE system privilege
- grant and revoke the role to and from other users
- drop the role

Related Topics

ALTER USER on page 4-150
CREATE USER on page 4-357
GRANT (Object Privileges) on page 4-444
REVOKE (System Privileges and Roles) on page 4-475

GRANT (Object Privileges)

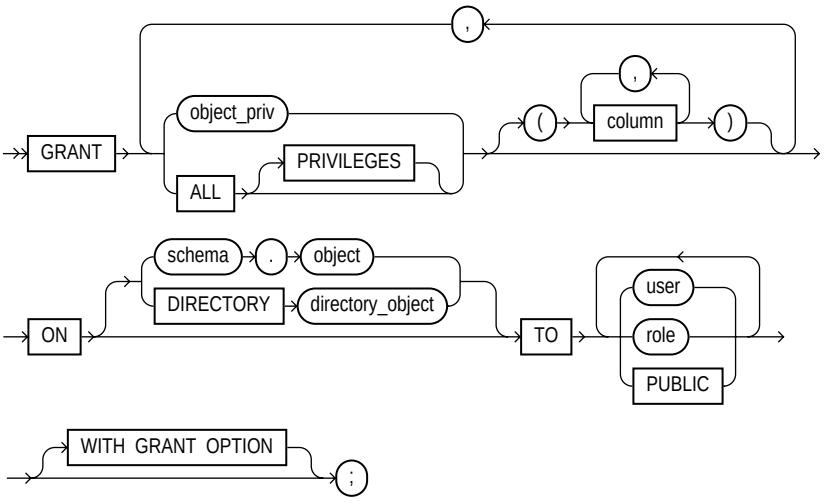
Purpose

To grant privileges for a particular object to users and roles. To grant system privileges and roles, use the GRANT command (System Privileges and Roles) described in the previous section of this chapter. See also “Database Object Privileges” on page 4-446. For illustrations, see “Examples” on page 4-449.

Prerequisites

You must own the object or the owner of the object must have granted you the object privileges with the GRANT OPTION. This rule applies to users with the DBA role.

Syntax



Keywords and Parameters

object_priv is an object privilege to be granted. You can substitute any of the following values:

ALTER

EXECUTE

INDEX

INSERT

READ

REFERENCES

SELECT

UPDATE

ALL [PRIVILEGES] grants 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 optional.)

column specifies a table or view column on which privileges are granted. You can specify columns only when granting the INSERT, REFERENCES, or UPDATE privilege. If you do not list columns, the grantee has the specified privilege on all columns in the table or view.

ON identifies the object on which the privileges are granted.

DIRECTORY identifies a *directory_object* on which privileges are granted by the DBA. You cannot qualify *directory_object* with a schema name. See also "Directory Privileges" on page 4-448.

See CREATE DIRECTORY on page 4-230.

object identifies the schema object on which the privileges are granted. If you do not qualify object with *schema*, Oracle assumes the object is in your own schema. The object can be one of the following types:

- table
- view
- sequence
- procedure, function, or package
- snapshots
- synonym for a table, view, sequence, snapshot, procedure, function, or package (see also "Synonym Privileges" on page 4-448)
- library
- object types

TO	identifies users or roles to which the object privilege is granted.
	PUBLIC grants object privileges to all users.
WITH GRANT OPTION	allows the grantee to grant the object privileges to other users and roles. The grantee must be a user or PUBLIC, rather than a role.

Database Object Privileges

You can use this form of the GRANT statement to grant object privileges to users, roles, and PUBLIC. Each database object privilege that you grant authorizes the grantee to perform some operation on the object. Table 4–12 summarizes the object privileges that you can grant on each type of object.

Table 4–12 Object Privileges

Object Privilege	Table	View	Sequence	Procedure-Functions-Packages	Snapshot	Directory	Library
ALTER	X		X				
DELETE	X	X			X[a]		
EXECUTE				X			X
INDEX	X						
INSERT	X	X			X[a]		
READ						X	
REFERENCES	X						
SELECT	X	X	X		X		
UPDATE	X	X			X[a]		

[a] The DELETE, INSERT, and UPDATE privileges can be granted only to *updatable* snapshots.

If you grant **a privilege to a user,** Oracle adds the privilege to the user's privilege domain. The user can immediately exercise the privilege.

If you grant **a privilege to a role,** Oracle adds the privilege to the role's privilege domain. 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.

If you grant **a privilege to PUBLIC,** Oracle adds the privilege to the privilege domain of each user. All users can immediately exercise the privilege.

- A privilege cannot appear more than once in the list of privileges to be granted.
- A user or role cannot appear more than once in the TO clause.

Table 4–13 lists object privileges and the operations that they authorize. You can grant any of these system privileges with the GRANT command.

Table 4–13 Object Privileges and the Operations They Authorize

Object Privilege	Allows Grantee to . . .
Table privileges authorize operations on a table. Any one of following object privileges allows the grantee to lock the table in any lock mode with the LOCK TABLE command.	
ALTER	allows the grantee to change the table definition with the ALTER TABLE command.
DELETE	remove rows from the table with the DELETE command.
	Note: You must grant the SELECT privilege on the table along with the DELETE privilege.
INDEX	create an index on the table with the CREATE INDEX command.
INSERT	add new rows to the table with the INSERT command.
REFERENCES	create a constraint that refers to the table. You cannot grant this privilege to a role.
SELECT	query the table with the SELECT command.
UPDATE	change data in the table with the UPDATE command.
	Note: You must grant the SELECT privilege on the table along with the UPDATE privilege.
View privileges authorizes operations on a view. Any one of the above object privileges allows the grantee to lock the view in any lock mode with the LOCK TABLE command.	
To grant a privilege on a view, you must have that privilege with the GRANT OPTION on all of the view's base tables.	
DELETE	remove rows from the view with the DELETE command.
INSERT	add new rows to the view with the INSERT command.
SELECT	query the view with the SELECT command.
UPDATE	change data in the view with the UPDATE command.
Sequence privileges authorize operations on a sequence.	

Table 4–13 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Allows Grantee to . . .
ALTER	change the sequence definition with the ALTER SEQUENCE command.
SELECT	examine and increment values of the sequence with the CURRVAL and NEXTVAL pseudocolumns.
Procedure, function, and package privileges authorize operations on procedures, functions, or packages.	
EXECUTE	execute the procedure or function or to access any program object declared in the specification of a package.
Snapshot privileges authorize operations on a snapshot.	
SELECT	query the snapshot with the SELECT command.
Synonym privileges are the same as the privileges for the base object. See "Synonym Privileges" below.	
Directory privileges provide secured access to the files stored in the operating system directory. See "Directory Privileges" below.	
READ	read files in the directory.

Synonym Privileges

The object privileges available for a synonym are the same as the privileges for the synonym's base 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 a user a privilege on a synonym, the user can use either the synonym name or the base object name in the SQL statement that exercises the privilege.

Directory Privileges

The object privileges available for a directory provide secured database 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 pathname of the operating system directory where the files reside. Because the files are actually stored outside the database, Oracle8 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 Oracle8 to enforce security during file operations.

Examples

Example I. To grant READ on directory BFILE_DIR1 to user SCOTT, with the GRANT OPTION, issue the following statement:

```
GRANT READ ON DIRECTORY bfile_dir1 TO scott
WITH GRANT OPTION;
```

Example II. To grant all privileges on the table BONUS to the user JONES with the GRANT OPTION, issue the following statement:

```
GRANT ALL ON bonus TO jones
WITH GRANT OPTION;
```

JONES can subsequently perform the following operations:

- exercise any privilege on the BONUS table
- grant any privilege on the BONUS table to another user or role

Example III. To grant SELECT and UPDATE privileges on the view GOLF_HANDICAP to all users, issue the following statement:

```
GRANT SELECT, UPDATE
ON golf_handicap TO PUBLIC;
```

All users can subsequently query and update the view of golf handicaps.

Example IV. To grant SELECT privilege on the ESEQ sequence in the schema ELLY to the user BLAKE, issue the following statement:

```
GRANT SELECT
ON elly.eseq TO blake;
```

BLAKE can subsequently generate the next value of the sequence with the following statement:

```
SELECT elly.eseq.NEXTVAL
FROM DUAL;
```

Example V. To grant BLAKE the REFERENCES privilege on the EMPNO column and the UPDATE privilege on the EMPNO, SAL, and COMM columns of the EMP table in the schema SCOTT, issue the following statement:

```
GRANT REFERENCES (empno), UPDATE (empno, sal, comm)
ON scott.emp
TO blake;
```

BLAKE can subsequently update values of the EMPNO, SAL, and COMM columns. BLAKE can also define referential integrity constraints that refer to the EMPNO column. However, because the GRANT statement lists only these columns, BLAKE cannot perform operations on any of the other columns of the EMP table.

For example, BLAKE can create a table with a constraint:

```
CREATE TABLE dependent
(dependno NUMBER,
dependname VARCHAR2(10),
employee NUMBER
CONSTRAINT in_emp REFERENCES scott.emp(empno) );
```

The constraint IN_EMP ensures that all dependents in the DEPENDENT table correspond to an employee in the EMP table in the schema SCOTT.

Related Topics

GRANT (System Privileges and Roles) on page 4-434
REVOKE (Schema Object Privileges) on page 4-478

INSERT

Purpose

To add rows to:

- a table
- a view's base table
- a partition of a partitioned table
- OBJ an object table
- OBJ an object view's base table

For illustrations of inserting, see "Examples" on page 4-455.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

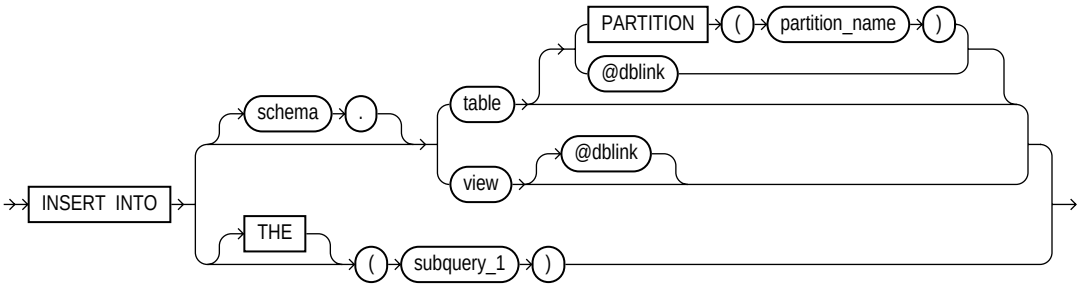
Prerequisites

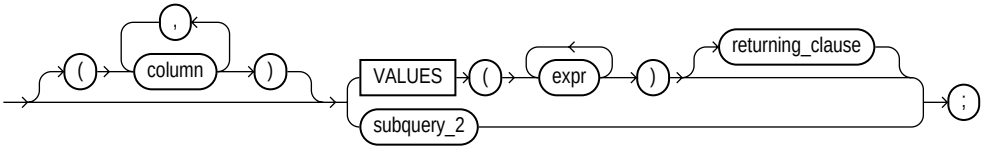
For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

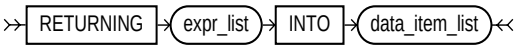
If you have the INSERT ANY TABLE system privilege, you can also insert rows into any table or any view's base table.

Syntax

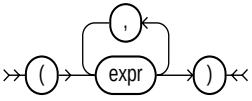




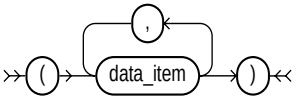
returning_clause ::=



expr_list ::=



data_item_list ::=



Keywords and Parameters

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table / view	is the name of the table or object table into which rows are to be inserted. If you specify a view or object view, Oracle inserts rows into the view's base table. See also "Inserting Into Views" on page 4-454.
PARTITION *(partition_name)*	specifies partition-level row inserts for *table*. The *partition_name* is the name of the partition within table targeted for inserts. **Note:** OBJ This option is not valid for object tables or object views.
dblink	is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see "Referring to Schema Objects and Parts" on page 2-51. You can only insert rows into a remote table or view if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes that the table or view is on the local database.
OBJ THE	informs Oracle that the column value returned by the subquery is a nested table, not a scalar value. A subquery prefixed by THE is called a *flattened subquery*. See "Using Flattened Subqueries" on page 4-533.
subquery_1	is a subquery that Oracle treats in the same manner as a view. See "Subqueries" on page 4-530.
column	is a column of the table or view. In the inserted row, each column in this list is assigned a value from the VALUES clause or the subquery. If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If you omit the column list altogether, the VALUES clause or query must specify values for all columns in the table.
VALUES	specifies a row of values to be inserted into the table or view. See the syntax description in "Expressions" on page 3-78. You must specify a value in the VALUES clause for each column in the column list. See also "The VALUES Clause and Subqueries" on page 4-454.
subquery_2	is a subquery that returns rows that are inserted into the table. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. See "Subqueries" on page 4-530.
RETURNING	retrieves the rows affected by the INSERT. You can retrieve only scalar, LOB, ROWID, and REF types. See also "The RETURNING Clause" on page 4-455.
expr	is some of the syntax descriptions in "Expressions" on page 3-78. You must specify a column expression in the RETURNING clause for each variable in the *data_item_list*.
INTO	indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item_list*.
data_item	is a PL/SQL variable or bind variable that stores a retrieved *expr* value.

You cannot use the RETURNING clause with Parallel DML or with remote objects. See also "Parallel DML" on page 4-454.

The VALUES Clause and Subqueries

An INSERT statement with a VALUES clause adds to the table a single row containing the values specified in the VALUES clause.

An INSERT statement with a subquery instead of a VALUES clause adds to the table all rows returned by the subquery. Oracle processes the subquery and inserts each returned row into the table. If the subquery selects no rows, Oracle inserts no rows into the table. The subquery can refer to any table, view, or snapshot, including the target table of the INSERT statement.

The number of columns in the column list of the INSERT statement must be the same as the number of values in the VALUES clause or the number of columns selected by the subquery. If you omit the column list, then the VALUES clause or the subquery must provide values for every column in the table.

Oracle assigns values to fields in new rows based on the internal position of the columns in the table and the order of the values in the VALUES clause or in the select list of the query. You can determine the position of each column in the table by examining the data dictionary. See *Oracle8 Reference*.

If you omit any columns from the column list, Oracle assigns them their default values as specified when the table was created. For more information on default column values, see CREATE TABLE on page 4-306. If any of these columns has a NOT NULL constraint, then Oracle returns an error indicating that the constraint has been violated and rolls back the INSERT statement.

Issuing an INSERT statement against a table fires any INSERT triggers defined on the table.

Parallel DML

You can place a parallel hint immediately after the INSERT keyword to parallelize an INSERT operation. Parallel DML must also be enabled for the session. See ALTER SESSION on page 4-58 for information about enabling parallel DML. For detailed information about Parallel DML, see *Oracle8 Tuning*, *Oracle8 Parallel Server Concepts and Administration*, and *Oracle8 Concepts*.

Inserting Into Views

If a view was created using the WITH CHECK OPTION, then you can insert into the view only rows that satisfy the view's defining query.

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 if the view's defining query contains one of the following constructs:

- set operator
- GROUP BY clause
- group function
- DISTINCT operator
- flattened subqueries
- nested table columns
- CAST and MULTISET expressions

The RETURNING Clause

An INSERT statement with a RETURNING clause retrieves the rows inserted and stores them in PL/SQL variables or bind variables. Using a RETURNING clause in INSERT statements with a VALUES clause enables you to return column expressions, ROWIDs, and REFs and store them in output bind variables. You can also use INSERT with a RETURNING clause for views with single base tables.

PL/SQL does not allow multiple row inserts; you can retrieve only a single row value into a PL/SQL variable. For information about using the RETURNING clause, see the *PL/SQL User's Guide and Reference*.

Examples

Example I. The following statement inserts a row into the DEPT table:

```
INSERT INTO dept
   VALUES (50, 'PRODUCTION', 'SAN FRANCISCO');
```

Example II. The following statement inserts a row with six columns into the EMP table. One of these columns is assigned NULL and another is assigned a number in scientific notation:

```
INSERT INTO emp (empno, ename, job, sal, comm, deptno)
   VALUES (7890, 'JINKS', 'CLERK', 1.2E3, NULL, 40);
```

Example III. The following statement has the same effect as Example II:

```
INSERT INTO (select empno, ename, job, sal, comm, deptno from emp)
   VALUES (7890, 'JINKS', 'CLERK', 1.2E3, NULL, 40);
```

Example IV. The following statement copies managers and presidents or employees whose commission exceeds 25% of their salary into the BONUS table:

```
INSERT INTO bonus
   SELECT ename, job, sal, comm
   FROM emp
   WHERE comm > 0.25 * sal
   OR job IN ('PRESIDENT', 'MANAGER');
```

Example V. The following statement inserts a row into the ACCOUNTS table owned by the user SCOTT on the database accessible by the database link SALES:

```
INSERT INTO scott.accounts@sales (acc_no, acc_name)
   VALUES (5001, 'BOWER');
```

Assuming that the ACCOUNTS table has a BALANCE column, the newly inserted row is assigned the default value for this column because this INSERT statement does not specify a BALANCE value.

Example VI. The following statement inserts a new row containing the next value of the employee sequence into the EMP table:

```
INSERT INTO emp
   VALUES  (empseq.nextval, 'LEWIS', 'CLERK',
            7902, SYSDATE, 1200, NULL, 20);
```

Example VII. The following example adds rows from LATEST_DATA into partition OCT96 of the SALES table:

```
INSERT INTO sales PARTITION (oct96)
   SELECT * FROM latest_data;
```

Example VIII. The following example returns the values of the inserted rows into output bind variables :BND1 and :BND2:

```
INSERT INTO emp VALUES (empseq.nextval, 'LEWIS', 'CLARK',
                        7902, SYSDATE, 1200, NULL, 20)
   RETURNING sal*12, job INTO :bnd1, :bnd2;
```

Example IX. The following example returns the reference value for the inserted row into bind array :1:

```
INSERT INTO employee
```

```
   VALUES ('Kitty Mine', 'Peaches Fuzz', 'Meena Katz')
RETURNING REF(employee) INTO :1;
```

Related Topics

DELETE on page 4-374
UPDATE on page 4-542

LOCK TABLE

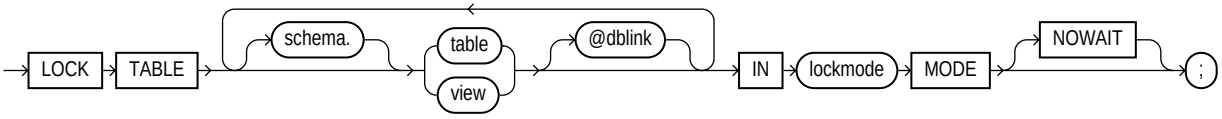
Purpose

To lock one or more tables 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. See also "Locking Tables" on page 4-459.

Prerequisites

The table or view must be in your own schema or you must have LOCK ANY TABLE system privilege or you must have any object privilege on the table or view.

Syntax



Keywords and Parameters

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table / view is the name of the table to be locked. If you specify *view*, Oracle locks the view's base tables.

dblink is a database link to a remote Oracle database where the table or view is located. For information on specifying database links, see the section, "Referring to Objects in Remote Databases" on page 2-54. You can lock tables and views on a remote database only if you are using Oracle's distributed functionality. All tables locked by a LOCK TABLE statement must be on the same database.

If you omit *dblink*, Oracle assumes the table or view is on the local database.

lockmode	is one of the following: **ROW SHARE** allows 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. **ROW EXCLUSIVE** is the same as ROW SHARE, but also prohibits locking in SHARE mode. Row Exclusive locks are automatically obtained when updating, inserting, or deleting. **SHARE UPDATE**—see ROW SHARE. **SHARE** allows concurrent queries but prohibits updates to the locked table. **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 updating rows. **EXCLUSIVE** allows queries on the locked table but prohibits any other activity on it.
NOWAIT	specifies that Oracle returns control to you immediately if the specified table is already locked by another user. In this case, Oracle returns a message indicating that the table is already locked by another user. If you omit this clause, Oracle waits until the table is available, locks it, and returns control to you.

Locking Tables

Some forms of locks can be placed on the same table at the same time; other locks only allow one lock per table. For example, multiple users can place SHARE locks on the same table at the same time, but only one user can place an EXCLUSIVE lock on a table at a time. For a complete description of the interaction of lock modes, see *Oracle8 Concepts*.

When you lock a table, you choose how other users can access it. 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.

Example I. The following statement locks the EMP table in exclusive mode, but does not wait if another user already has locked the table:

```
LOCK TABLE emp
IN EXCLUSIVE MODE
NOWAIT;
```

Example II. The following statement locks the remote ACCOUNTS table that is accessible through the database link BOSTON:

```
LOCK TABLE accounts@boston
IN SHARE MODE;
```

Related Topics

DELETE on page 4-374
INSERT on page 4-451
UPDATE on page 4-542
COMMIT on page 4-185
ROLLBACK on page 4-484
SAVEPOINT on page 4-487

NOAUDIT (SQL Statements)

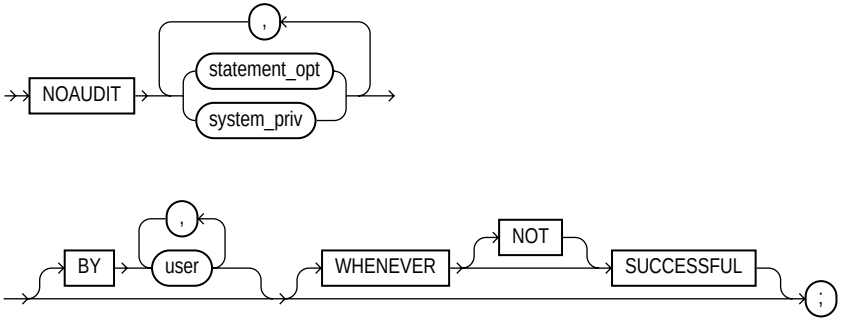
Purpose

To stop auditing previously enabled by the AUDIT command (SQL Statements). To stop auditing enabled by the AUDIT command (Schema Objects), refer to NOAUDIT (Schema Objects) on page 4-463. See also "Stopping Auditing" on page 4-462.

Prerequisites

You must have AUDIT SYSTEM system privilege.

Syntax



Keywords and Parameters

statement_opt	is a statement option for which auditing is stopped. For a list of the statement options and the SQL statements they audit, see Table 4–6 on page 4-172 and Table 4–7 on page 4-174.
system_priv	is a system privilege for which auditing is stopped. For a list of the system privileges and the statements they authorize, see Table 4–6 on page 4-172.
BY *user*	stops auditing only for SQL statements issued by specified users in their subsequent sessions. If you omit this clause, Oracle stops auditing for all users' statements, except for the situation described in the section that follows.
WHENEVER SUCCESSFUL	stops auditing only for SQL statements that complete successfully.
NOT	stops auditing only for statements that result in Oracle errors.

If you omit the WHENEVER clause entirely, Oracle stops auditing for all statements, regardless of success or failure.

Stopping Auditing

The NOAUDIT statement must have the same syntax as the previous AUDIT statement. Further, it reverses the effects only of that particular statement. Therefore, if one AUDIT statement (statement A) enables auditing for a specific user, and a second (statement B) enables auditing for all users, then a NOAUDIT statement to disable auditing for all users (statement C) reverses statement B, but leaves statement A in effect and continues to audit the user that statement A specified. For information on auditing specific SQL commands, see the AUDIT (SQL Statements) on page 4-170.

The following examples correspond to the first three examples listed in AUDIT (SQL Statements) on page 4-170.

Example I. If you have chosen auditing for every SQL statement that creates or drops a role, you can stop auditing of such statements by issuing the following statement:

```
NOAUDIT ROLE;
```

Example II. If you have chosen auditing for any statement that queries or updates any table issued by the users SCOTT and BLAKE, you can stop auditing for SCOTT's queries by issuing the following statement:

```
NOAUDIT SELECT TABLE
   BY scott;
```

The above statement stops auditing only SCOTT's queries, so Oracle continues to audit BLAKE's queries and updates and SCOTT's updates.

Example III. To stop auditing on all statements that are authorized by DELETE ANY TABLE system privilege, issue the following statement:

```
NOAUDIT DELETE ANY TABLE;
```

Related Topics

NOAUDIT (Schema Objects) on page 4-463

NOAUDIT (Schema Objects)

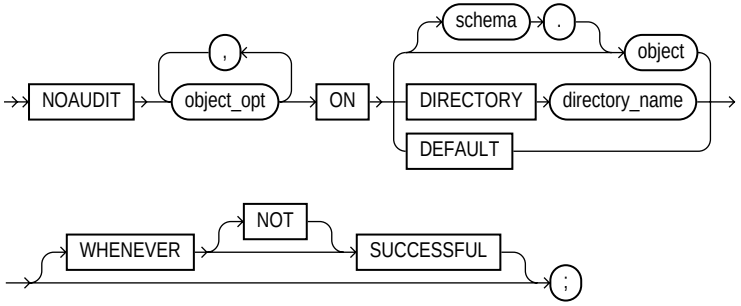
Purpose

To stop auditing previously enabled by the AUDIT command (Schema Objects). To stop auditing enabled by the AUDIT command (SQL Statements), refer to NOAUDIT (SQL Statements) on page 4-461. For illustrations, see "Examples" on page 4-455.

Prerequisites

The object on which you stop 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, even if you created it, you must have AUDIT ANY system privilege.

Syntax



Keywords and Parameters

object_opt stops auditing for particular operations on the object. For a list of these options, see Table 4–8 on page 4-180.

ON identifies the object on which auditing is stopped. If you do not qualify object with *schema*, Oracle assumes the object is in your own schema.

object identifies the object on which auditing is stopped. The object must a table; view; sequence; stored procedure, function, or package; snapshot; or library.

For information on auditing specific schema objects, refer to AUDIT (Schema Objects) on page 4-178.

DIRECTORY *directory_name*	identifies the name of the directory on which auditing is being stopped.
DEFAULT	removes the specified object options as default object options for subsequently created objects.
WHENEVER SUCCESSFUL	stops auditing only for SQL statements that complete successfully.
	NOT stops auditing only for statements that result in Oracle errors.
	If you omit the WHENEVER clause entirely, Oracle stops auditing for all statements, regardless of success or failure.

Examples

If you have chosen auditing for every SQL statement that queries the EMP table in the schema SCOTT, you can stop auditing for such queries by issuing the following statement:

```
NOAUDIT SELECT
   ON scott.emp;
```

You can stop auditing for such queries that complete successfully by issuing the following statement:

```
NOAUDIT SELECT
   ON scott.emp
   WHENEVER SUCCESSFUL;
```

This statement stops auditing only for successful queries, Oracle continues to audit queries resulting in Oracle errors.

Related Topics

AUDIT (Schema Objects) on page 4-178
NOAUDIT (SQL Statements) on page 4-461

PARALLEL clause

Purpose

To specify whether Oracle should execute an operation serially or in parallel. See also "Using the PARALLEL Clause" on page 4-466. For illustrations, see "Examples" on page 4-467.

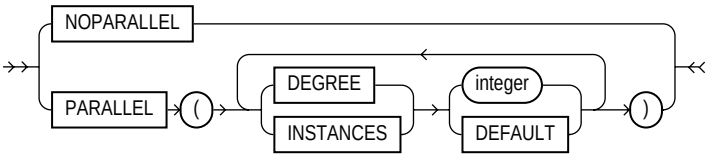
Prerequisites

This clause can only be used in the following commands:

- ALTER CLUSTER
- ALTER DATABASE ... RECOVER
- ALTER INDEX ... REBUILD
- ALTER TABLE
- CREATE CLUSTER
- CREATE INDEX
- CREATE TABLE

> **Note:** The PARALLEL clause syntax is allowed when creating a table, index, or cluster in a CREATE SCHEMA statement. However, parallelism is not used and no error message is issued.

Syntax



Keywords and Parameters

NOPARALLEL	specifies serial execution of an operation. This is the default.
PARALLEL	specifies parallel execution of an operation.

DEGREE	determines the degree of parallelism for an operation on a single instance—that is, the number of query servers used in the parallel operation.	
	integer	uses *integer* query servers.
	DEFAULT	the default number of query servers used is calculated from the number of CPUs and the number of DEVICES storing tables to be scanned in parallel.
INSTANCES	determines the number of parallel server instances used in the parallel operation. This keyword is ignored if you do not have a parallel server.	
	integer	uses *integer* instances
	DEFAULT	uses all available instances
		Note: INSTANCES only applies to an instance using the Oracle Parallel Server.
		See also "Nonpartitioned Tables and Indexes" on page 4-466 and "Partitioned Tables and Indexes" on page 4-467.

Using the PARALLEL Clause

Use the PARALLEL clause to specify table parallelism in the CREATE TABLE and ALTER TABLE commands. When you specify this clause in a table definition, Oracle uses the clause to determine parallelism of DML statements as well as queries. Explicit parallel hints, however, override the effect of the PARALLEL clauses for that table.

If you do not specify the PARALLEL clause, Oracle determines the type of parallelism to use by the default PARALLEL attributes of the table or index.

For more information on parallelized operations, see *Oracle8 Tuning*, *Oracle8 Concepts*, and *Oracle8 Parallel Server Concepts and Administration*.

Nonpartitioned Tables and Indexes

Used in a CREATE command, the PARALLEL clause causes the creation of the schema object to be parallelized. If the CREATE command is CREATE TABLE, the PARALLEL clause also sets the default degree of parallelism for queries and DML on the table after creation.

Used in a command to alter an object, the PARALLEL clause changes the default degree of parallelism for queries and DML on the object. In an ALTER DATABASE RECOVER command, the PARALLEL clause causes the recovery to be parallelized.

Specifying PARALLEL (DEGREE 1 INSTANCES 1) is equivalent to specifying NOPARALLEL.

A hint in a query can override a default of NOPARALLEL. Likewise, a hint in a query can override a default of PARALLEL.

Partitioned Tables and Indexes

The INSTANCES parameter of CREATE TABLE ... AS SELECT and CREATE INDEX determines the number of instances used by the CREATE operation. Instances are chosen for physical affinity to the (first) datafiles underlying the partitions. If the INSTANCES parameter is greater than the number of instances with affinity to the underlying datafiles, additional instances (up to the total number of partitions) are chosen arbitrarily. The DEGREE and INSTANCES parameters, stored in the data dictionary, are used later to compute the default PARALLEL attributes of the schema object.

Examples

Example I. The following command creates a table using 10 query servers, 5 to scan SCOTT.EMP and another 5 to populate EMP_DEPT:

```
CREATE TABLE emp_dept
PARALLEL (DEGREE 5)
AS SELECT * FROM scott.emp
WHERE deptno = 10;
```

Example II. The following command creates an index using 10 query servers, 5 to scan SCOTT.EMP and another 5 to populate the EMP_IDX index:

```
CREATE INDEX emp_idx
ON scott.emp (ename)
PARALLEL 5;
```

Example III. The following command performs tablespace recovery using 5 recovery processes on 5 instances in a parallel server, for a total of 25 (5 * 5) query servers:

```
ALTER DATABASE
RECOVER TABLESPACE binky
PARALLEL (DEGREE 5 INSTANCES 5);
```

Example IV. The following command changes the default number of query servers used to query the EMP table:

```
ALTER TABLE emp
PARALLEL (DEGREE 9);
```

Example V. The following command causes the index to be rebuilt from the existing index by using 6 query servers, 3 each to scan the old and to build the new index:

```
ALTER INDEX emp_idx
REBUILD
PARALLEL 3;
```

Related Topics

ALTER CLUSTER on page 4-11
ALTER DATABASE on page 4-15
ALTER INDEX on page 4-28
ALTER TABLE on page 4-106
CREATE CLUSTER on page 4-207
CREATE INDEX on page 4-237
CREATE TABLE on page 4-306
Oracle8 Tuning
Oracle8 Parallel Server Concepts and Administration

RECOVER clause

Purpose

To perform media recovery.

Use the ALTER DATABASE command with the RECOVER clause if you want to write your own specialized media recovery application using SQL. For other situations, Oracle recommends that you use the Server Manager RECOVER command rather than the ALTER DATABASE command with the RECOVER clause to perform media recovery.

For more information on media recovery, see the *Oracle8 Backup and Recovery Guide* and *Oracle8 Administrator's Guide*. For illustrations, see "Examples" on page 4-471.

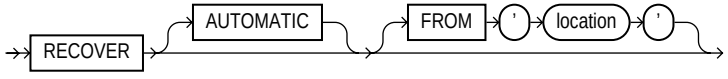
Prerequisites

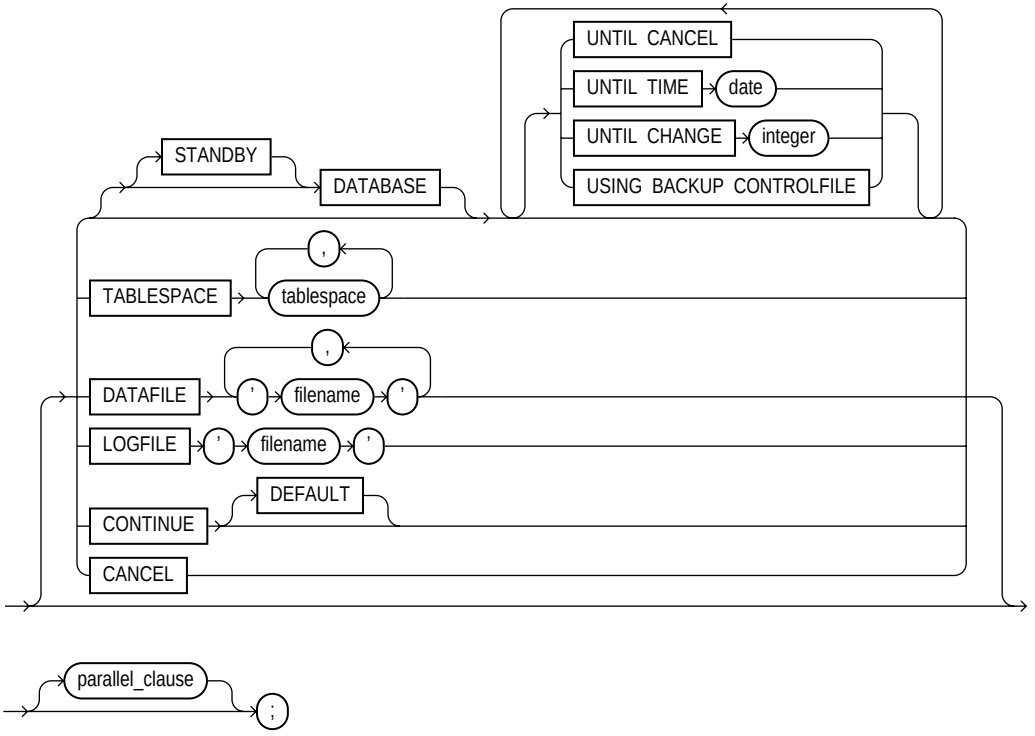
The RECOVER clause must appear in an ALTER DATABASE statement. You must have the privileges necessary to issue this statement. For information on these privileges, see ALTER DATABASE on page 4-15.

In addition:

- You must also have the OSDBA role enabled.
- You cannot be connected to Oracle through the multithreaded server architecture.
- Your instance must have the database mounted in exclusive mode.

Syntax





parallel_clause: See the PARALLEL clause on page 4-465.

Keywords and Parameters

AUTOMATIC	automatically generates the names of the redo log files to apply during media recovery. If you omit this option, then you must specify the names of redo log files using the ALTER DATABASE ... RECOVER command with the LOGFILE clause.
FROM	specifies 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, Oracle assumes the archived redo log file group is in the location specified by the initialization parameter LOG_ARCHIVE_DEST.
STANDBY	recovers the standby database using the control file and archived redo log files copied from the primary database. For more information, see *Oracle8 Administrator's Guide*.

DATABASE	recovers the entire database. This is the default option. You can use this option only when the database is closed. **Note:** This option will recover only online datafiles.
UNTIL CANCEL	performs cancel-based recovery. This option recovers the database until you issue the ALTER DATABASE RECOVER command with the CANCEL clause.
UNTIL TIME	performs 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'.
UNTIL CHANGE	performs change-based recovery. This parameter recovers the database to a transaction-consistent state immediately before the system change number (SCN) specified by *integer*.
USING BACKUP CONTROLFILE	specifies that a backup control file is being used instead of the current control file.
TABLESPACE	recovers only the specified tablespaces. You can use this option if the database is open or closed, provided the tablespaces to be recovered are offline.
DATAFILE	recovers the specified datafiles. You can use this option when the database is open or closed, provided the datafiles to be recovered are offline.
LOGFILE	continues media recovery by applying the specified redo log file.
CONTINUE	continues multi-instance recovery after it has been interrupted to disable a thread.
CONTINUE DEFAULT	continues recovery by applying the redo log file that Oracle has automatically generated.
CANCEL	terminates cancel-based recovery.
parallel_clause	specifies degree of parallelism to use when recovering. See the PARALLEL clause on page 4-465.

Examples

Example I . The following statement performs complete recovery of the entire database:

```
ALTER DATABASE
  RECOVER AUTOMATIC DATABASE;
```

Oracle automatically generates the names of redo log files to apply and prompts you with them. The following statement applies a suggested file:

```
ALTER DATABASE
    RECOVER CONTINUE DEFAULT;
```

The following statement explicitly names a redo log file for Oracle to apply:

```
ALTER DATABASE
    RECOVER LOGFILE 'diska:arch0006.arc';
```

Example II. The following statement performs time-based recovery of the database:

```
ALTER DATABASE AUTOMATIC
    RECOVER UNTIL TIME '1992-10-27:14:00:00';
```

Oracle recovers the database until 2:00 pm on October 27, 1992.

Example III. The following statement recovers the tablespace USER5:

```
ALTER DATABASE
    RECOVER TABLESPACE user5;
```

Related Topics

ALTER DATABASE on page 4-15

RENAME

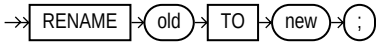
Purpose

To rename a table, view, sequence, or private synonym for a table, view, or sequence. See also “Renaming Objects” on page 4-473.

Prerequisites

The object must be in your own schema. See also “Restrictions” on page 4-473.

Syntax



Keywords and Parameters

old	is the name of an existing table, view, sequence, or private synonym.
new	is 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 defined in the section “Schema Object Naming Rules” on page 2-47.

Renaming Objects

Integrity constraints, indexes, and grants on the old object are automatically transferred to the new object. Oracle invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

To change the name of table DEPT to EMP_DEPT, issue the following statement:

```
RENAME dept TO emp_dept;
```

Restrictions

You cannot use this command to rename public synonyms. To rename a public synonym, you must first drop it with the DROP SYNONYM command and then create another public synonym with the new name using the CREATE SYNONYM command.

You cannot use this command to rename columns. You can rename a column using the CREATE TABLE command with the AS clause. For example, the following statement re-creates the table STATIC, renaming a column from OLDNAME to NEWNAME:

```
CREATE TABLE temporary (newname, col2, col3)
    AS SELECT oldname, col2, col3 FROM static
DROP TABLE static
RENAME temporary TO static;
```

Related Topics

CREATE SEQUENCE on page 4-281
CREATE SYNONYM on page 4-302
CREATE TABLE on page 4-306
CREATE VIEW on page 4-363

REVOKE (System Privileges and Roles)

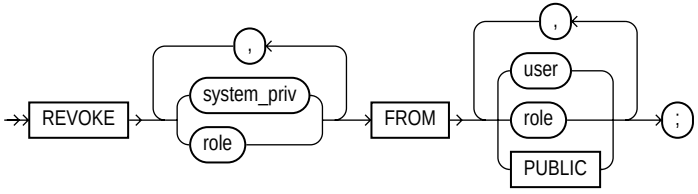
Purpose

To revoke system privileges and roles from users and roles. To revoke object privileges from users and roles, refer to REVOKE (Schema Object Privileges) on page 4-478. For illustrations, see "Examples" on page 4-477.

Prerequisites

You must have been granted the system privilege or role with the ADMIN OPTION. Also, you can revoke any role if you have the GRANT ANY ROLE system privilege. See also "Limitations" on page 4-476.

Syntax



Keywords and Parameters

system_priv	is a system privilege to be revoked. For a list of the system privileges, see Table 4–19 on page -993. See also "Revoking Privileges" on page 4-475.
role	is a role to be revoked. For a list of the roles predefined by Oracle, see *Oracle8 Administrator's Guide*. See also "Revoking Roles" on page 4-476.
FROM	identifies users and roles from which the system privileges or roles are to be revoked.
PUBLIC	revokes the system privilege or role from all users.

Revoking Privileges

If you revoke **a privilege from a user,** Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

If you revoke **a privilege from a role**, Oracle removes the privilege from the role's privilege domain. 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.

If you revoke **a privilege from PUBLIC,** Oracle 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.

Revoking Roles

If you revoke **a role from a user,** Oracle makes the role unavailable to the user. If the role is currently enabled for the user, 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,** Oracle removes the revoked role's privilege domain from the revokee role's privilege domain. Users who have been granted and have enabled the revokee role can continue to exercise the privileges in the revoked role's privilege domain 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.

If you revoke **a role from PUBLIC,** Oracle 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. Note that the role is not revoked from users who have been granted the privilege directly or through other roles.

Limitations

The REVOKE command can revoke only privileges and roles that were previously granted directly with a GRANT statement. The REVOKE command cannot perform the following operations:

- revoke privileges or roles not granted to the revokee
- revoke roles granted through the operating system
- revoke privileges or roles granted to the revokee through roles

A system privilege or role cannot appear more than once in the list of privileges and roles to be revoked. A user, a role, or PUBLIC cannot appear more than once in the FROM clause.

Examples

Example I. The following statement revokes DROP ANY TABLE system privilege from the users BILL and MARY:

```
REVOKE DROP ANY TABLE
    FROM bill, mary;
```

BILL and MARY can no longer drop tables in schemas other than their own.

Example II. The following statement revokes the role CONTROLLER from the user HANSON:

```
REVOKE controller
    FROM hanson;
```

HANSON can no longer enable the CONTROLLER role.

Example III. The following statement revokes the CREATE TABLESPACE system privilege from the CONTROLLER role:

```
REVOKE CREATE TABLESPACE
   FROM controller;
```

Enabling the CONTROLLER role no longer allows users to create tablespaces.

Example IV. To revoke the role VP from the role CEO, issue the following statement:

```
REVOKE vp
  FROM ceo;
```

VP is no longer granted to CEO.

Example V. To revoke the CREATE ANY DIRECTORY system privilege from user SCOTT, issue the following statement:

```
REVOKE CREATE ANY DIRECTORY FROM scott;
```

Related Topics

GRANT (System Privileges and Roles) on page -991
REVOKE (Schema Object Privileges) on page 4-478

REVOKE (Schema Object Privileges)

Purpose

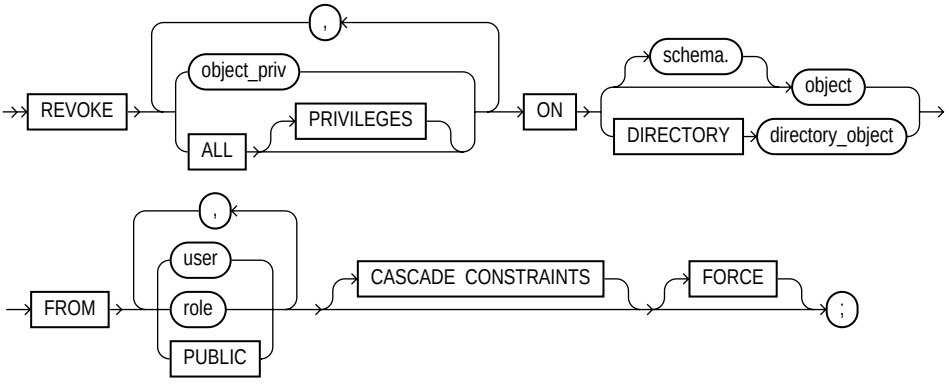
To revoke object privileges for a particular object from users and roles. To revoke system privileges or roles, refer to . REVOKE (System Privileges and Roles) on page 4-475. See also "Revoking Object Privileges" on page 4-480.

Each object privilege authorizes some operation on an object. By revoking an object privilege, you prevent the revokee from performing that operation. For a summary of the object privileges for each type of object, see Table 4–13 on page 4-447. For illustrations, see "Examples" on page 4-481.

Prerequisites

You must have previously granted the object privileges to each user and role. See also "Revoking Multiple Identical Grants" on page 4-480.

Syntax



Keywords and Parameters

object_priv	is an object privilege to be revoked. You can substitute any of the following values: ALTER DELETE EXECUTE INDEX INSERT READ REFERENCES SELECT UPDATE
ALL PRIVILEGES	revokes all object privileges that you have granted to the revokee. **Note:** If *no* privileges have been granted on the object, Oracle takes no action and does not return an error message.
ON DIRECTORY *directory_object*	identifies a directory object on which privileges are revoked. You cannot qualify *directory_object* with *schema* when using the DIRECTORY option. The object must be a directory. See CREATE DIRECTORY on page 4-230.
ON *object*	identifies the object on which the object privileges are revoked. This object can be a table; view; sequence; procedure, stored function, or package; snapshot; synonym for a table, view, sequence, procedure, stored function, package, or snapshot; or library. If you do not qualify *object* with *schema*, Oracle assumes the object is in your own schema.
FROM	identifies users and roles from which the object privileges are revoked. PUBLIC revokes object privileges from all users.
CASCADE CONSTRAINTS	drops any referential integrity constraints that the revokee has defined using the REFERENCES privilege or the ALL PRIVILEGES option if the revokee has exercised the REFERENCES privilege to define a referential integrity constraints. See also "Cascading Revokes" on page 4-481.
FORCE	revokes EXECUTE object privileges on user-defined type objects with table dependencies. You must use the FORCE option to revoke the EXECUTE object privilege on user-defined type objects with table dependencies. See also "Using FORCE" on page 4-480. For detailed information about type dependencies and user-defined object privileges, see *Oracle8 Concepts*.

Revoking Object Privileges

If you revoke **a privilege from a user,** Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

If you revoke **a privilege from a role,** Oracle removes the privilege from the role's privilege domain. 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 **a privilege from PUBLIC,** Oracle 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.

You can use the REVOKE command only to revoke object privileges that you previously granted directly to the revokee. You cannot use the REVOKE command to perform the following operations:

- revoke object privileges that you did not grant to the revokee
- revoke privileges granted through the operating system
- revoke privileges granted to roles granted to the revokee

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.

Using FORCE

You must use the FORCE option to revoke the EXECUTE object privilege on user-defined type objects with table dependencies. The FORCE option causes the data in the dependent tables to become inaccessible. Regranting the necessary type privilege will revalidate the table. For detailed information about type dependencies and user-defined object privileges, see *Oracle8 Concepts*.

Revoking Multiple Identical Grants

Multiple users may grant the same object 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, the grantee can still exercise the privilege by virtue of that grant.

Cascading Revokes

Revoking an object privilege that a user has either granted or exercised to define an object or a referential integrity constraint has the following cascading effects:

- If you revoke an object privilege from a user who has granted the privilege to other users or roles, Oracle also revokes the privilege from the grantees.
- If you revoke an object privilege from a user whose schema contains a procedure, function, or package that contains SQL statements that exercise the privilege, the procedure, function, or package can no longer be executed.
- If you revoke an object privilege on an object from a user whose schema contains a view on that object, Oracle invalidates the view.
- If you revoke the REFERENCES privilege from a user who has exercised the privilege to define referential integrity constraints, you must specify the CASCADE CONSTRAINTS option. Oracle then revokes the privilege and drops the constraints.

Examples

Example I. You can grant DELETE, INSERT, SELECT, and UPDATE privileges on the table BONUS to the user PEDRO with the following statement:

```
GRANT ALL
    ON bonus TO pedro;
```

To revoke the DELETE privilege on BONUS from PEDRO, issue the following statement:

```
REVOKE DELETE
    ON bonus FROM pedro;
```

To revoke the remaining privileges on BONUS that you granted to PEDRO, issue the following statement:

```
REVOKE ALL
    ON bonus FROM pedro;
```

Example II. You can grant SELECT and UPDATE privileges on the view REPORTS to all users by granting the privileges to the role PUBLIC:

```
GRANT SELECT, UPDATE
    ON reports TO public;
```

The following statement revokes UPDATE privilege on REPORTS from all users:

```
REVOKE UPDATE
    ON reports FROM public;
```

Users can no longer update the REPORTS view, although users can still query it. However, if you have also granted UPDATE privilege on REPORTS to any users (either directly or through roles), these users retain the privilege.

Example III. You can grant the user BLAKE the SELECT privilege on the ESEQ sequence in the schema ELLY with the following statement:

```
GRANT SELECT
    ON elly.eseq TO blake;
```

To revoke the SELECT privilege on ESEQ from BLAKE, issue the following statement:

```
REVOKE SELECT
    ON elly.eseq FROM blake;
```

However, if the user ELLY has also granted SELECT privilege on ESEQ to BLAKE, BLAKE can still use ESEQ by virtue of ELLY's grant.

Example IV. You can grant BLAKE the privileges REFERENCES and UPDATE on the EMP table in the schema SCOTT with the following statement:

```
GRANT REFERENCES, UPDATE
    ON scott.emp TO blake;
```

BLAKE can exercise the REFERENCES privilege to define a constraint in his own DEPENDENT table that refers to the EMP table in the schema SCOTT:

```
CREATE TABLE dependent
(dependno NUMBER,
 dependname VARCHAR2(10),
 employee NUMBER
    CONSTRAINT in_emp REFERENCES scott.emp(ename) );
```

You can revoke the REFERENCES privilege on SCOTT.EMP from BLAKE, by issuing the following statement that contains the CASCADE CONSTRAINTS option:

```
REVOKE REFERENCES
    ON scott.emp
    FROM blake
    CASCADE CONSTRAINTS;
```

Revoking BLAKE's REFERENCES privilege on SCOTT.EMP causes Oracle to drop the IN_EMP constraint, because BLAKE required the privilege to define the constraint.

However, if BLAKE has also been granted the REFERENCES privilege on SCOTT.EMP by a user other than you, Oracle does not drop the constraint. BLAKE still has the privilege necessary for the constraint by virtue of the other user's grant.

Example V. You can revoke READ privilege on directory BFILE_DIR1 from SUE, by issuing the following statement:

```
REVOKE READ ON DIRECTORY bfile_dir1 FROM sue;
```

Related Topics

GRANT (Object Privileges) on page 4-444
REVOKE (System Privileges and Roles) on page 4-475

ROLLBACK

Purpose

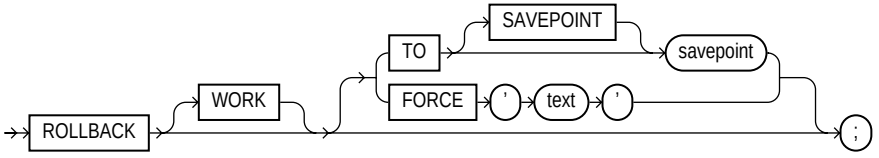
To undo work done in the current transaction, or to manually undo the work done by an in-doubt distributed transaction. See also “Rolling Back Transactions” on page 4-484.

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 FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Syntax



Keywords and Parameters

WORK	is optional and is provided for ANSI compatibility.
TO	rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.
FORCE	manually rolls back an in-doubt distributed transaction. The transaction is identified by the ’*text*’ containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. See also “Distributed Transactions” on page 4-486. ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

Rolling Back Transactions

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle treats as a single unit. A transaction begins with the first executable SQL statement

after a COMMIT, ROLLBACK, or connection to the database. A transaction ends with a COMMIT statement, a ROLLBACK statement, or disconnection (intentional or unintentional) from the database.

> **Note:** Oracle issues an implicit COMMIT statement before and after processing any data definition language (DDL) statement.

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 the transaction's locks

Using ROLLBACK with the TO SAVEPOINT clause performs the following operations:

- rolls back just the portion of the transaction after the savepoint.
- erases all savepoints created after that savepoint. Note that 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. Note that 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.

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, Oracle rolls back the last uncommitted transaction.

Example I. The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

Example II. The following statement rolls back your current transaction to savepoint SP5:

```
ROLLBACK TO SAVEPOINT sp5;
```

Distributed Transactions

Oracle's distributed functionality enables you to perform distributed transactions, or transactions that modify data on multiple databases. To commit or roll back a distributed transaction, you need only issue a COMMIT or ROLLBACK statement as you would any other transaction.

If a network failure occurs during the commit process for a distributed transaction, the state of the transaction may be unknown, or *in doubt*. After consultation with the administrators of the other databases involved in the transaction, you may decide to manually commit or roll back the transaction on your local database. You can manually roll back the transaction on your local database by issuing a ROLLBACK statement with the FORCE clause.

For more information on when to roll back in-doubt transactions, see *Oracle8 Distributed Database Systems*.

You cannot manually roll back an in-doubt transaction to a savepoint.

A ROLLBACK statement with a FORCE clause only rolls back the specified transaction. Such a statement does not affect your current transaction.

Example. The following statement manually rolls back an in-doubt distributed transaction:

```
ROLLBACK WORK
    FORCE '25.32.87';
```

Related Topics

COMMIT on page 4-185
SAVEPOINT on page 4-487
SET TRANSACTION on page 4-519

SAVEPOINT

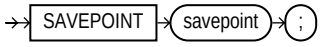
Purpose

To identify a point in a transaction to which you can later roll back. See also "Creating Savepoints" on page 4-487.

Prerequisites

None.

Syntax



Keywords and Parameters

savepoint	is the name of the savepoint to be created.

Creating Savepoints

Savepoints are used with the ROLLBACK command to roll back portions of the current transaction. For more information, see "Rolling Back Transactions" on page 4-484.

Savepoints are useful in interactive programs, because you can create and name intermediate steps of a program. This allows you more control over longer, more complex programs. For example, you can use savepoints throughout a long complex series of updates, so that if you make an error, you need not resubmit every statement.

Savepoints are similarly useful in application programs: if a program contains several subprograms, you can create a savepoint before each subprogram begins. If a subprogram fails, you can easily return the data to its state before the subprogram began and then reexecute the subprogram with revised parameters or perform a recovery action.

Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, 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.

Example. To update BLAKE's and CLARK's salary, check that the total company salary does not exceed 20,000, then reenter CLARK's salary, enter:

```
UPDATE emp
    SET sal = 2000
    WHERE ename = 'BLAKE'
SAVEPOINT blake_sal
UPDATE emp
    SET sal = 1500
    WHERE ename = 'CLARK'
SAVEPOINT clark_sal
SELECT SUM(sal) FROM emp
ROLLBACK TO SAVEPOINT blake_sal
UPDATE emp
    SET sal = 1300
    WHERE ename = 'CLARK'
COMMIT;
```

Related Topics

COMMIT on page 4-185
ROLLBACK on page 4-484
SET TRANSACTION on page 4-519

SELECT

Purpose

To retrieve data from one or more tables, object tables, views, object views, or snapshots.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

Prerequisites

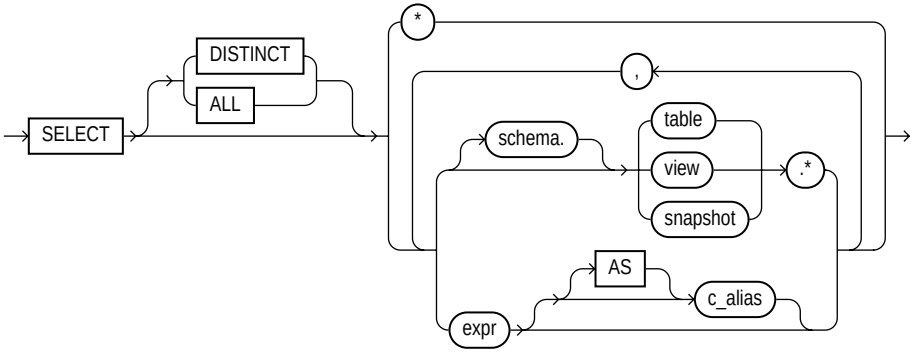
For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

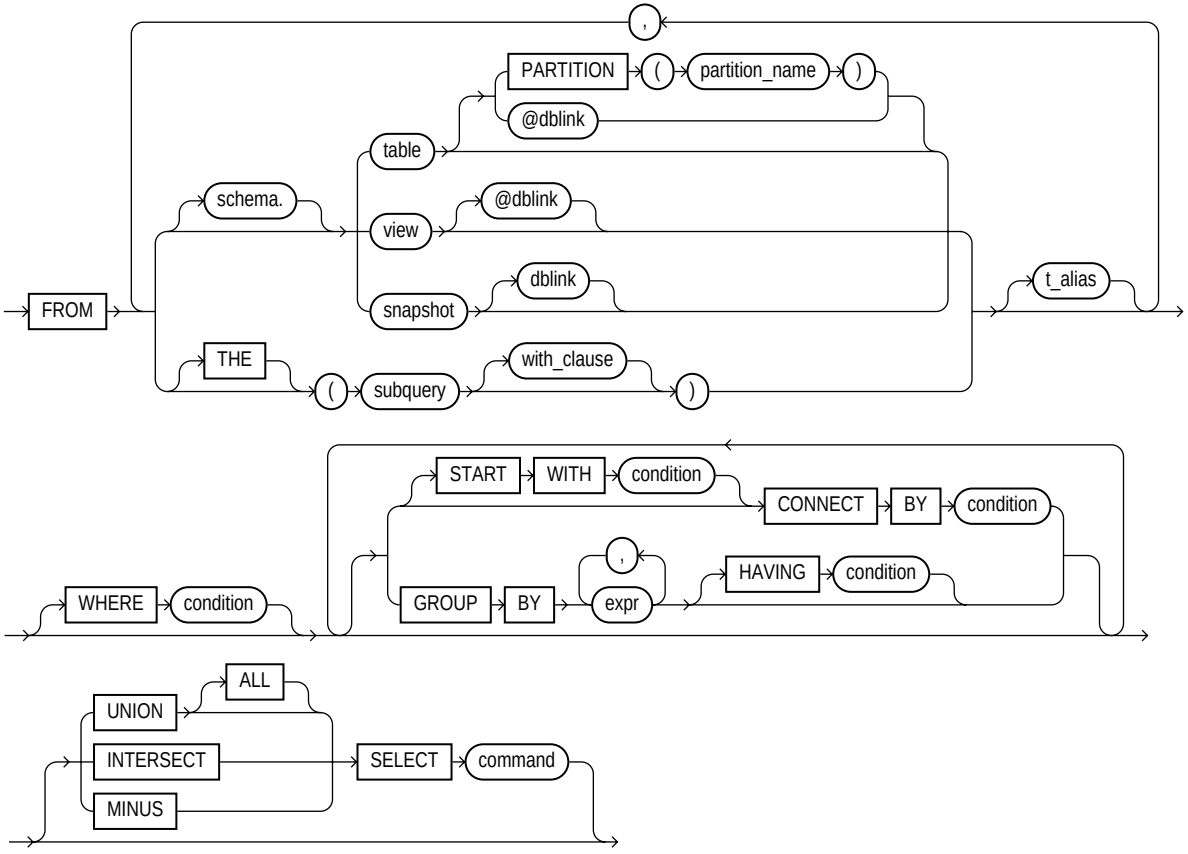
For you to select rows from the base tables of a view,

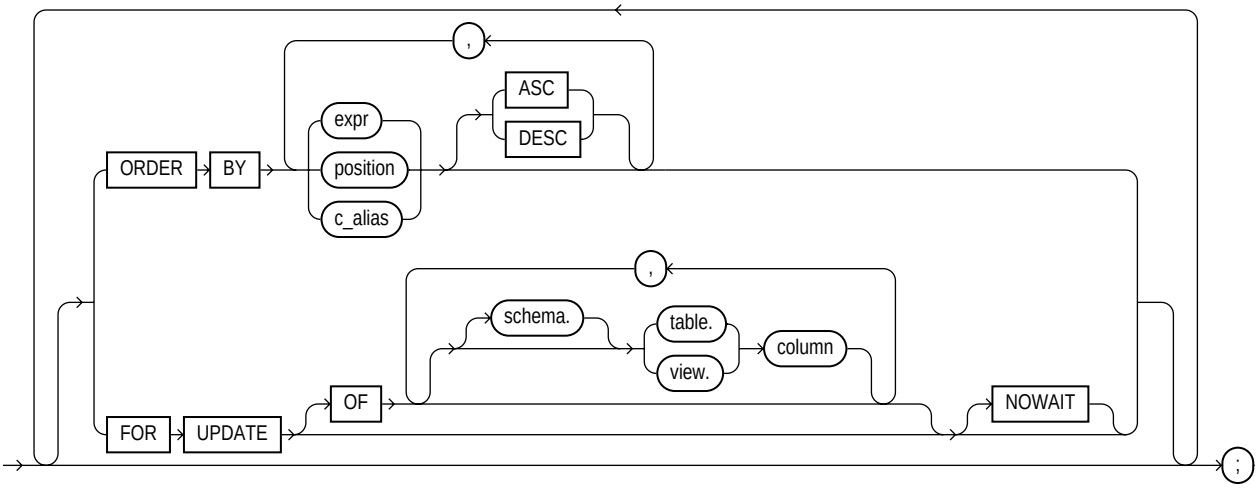
- You must have SELECT privilege on the view, and
- Whoever owns the schema containing the view must have SELECT privilege on the base tables.

The SELECT ANY TABLE system privilege also allows you to select data from any table or any snapshot or any view's base table.

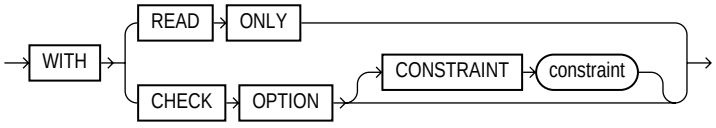
Syntax




,
PARTITION
(
partition_name
)
@dblink
table
schema.
@dblink
view
snapshot
dblink
FROM
t_alias
THE
(
subquery
with_clause
)
START
WITH
condition
CONNECT
BY
condition
WHERE
condition
GROUP
BY
,
expr
HAVING
condition
ALL
UNION
INTERSECT
MINUS
SELECT
command



WITH_clause::=



Keywords and Parameters

DISTINCT	returns only one copy of each set of duplicate rows selected. Duplicate rows are those with matching values for each expression in the select list.
ALL	returns all rows selected, including all copies of duplicates. The default is ALL.
*	selects all columns from all tables, views, or snapshots, listed in the FROM clause.
*table.** *view.** *snapshot.**	selects all columns from the specified table, view, or snapshot. You can use the schema qualifier to select from a table, view, or snapshot in a schema other than your own. See also "Joins" on page 4-505.
expr	selects an expression. See the syntax description of *expr* in "Expressions" on page 3-78; see also "Creating Simple Queries" on page 4-493. A column name in this list can be qualified only with schema if the table, view, or snapshot containing the column is qualified with *schema* in the FROM clause.

c_alias	provides a different name for the column expression and causes the alias to be used in the column heading. 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.
PARTITION (*partition_name)*	specifies partition-level data retrieval. The *partition_name* parameter may be the name of the partition within table from which to retrieve data or a more complicated predicate restricting retrieval to just one partition of the table.
schema	is the schema containing the selected table, view, or snapshot. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.
table, view, snapshot	is the name of a table, view, or snapshot from which data is selected.
dblink	is the complete or partial name for a database link to a remote database where the table, view, or snapshot is located. For more information on referring to database links, see the section "Referring to Objects in Remote Databases" on page 2-54. Note that this database need not be an Oracle database. If you omit *dblink*, Oracle assumes that the table, view, or snapshot is on the local database. If you apply the keyword THE, the subquery must return a single column value which must be a nested table or an expression that yields a nested table.
OBJ THE	informs Oracle that the column value returned by the subquery is a nested table, not a scalar value. A subquery prefixed by THE is called a *flattened subquery*. "Using Flattened Subqueries" on page 4-533. Note: You cannot use the set operators in a flattened subquery; see *set operators* below.
subquery	is a subquery that is treated in the same manner as a view. "Subqueries" on page 4-530. Oracle executes the subquery and then uses the resulting rows as a view in the FROM clause.
t_alias	provides a different name for the table, view, snapshot, or subquery for evaluating the query and is most often used in a correlated query. Other references to the table, view, or snapshot throughout the query must refer to the alias.
WHERE	restricts the rows selected to those for which the *condition* is TRUE. If you omit this clause, Oracle returns all rows from the tables, views, or snapshots in the FROM clause. See the syntax description of condition in "Conditions" on page 3-90.
START WITH ... CONNECT BY	returns rows in a hierarchical order. See also "Hierarchical Queries" on page 4-495.
GROUP BY	groups the selected rows based on the value of expr for each row, and returns a single row of summary information for each group. See also "GROUP BY Clause" on page 4-499.

HAVING	restricts the groups of rows returned to those groups for which the specified *condition* is TRUE. If you omit this clause, Oracle returns summary rows for all groups. See also "HAVING Clause" on page 4-500. See also the syntax description of *expr* in "Expressions" on page 3-78 and the syntax description of *condition* in "Conditions" on page 3-90.
set operators: UNION UNION ALL INTERSECT MINUS	combine the rows returned by two SELECT statements using a set operation. To reference a column, you must use an alias to name the column. The FOR UPDATE clause cannot be used with these set operators. OBJ SELECT statements using THE or MULTISET keywords cannot be used with these set operators. See also "UNION, UNION ALL, INTERSECT, and MINUS" on page 4-501.
ORDER BY	orders rows returned by the statement. *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 snapshots in the FROM clause. *position* orders rows based on their value for the expression in this position of the select list. ASC and DESC specify either ascending or descending order. ASC is the default. See also "ORDER BY Clause" on page 4-501.
FOR UPDATE	locks the selected rows. OF — Locks the select rows only for a particular table in a join. NOWAIT — returns control to you if the SELECT statement attempts to lock a row that is locked by another user. If you omit this clause, Oracle waits until the row is available and then returns the results of the SELECT statement. See also "FOR UPDATE Clause" on page 4-503.

Creating Simple Queries

The list of expressions that appears after the SELECT keyword and before the FROM clause is called the *select list*. Each expression *expr* becomes the name of one column in the set of returned rows, and each *table*.* becomes a set of columns, one for each column in the table in the order they were defined when the table was created. The datatype and length of each expression is determined by the elements of the expression.

If two or more tables have some column names in common, you must qualify column names with names of tables. Otherwise, fully qualified column names are optional, although it is always better to explicitly qualify table and column

references. Oracle often does less work with fully qualified table and column names.

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 NOV96 partition of the SALES table:

```
SELECT * FROM sales PARTITION (nov96) s
  WHERE s.amount_of_sale > 1000;
```

You can use a column alias, *c_alias*, to label the 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.

If you use the DISTINCT option to return only a single copy of duplicate rows, 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 can use comments in a SELECT statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see *Oracle8 Tuning*.

Example I. The following statement selects rows from the EMP table with the department number of 30:

```
SELECT *
    FROM emp
    WHERE deptno = 30;
```

Example II. The following statement selects the name, job, salary and department number of all employees except sales people from department number 30:

```
SELECT ename, job, sal, deptno
    FROM emp
    WHERE NOT (job = 'SALESMAN' AND deptno = 30);
```

Example III. The following statement selects from subqueries in the FROM clause and gives departments' total employees and salaries as a decimal value of all the departments:

```
SELECT a.deptno "Department",
       a.num_emp/b.total_count "%Employees",
       a.sal_sum/b.total_sal "%Salary"
  FROM
```

```
(SELECT deptno, COUNT(*) num_emp, SUM(SAL) sal_sum
   FROM scott.emp
   GROUP BY deptno) a,
(SELECT COUNT(*) total_count, SUM(sal) total_sal
   FROM scott.emp) b ;
```

Hierarchical Queries

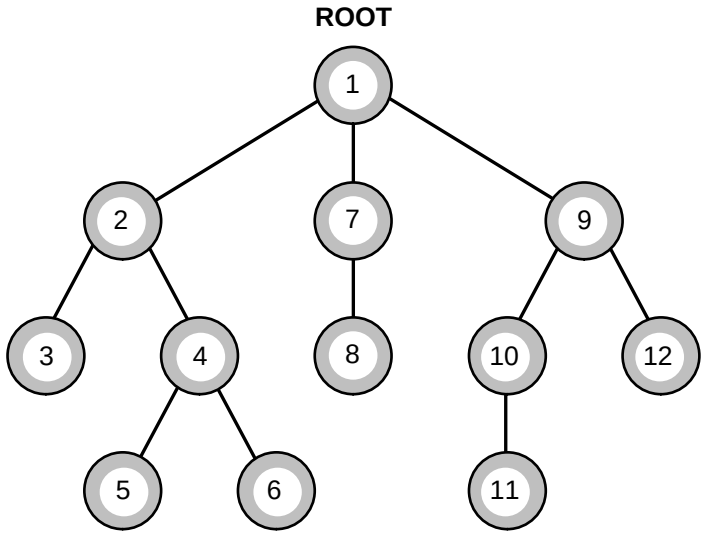
If a table contains hierarchical data, you can select rows in a hierarchical order using the following clauses:

START WITH	specifies the root row(s) of the hierarchy
CONNECT BY	specifies the relationship between parent rows and child rows of the hierarchy
WHERE	restricts the rows returned by the query without affecting other rows of the hierarchy

Oracle uses the information from the above clause to form the hierarchy using the following steps:

1. Oracle selects the root row(s) of the hierarchy—those rows that satisfy the condition of the START WITH clause.
2. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the CONNECT BY clause 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, 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 4–1. In the diagram children appear below their parents.

Figure 4–1 Hierarchical Queries



SELECT statements performing hierarchical queries are subject to the following restrictions:

- A SELECT statement that performs a hierarchical query cannot also perform a join.
- A SELECT statement that performs a hierarchical query cannot select data from a view whose query performs a join.
- If you use the ORDER BY clause in a hierarchical query, Oracle orders rows by the ORDER BY clause, rather than in the order shown in Figure 4–1.

The following sections discuss the START WITH and CONNECT BY clauses.

START WITH Clause

The START WITH clause identifies the row(s) to be used as the root(s) of a hierarchical query. This clause specifies a condition that the roots must satisfy. If you omit this clause, Oracle uses all rows in the table as root rows. A START WITH condition can contain a subquery.

CONNECT BY Clause

The CONNECT BY clause specifies the relationship between parent and child rows in a hierarchical query. This clause contains a condition that defines this

relationship. This condition can be any condition as described in "Conditions" on page 3-90; however, some part of the condition must use the PRIOR operator to refer to the parent row. The part of the condition containing the PRIOR operator must have one of the following forms:

```
PRIOR expr comparison_operator expr
expr comparison_operator PRIOR expr
```

To find the children of a parent row, Oracle evaluates the PRIOR expression 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 clause can contain other conditions to further filter the rows selected by the query. The CONNECT BY clause cannot contain a subquery.

If the CONNECT BY clause results in a loop in the hierarchy, 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.

Example I. The following CONNECT BY clause defines a hierarchical relationship in which the EMPNO value of the parent row is equal to the MGR value of the child row:

```
CONNECT BY PRIOR empno = mgr;
```

Example II. In the following CONNECT BY clause, the PRIOR operator applies only to the EMPNO value. To evaluate this condition, Oracle evaluates EMPNO values for the parent row and MGR, SAL, and COMM values for the child row:

```
CONNECT BY PRIOR empno = mgr AND sal > comm;
```

To qualify as a child row, a row must have a MGR value equal to the EMPNO value of the parent row and it must have a SAL value greater than its COMM value.

The LEVEL Pseudocolumn

SELECT statements that perform hierarchical queries can use the LEVEL pseudocolumn. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, etc. For more information on LEVEL, see the section "Pseudocolumns" on page 2-32.

The number of levels returned by a hierarchical query may be limited by available user memory.

Example I. The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is 'PRESIDENT'. 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)) || ename org_chart,
        empno, mgr, job
    FROM emp
    START WITH job = 'PRESIDENT'
    CONNECT BY PRIOR empno = mgr;

ORG_CHART    EMPNO      MGR        JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
  JONES            7566       7839 MANAGER
    SCOTT          7788       7566 ANALYST
      ADAMS        7876       7788 CLERK
    FORD           7902       7566 ANALYST
      SMITH        7369       7902 CLERK
  BLAKE            7698       7839 MANAGER
    ALLEN          7499       7698 SALESMAN
    WARD           7521       7698 SALESMAN
    MARTIN         7654       7698 SALESMAN
    TURNER         7844       7698 SALESMAN
      JAMES        7900       7698 CLERK
 CLARK             7782       7839 MANAGER
   MILLER          7934       7782 CLERK
```

The following statement is similar to the previous one, except that it does not select employees with the job 'ANALYST'.

```
SELECT LPAD(' ', 2*(LEVEL-1)) || ename org_chart,
     empno, mgr, job
     FROM emp
     WHERE job != 'ANALYST'
     START WITH job = 'PRESIDENT'
     CONNECT BY PRIOR empno = mgr;

ORG_CHART            EMPNO      MGR        JOB
-------------------- ---------- ---------- ---------
KING                       7839            PRESIDENT
  JONES                    7566       7839 MANAGER
     ADAMS                 7876       7788 CLERK
     SMITH                 7369       7902 CLERK
  BLAKE                    7698       7839 MANAGER
    ALLEN                  7499       7698 SALESMAN
    WARD                   7521       7698 SALESMAN
    MARTIN                 7654       7698 SALESMAN
```

```
    TURNER                7844       7698 SALESMAN
    JAMES                 7900       7698 CLERK
  CLARK                   7782       7839 MANAGER
    MILLER                7934       7782 CLERK
```

Oracle does not return the analysts SCOTT and FORD, although it does return employees who are managed by SCOTT and FORD.

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)) || ename org_chart,
empno, mgr, job
    FROM emp
    START WITH job = 'PRESIDENT'
    CONNECT BY PRIOR empno = mgr AND LEVEL <= 2;

ORG_CHART    EMPNO      MGR        JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
  JONES            7566       7839 MANAGER
  BLAKE            7698       7839 MANAGER
  CLARK            7782       7839 MANAGER
```

GROUP BY Clause

Use the GROUP BY clause to group selected rows and return a single row of summary information. Oracle collects each group of rows based on the values of the expression(s) specified in the GROUP BY clause.

If a SELECT statement contains the GROUP BY clause, the select list can contain only the following types of expressions:

- constants
- group functions
- the functions USER, UID, and SYSDATE
- expressions identical to those in the GROUP BY clause
- expressions involving the above expressions that evaluate to the same value for all rows in a group

Expressions in the GROUP BY clause can contain any columns in the tables, views, and snapshots in the FROM clause, regardless of whether the columns appear in the select list.

The GROUP BY clause can contain no more than 255 expressions. The total number of bytes in all expressions in the GROUP BY clause is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter DB_BLOCK_SIZE.

Example I. To return the minimum and maximum salaries for each department in the employee table, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     GROUP BY deptno;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        10       1300       5000
        20        800       3000
        30        950       2850
```

Example II. To return the minimum and maximum salaries for the clerks in each department, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     WHERE job = 'CLERK'
     GROUP BY deptno;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        10       1300      1300
        20        800      1100
        30        950       950
```

HAVING Clause

Use the HAVING clause to restrict which groups of rows defined by the GROUP BY clause are returned by the query. Oracle processes the WHERE, GROUP BY, and HAVING clauses in the following manner:

1. If the statement contains a WHERE clause, Oracle eliminates all rows that do not satisfy it.
2. Oracle calculates and forms the groups as specified in the GROUP BY clause.
3. Oracle removes all groups that do not satisfy the HAVING clause.

Specify the GROUP BY and HAVING clauses after the WHERE and CONNECT BY clauses. If both the GROUP BY and HAVING clauses are specified, they can appear in either order.

Example I. To return the minimum and maximum salaries for the clerks in each department whose lowest salary is below $1,000, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     WHERE job = 'CLERK'
     GROUP BY deptno
     HAVING MIN(sal) < 1000;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        20        800       1100
        30        950        950
```

UNION, UNION ALL, INTERSECT, and MINUS

The UNION, UNION ALL, INTERSECT, and MINUS operators combine the results of two queries into a single result. The number and datatypes of the columns selected by each component query must be the same, but the column lengths can be different. For information, see “Set Operators” on page 3-12.

If more than two queries are combined with set operators, adjacent pairs of queries are evaluated from left to right. You can use parentheses to specify a different order of evaluation.

The total number of bytes in all select list expressions of a component query is 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.

OBJ You cannot use these set operators to combine the results of queries that use the THE or MULTISET keywords.

ORDER BY Clause

Use the ORDER BY clause to order the rows selected by a query. Without an ORDER BY clause, it is not guaranteed that the same query executed more than once will retrieve rows in the same order. The clause specifies either expressions or positions or aliases of expressions in the select list of the statement. Oracle returns rows based on their values for these expressions.

You can specify multiple expressions in the ORDER BY clause. Oracle 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. Oracle sorts nulls following all others in ascending order and preceding all others in descending order.

Sorting by position is useful in the following cases:

- To order by a lengthy select list expression, you can specify its position, rather than duplicate the entire expression, in the ORDER BY clause.
- For compound queries (containing set operators UNION, INTERSECT, MINUS, or UNION ALL), the ORDER BY clause must use positions, rather than explicit expressions. Also, the ORDER BY clause can only appear in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The mechanism by which Oracle sorts values for the ORDER BY clause is specified either explicitly by the NLS_SORT initialization parameter or implicitly by the NLS_LANGUAGE initialization parameter. For information on these parameters, see *Oracle8 Reference*. You can also change the sort mechanism dynamically from one linguistic sort sequence to another using the ALTER SESSION command. You can also specify a specific sort sequence for a single query by using the NLSSORT function with the NLS_SORT parameter in the ORDER BY clause.

The ORDER BY clause is subject to the following restrictions:

- If the ORDER BY clause and the DISTINCT operator both appear in a SELECT statement, the ORDER BY clause cannot refer to columns that do not appear in the select list.
- The ORDER BY clause cannot appear in subqueries within other statements.
- The ORDER BY clause can contain no more than 255 expressions.

If you use the ORDER BY and GROUP BY clauses together, the expressions that can appear in the ORDER BY clause are subject to the same restrictions as the expressions in the select list, described in the "GROUP BY Clause" on page 4-499.

If you use the ORDER BY clause in a hierarchical query, Oracle uses the ORDER BY clause rather than the hierarchy to order the rows.

Example I. To select all salesmen's records from EMP, and order the results by commission in descending order, issue the following statement:

```
SELECT *
    FROM emp
    WHERE job = 'SALESMAN'
```

```
    ORDER BY comm DESC;
```

Example II. To select the employees from EMP ordered first by ascending department number and then by descending salary, issue the following statement:

```
SELECT ename, deptno, sal
    FROM emp
    ORDER BY deptno ASC, sal DESC;
```

To select the same information as the previous SELECT and use the positional ORDER BY notation, issue the following statement:

```
SELECT ename, deptno, sal
    FROM emp
    ORDER BY 2 ASC, 3 DESC;
```

FOR UPDATE Clause

The FOR UPDATE clause locks the rows selected by the query. Once you have selected a row for update, other users cannot lock or update it until you end your transaction. The FOR UPDATE clause signals that you intend to insert, update, or delete the rows returned by the query, but does not require that you perform one of these operations. A SELECT statement with a FOR UPDATE clause is often followed by one or more UPDATE statements with WHERE clauses.

The FOR UPDATE clause cannot be used with the following other constructs:

- DISTINCT operator
- GROUP BY clause
- set operators
- group functions
- OBJ CURSOR operator

The tables locked by the FOR UPDATE clause must all be located on the same database. These locked tables must also be on the same database as any LONG columns and sequences referenced in the same statement.

If a row selected for update is currently locked by another user, Oracle waits until the row is available, locks it, and then returns control to you. You can use the NOWAIT option to cause Oracle to terminate the statement without waiting if such a row is already locked.

OBJ The rows returned from subqueries whose column value is a nested table or a VARRAY, not a scalar value, are not locked. Only the top-level rows of such select lists are locked.

LOB Locking

Prior to updating a LOB value, you must lock the row containing the LOB. One way to lock the row is with a SELECT... FOR UPDATE statement.

Example.

```
INSERT INTO t_table VALUES (1, 'abcd');

COMMIT;
DECLARE
  num_var      NUMBER;
  clob_var     CLOB;
  clob_locked  CLOB;
  write_amount NUMBER;
  write_offset NUMBER;
  buffer       VARCHAR2(20) := 'efg';

BEGIN
  SELECT clob_col INTO clob_locked FROM t_table
  WHERE num_col = 1 FOR UPDATE;

  write_amount := 3;
   dbms_lob.write(clob_locked, write_amount, write_offset, buffer);
END;
```

FOR UPDATE OF Clause

The columns in the OF clause only specify which tables' rows are locked. The specific columns of the table that you specify are not significant. If you omit the OF clause, Oracle locks the selected rows from all the tables in the query.

Example I. The following statement locks rows in the EMP table with clerks located in New York and locks rows in the DEPT table with departments in New York that have clerks:

```
SELECT empno, sal, comm
    FROM emp, dept
    WHERE job = 'CLERK'
        AND emp.deptno = dept.deptno
        AND loc = 'NEW YORK'
    FOR UPDATE;
```

Example II. The following statement locks only those rows in the EMP table with clerks located in New York; no rows are locked in the DEPT table:

```
SELECT empno, sal, comm
```

```
FROM emp, dept
WHERE job = 'CLERK'
    AND emp.deptno = dept.deptno
    AND loc = 'NEW YORK'
FOR UPDATE OF emp.sal;
```

Joins

A *join* is a query that combines rows from two or more tables, views, or snapshots. Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a *join condition*. To execute a join, Oracle 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, in the case of the cost-based optimization approach, statistics for the tables.

In addition to join conditions, the WHERE clause of a join query 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.

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.

Example I. This equijoin returns the name and job of each employee and the number and name of the department in which the employee works:

```
SELECT ename, job, dept.deptno, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno;

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
```

You must use a join to return this data because employee names and jobs are stored in a different table than department names. Oracle combines rows of the two tables according to this join condition:

```
emp.deptno = dept.deptno
```

Example II. The following equijoin returns the name, job, department number, and department name of all clerks:

```
SELECT ename, job, dept.deptno, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno
     AND job = 'CLERK';

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
JAMES      CLERK             30 SALES
```

This query is identical to Example XII except that it uses an additional WHERE clause condition to return only rows with a JOB value of 'CLERK':

```
job = 'CLERK'
```

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 combines and returns rows of the table that satisfy the join condition.

Example . This query uses a self join to return the name of each employee along with the name of the employee's manager:

```
SELECT e1.ename||' works for '||e2.ename
"Employees and their Managers"
    FROM emp e1, emp e2   WHERE e1.mgr = e2.empno;

Employees and their Managers
------------------------------
BLAKE works for KING
CLARK works for KING
JONES works for KING
FORD works for JONES
SMITH works for FORD
ALLEN works for BLAKE
WARD works for BLAKE
MARTIN works for BLAKE
SCOTT works for JONES
TURNER works for BLAKE
ADAMS works for SCOTT
JAMES works for BLAKE
MILLER works for CLARK
```

The join condition for this query uses the aliases E1 and E2 for the EMP table:

```
e1.mgr = e2.empno
```

Cartesian Products

If two tables in a join query have no join condition, Oracle 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

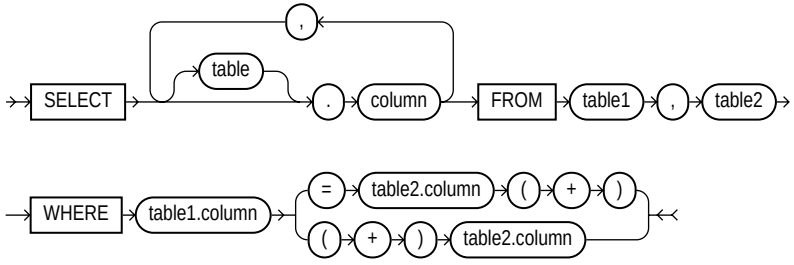
joins three or more tables and you do not specify a join condition for a specific pair, the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Outer Joins

The outer join extends the result of a simple join. An *outer join* returns all rows that satisfy the join condition and those rows from one table for which no rows from the other satisfy the join condition. Such rows are not returned by a simple join. To write a query that performs an outer join of tables A and B and returns all rows from A, apply the outer join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, Oracle returns NULL for any select list expressions containing columns of B.

This is the basic syntax of an outer join of two tables:

outer_join ::=



Outer join queries are subject to the following rules and restrictions:

- The (+) operator can appear only in the WHERE clause, not in the select list, and can be applied only to a column of a table or view.
- If A and B are joined by multiple join conditions, the (+) operator must be used in all of these conditions.
- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain a column marked with the (+) operator.
- A condition containing the (+) operator cannot be combined with another condition using the OR logical operator.
- A condition cannot use the IN comparison operator to compare with another expression a column marked with the (+) operator.

- A condition cannot compare with a subquery any column marked with the (+) operator.

If the WHERE clause contains a condition that compares a column from table B with a constant, 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.

In a query that performs outer joins of more than two pairs of tables, a single table can be the NULL-generated table for only one other table. For this reason, you cannot apply the (+) operator to columns of B in the join condition for A and B and the join condition for B and C.

Example I. This query uses an outer join to extend the results of Example XIV:

```
SELECT ename, job, dept.deptno, dname
    FROM emp, dept
    WHERE emp.deptno (+) = dept.deptno;

ENAME      JOB       DEPTN      DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
                             40 OPERATIONS
```

In this outer join, Oracle returns a row containing the OPERATIONS department even though no employees work in this department. Oracle returns NULL in the ENAME and JOB columns for this row. The join query in Example X selects only departments that have employees.

The following query uses an outer join to extend the results of Example XV:

```
SELECT ename, job, dept.deptno, dname
    FROM emp, dept
    WHERE emp.deptno (+) = dept.deptno
        AND job (+) = 'CLERK';
```

```
ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
JAMES      CLERK             30 SALES
                             40 OPERATIONS
```

In this outer join, Oracle returns a row containing the OPERATIONS department even though no clerks work in this department. The (+) operator on the JOB column ensures that rows for which the JOB column is NULL are also returned. If this (+) were omitted, the row containing the OPERATIONS department would not be returned because its JOB value is not 'CLERK'.

Example II. This example shows four outer join queries on the CUSTOMERS, ORDERS, LINEITEMS, and PARTS tables. These tables are shown here:

```
SELECT custno, custname
    FROM customers;

CUSTNO     CUSTNAME
---------- --------------------
1          Angelic Co.
2          Believable Co.
3          Cabels R Us

SELECT orderno, custno,
    TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE"
    FROM orders;

ORDERNO    CUSTNO     ORDERDATE
---------- ---------- -----------
      9001          1 OCT-13-1993
      9002          2 OCT-13-1993
      9003          1 OCT-20-1993
      9004          1 OCT-27-1993
      9005          2 OCT-31-1993

SELECT orderno, lineno, partno, quantity
    FROM lineitems;

ORDERNO    LINENO     PARTNO     QUANTITY
---------- ---------- ---------- ----------
      9001          1        101         15
```

```
      9001          2        102         10
      9002          1        101         25
      9002          2        103         50
      9003          1        101         15
      9004          1        102         10
      9004          2        103         20

SELECT partno, partname
    FROM parts;

PARTNO PARTNAME
------ --------
   101 X-Ray Screen
   102 Yellow Bag
   103 Zoot Suit
```

Note that the customer Cables R Us has placed no orders and that order number 9005 has no line items.

The following outer join returns all customers and the dates they placed orders. The (+) operator ensures that customers who placed no orders are also returned:

```
SELECT custname, TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE"
    FROM customers, orders
    WHERE customers.custno = orders.custno (+);

CUSTNAME             ORDERDATE
-------------------- --------------
Angelic Co.          OCT-13-1993
Angelic Co.          OCT-20-1993
Angelic Co.          OCT-27-1993
Believable Co.       OCT-13-1993
Believable Co.       OCT-31-1993
Cables R Us
```

The following outer join builds on the result of the previous one by adding the LINEITEMS table to the FROM clause, columns from this table to the select list, and a join condition joining this table to the ORDERS table to the WHERE clause. This query joins the results of the previous query to the LINEITEMS table and returns all customers, the dates they placed orders, and the part number and quantity of each part they ordered. The first (+) operator serves the same purpose as in the previous query. The second (+) operator ensures that orders with no line items are also returned:

```
SELECT custname,
```

```
TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE",
partno,
quantity
    FROM customers, orders, lineitems
    WHERE customers.custno = orders.custno (+)
    AND orders.orderno = lineitems.orderno (+);

CUSTNAME             ORDERDATE          PARTNO   QUANTITY
-------------------- -------------- ---------- ----------
Angelic Co.          OCT-13-1993           101         15
Angelic Co.          OCT-13-1993           102         10
Angelic Co.          OCT-20-1993           101         15
Angelic Co.          OCT-27-1993           102         10
Angelic Co.          OCT-27-1993           103         20
Believable Co.       OCT-13-1993           101         25
Believable Co.       OCT-13-1993           103         50
Believable Co.       OCT-31-1993
Cables R Us
```

The following outer join builds on the result of the previous one by adding the PARTS table to the FROM clause, the PARTNAME column from this table to the select list, and a join condition joining this table to the LINEITEMS table to the WHERE clause. This query joins the results of the previous query to the PARTS table to return all customers, the dates they placed orders, and the quantity and name of each part they ordered. The first two (+) operators serve the same purposes as in the previous query. The third (+) operator ensures that rows with NULL part numbers are also returned:

```
SELECT custname, TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE",
    quantity, partname
    FROM customers, orders, lineitems, parts
    WHERE customers.custno = orders.custno (+)
    AND orders.orderno = lineitems.orderno (+)
    AND lineitems.partno = parts.partno (+);

CUSTNAME             ORDERDATE        QUANTITY PARTNAME
-------------------- -------------- ---------- ------------
Angelic Co.          OCT-13-1993            15 X-Ray Screen
Angelic Co.          OCT-13-1993            10 Yellow Bag
Angelic Co.          OCT-20-1993            15 X-Ray Screen
Angelic Co.          OCT-27-1993            10 Yellow Bag
Angelic Co.          OCT-27-1993            20 Zoot Suit
Believable Co.       OCT-13-1993            25 X-Ray Screen
Believable Co.       OCT-13-1993            50 Zoot Suit
Believable Co.       OCT-31-1993
```

```
Cables R Us
```

Related Topics

DELETE on page 4-374
SET CONSTRAINT(S) on page 4-514
UPDATE on page 4-542

SET CONSTRAINT(S)

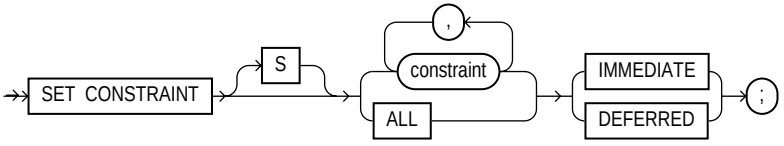
Purpose

To specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement (IMMEDIATE) or when the transaction is committed (DEFERRED). For illustrations, see “Examples” on page 4-514.

Prerequisites

Setting when a deferrable constraint is checked requires that the table to which the constraint applies must be in your own schema or you must have SELECT privilege on the table.

Syntax



Keywords and Parameters

constraint	is the name of the integrity constraint.
ALL	sets all deferrable constraints for this transaction.
IMMEDIATE	indicates that the conditions specified by the deferrable constraint are checked immediately after each DML statement.
DEFERRED	indicates that the conditions specified by the deferrable constraint are checked with the transaction is committed.

You can verify the success of deferrable constraints prior to committing them by issuing a SET CONSTRAINTS ALL IMMEDIATE statement.

Examples

Example I. The following example sets all deferrable constraints in this transaction to be checked immediately following each DML statement:

```
SET CONSTRAINTS ALL IMMEDIATE;
```

Example II. The following statement checks three deferred constraints when the transaction is committed:

```
SET CONSTRAINTS unq_name, scott.nn_sal,
  adams.pk_dept@dblink DEFERRED;
```

Related Topics

CREATE TABLE on page 4-306
ALTER TABLE on page 4-106
ENABLE clause on page 4-417
DISABLE clause on page 4-380
ALTER SESSION on page 4-58

SET ROLE

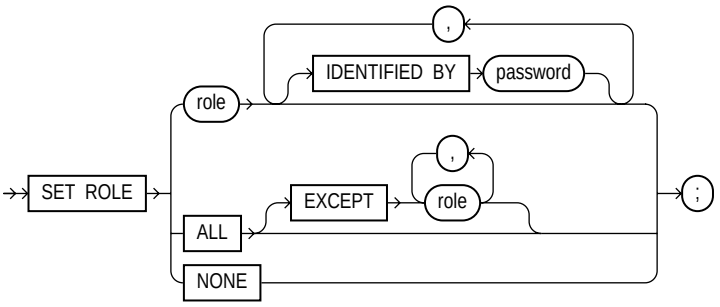
Purpose

To enable and disable roles for your current session. For illustrations, see "Examples" on page 4-518.

Prerequisites

You must already have been granted the roles that you name in the SET ROLE statement. See also "Privilege Domains" on page 4-517.

Syntax



Keywords and Parameters

role is a role to be enabled for the current session. Any roles not listed are disabled for the current session.

password is the password for a role. If the role has a password, you must specify the password to enable the role.

ALL enables all roles granted to you for the current session, except those optionally listed in the EXCEPT clause.

EXCEPT Roles listed in the EXCEPT clause must be roles granted directly to you; they cannot be roles granted to you through other roles. You cannot use this option to enable roles with passwords that have been granted directly to you.

	If you list a role in the EXCEPT clause that has been granted to you both directly and through another role, the role remains enabled by virtue of the role to which it has been granted.
NONE	disables all roles for the current session.

Privilege Domains

At logon, Oracle establishes your default privilege domain by enabling your default roles. Your default privilege domain contains all privileges granted explicitly to you and all privileges in the privilege domains of your default roles. You can then perform any operations authorized by the privileges in your default privilege domain.

Changing Your Privilege Domain

During your session, you can change your privilege domain with the SET ROLE command, which changes the roles currently enabled for your session. You can change your enabled roles any number of times during a session. The number of roles that can be concurrently enabled is limited by the initialization parameter MAX_ENABLED_ROLES.

You can use the SET ROLE command to enable or disable any of the following roles:

- roles that have been granted directly to you
- roles granted to you through other roles

You cannot use the SET ROLE command to enable roles that you have not been granted either directly or through other roles.

Your current privilege domain is also changed in the following cases:

- if you are granted a privilege
- if one of your privileges is revoked
- if one of your enabled roles is revoked
- if the privilege domain of one of your enabled roles is changed

If none of the above conditions occur and you do not issue the SET ROLE command, your default privilege domain remains in effect for the duration of your session. In the last two cases, the change in your privilege domain does not take effect until you log on to Oracle again or issue a SET ROLE statement.

You can determine which roles are in your current privilege domain at any time by examining the SESSION_ROLES data dictionary view.

To change your default roles, use the ALTER USER command.

Examples

Example I. To enable the role GARDENER identified by the password MARIGOLDS for your current session, issue the following statement:

```
SET ROLE gardener IDENTIFIED BY marigolds;
```

Example II. To enable all roles granted to you for the current session, issue the following statement:

```
SET ROLE ALL;
```

Example III. To enable all roles granted to you except BANKER, issue the following statement:

```
SET ROLE ALL EXCEPT banker IV;
```

Example IV. To disable all roles granted to you for the current session, issue the following statement:

```
SET ROLE NONE;
```

Related Topics

ALTER USER on page 4-150
CREATE ROLE on page 4-272

SET TRANSACTION

Purpose

For the current transaction, to:

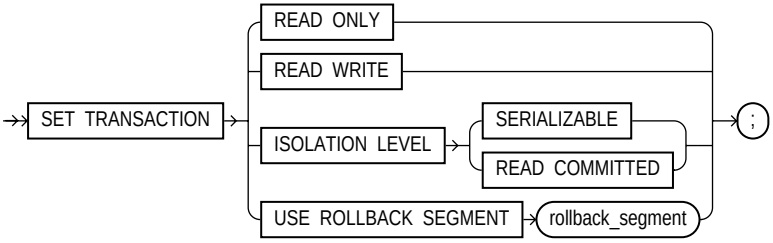
- establish it as a read-only or read-write transaction
- establish the isolation level
- assign it to a specified rollback segment

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. Note also that Oracle implicitly commits the current transaction before and after executing a data definition language statement.

Prerequisites

If you use a SET TRANSACTION statement, it must be the first statement in your transaction. However, a transaction need not have a SET TRANSACTION statement.

Syntax



Keywords and Parameters

READ ONLY establishes the current transaction as a read-only transaction. See also “Establishing Read-Only Transactions” on page 4-520.

READ WRITE establishes the current transaction as a read-write transaction.

ISOLATION LEVEL	specifies how transactions containing database modifications are handled.	
	SERIALIZABLE	specifies serializable transaction isolation mode as defined in SQL92. 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. **Note:** The COMPATIBLE initialization parameter must be set to 7.3.0 or higher for SERIALIZABLE mode to work.
	READ COMMITTED	is the default Oracle 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	assigns the current transaction to the specified rollback segment. This option also implicitly establishes the transaction as a read-write transaction. You cannot use the READ ONLY option and the USE ROLLBACK SEGMENT clause in a single SET TRANSACTION statement or in different statements in the same transaction. Read-only transactions do not generate rollback information and therefore are not assigned rollback segments. See also “Assigning Transactions to Rollback Segments” on page 4-521.	

Establishing Read-Only Transactions

The default state for all transactions is statement-level read consistency. You can explicitly specify this state by issuing a SET TRANSACTION statement with the READ WRITE option.

You can establish transaction-level read consistency by issuing a SET TRANSACTION statement with the READ ONLY option. After a transaction has been established as read-only, all subsequent queries in that transaction only see changes committed before the transaction began. Read-only transactions are very useful for reports that run multiple queries against one or more tables while other users update these same tables.

Only the following statements are permitted in a read-only transaction:

- SELECT (but not statements with the FOR UPDATE clause)
- LOCK TABLE
- SET ROLE
- ALTER SESSION
- ALTER SYSTEM

INSERT, UPDATE, and DELETE statements and SELECT statements with the FOR UPDATE clause are not permitted. Any DDL statement implicitly ends the read-only transaction.

The read consistency that read-only transactions provide is implemented in the same way as statement-level read consistency. Every statement by default uses a consistent view of the data as of the time the statement is issued. Read-only transactions present a consistent view of the data as of the time that the SET TRANSACTION READ ONLY statement is issued. Read-only transactions provide read consistency is for all nodes accessed by distributed queries and local queries.

You cannot toggle between transaction-level read consistency and statement-level read consistency in the same transaction. A SET TRANSACTION statement can only be issued as the first statement of a transaction.

Example . The following statements could be run at midnight of the last day of every month to count how many ships and containers the company owns. This report would not be affected by any other user who might be adding or removing ships and / or containers.

```
COMMIT
SET TRANSACTION READ ONLY
SELECT COUNT(*) FROM ship
SELECT COUNT(*) FROM container
COMMIT;
```

The last COMMIT statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

Assigning Transactions to Rollback Segments

If you issue a DML statement in a transaction, Oracle assigns the transaction to a rollback segment. The rollback segment holds the information necessary to undo the changes made by the transaction. You can issue a SET TRANSACTION statement with the USE ROLLBACK SEGMENT clause to choose a specific rollback segment for your transaction. If you do not choose a rollback segment, Oracle chooses one randomly and assigns your transaction to it.

SET TRANSACTION lets you to assign transactions of different types to rollback segments of different sizes:

- Assign OLTP transactions, or small transactions containing only a few DML statements that modify only a few rows, to small rollback segments if there are no long-running queries concurrently reading the same tables. Small rollback segments are more likely to remain in memory.

- Assign transactions that modify tables that are concurrently being read by long-running queries to large rollback segments so that the rollback information needed for the read-consistent queries is not overwritten.
- Assign transactions with bulk DML statements, or statements that insert, update, or delete large amounts of data, to rollback segments large enough to hold the rollback information for the transaction.

Example. The following statement assigns your current transaction to the rollback segment OLTP_5:

```
SET TRANSACTION USE ROLLBACK SEGMENT oltp_5;
```

Related Topics

COMMIT on page 4-185
ROLLBACK on page 4-484
SAVEPOINT on page 4-487

STORAGE clause

Purpose

To specify storage characteristics for tables, indexes, clusters, and rollback segments, and the default storage characteristics for tablespaces. See also “Specifying Storage Parameters” on page 4-526.

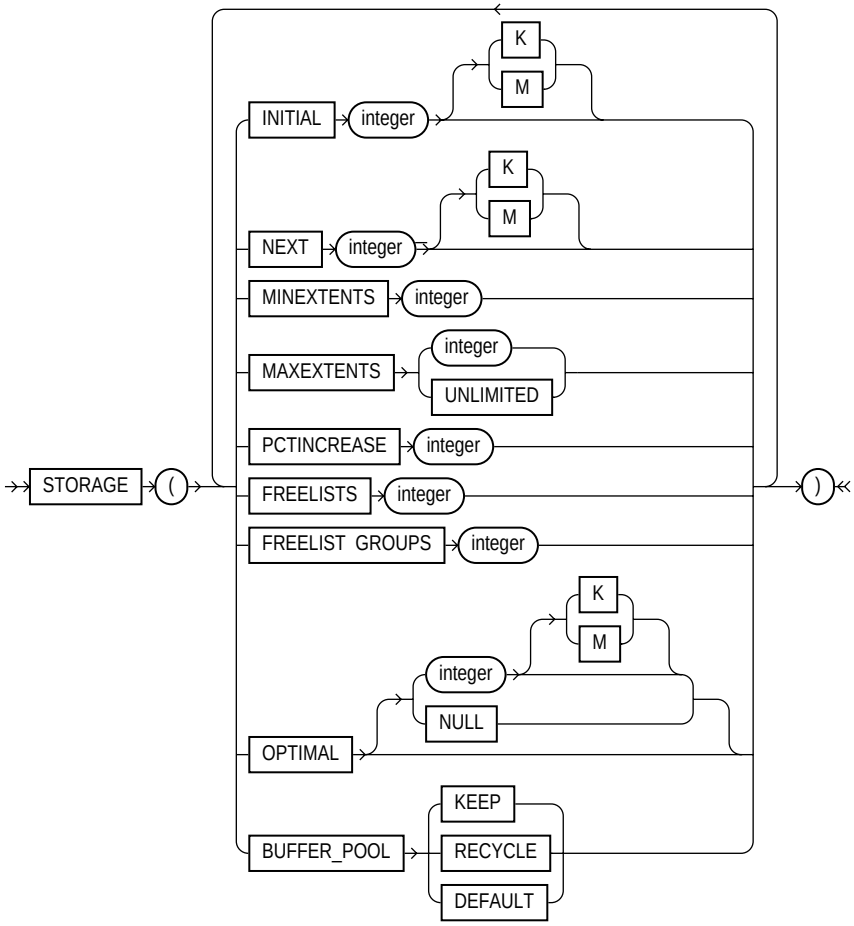
Prerequisites

The STORAGE clause can appear in commands that create or alter any of the following schema objects:

- clusters
- indexes
- rollback segments
- snapshots
- snapshot logs
- tables
- tablespaces
- partitions

To change the value of a STORAGE parameter, you must have the privileges necessary to use the appropriate create or alter command.

Syntax



Keywords and Parameters

INITIAL — specifies the size in bytes of the object's first extent. Oracle allocates space for this extent when you create the schema object. You can use K or M to specify this size in kilobytes or megabytes. The default value is the size of 5 data blocks. The minimum value is the size of 2 data blocks. 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, and rounds up to the next multiple of 5 data blocks for values greater than 5 data blocks.

NEXT	specifies the size in bytes of the next extent to be allocated to the object. You can use K or M to specify the size in kilobytes or megabytes. 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, as described in *Oracle8 Concepts*.
PCTINCREASE	specifies 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. You cannot specify PCTINCREASE for rollback segments. Rollback segments always have a PCTINCREASE value of 0. Oracle rounds the calculated size of each new extent up to the next multiple of the data block size.
MINEXTENTS	specifies the total number of extents to allocate when the object is created. This parameter enables you to allocate a large amount of space when you create an object, even if the space available is not contiguous. The default and minimum value is 1, meaning that Oracle only allocates the initial extent, except for rollback segments for which the default and minimum value is 2. The maximum value depends on your operating system. 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 parameters.
MAXEXTENTS	specifies 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 value of 2). The default and maximum values depend your data block size. UNLIMITED — specifies that extents should be allocated automatically as needed. Do not use this option for rollback segments. See also "Rollback Segments and MAXEXTENTS UNLIMITED" on page 4-527.
FREELIST GROUPS	for schema objects other than tablespace, specifies the number of groups of free lists for a table, partition, cluster, or index. The default and minimum value for this parameter is 1. Only use this parameter if you are using Oracle with the Parallel Server option in parallel mode.
FREELISTS	for objects other than tablespace, specifies the number of groups 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, Oracle returns an error message indicating the maximum value.

You can specify the FREELISTS and the FREELIST GROUPS parameters only in CREATE TABLE, CREATE CLUSTER, and CREATE INDEX statements.

OPTIMAL is relevant only to rollback segments. It specifies an optimal size in bytes for a rollback segment. You can use K or M to specify this size in kilobytes or megabytes. 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.

- NULL — specifies no optimal size for the rollback segment, meaning that Oracle never deallocates the rollback segment's extents. This is the default behavior.

The value of this parameter cannot be less than the space initially allocated for the rollback segment specified 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.

BUFFER_POOL defines a default buffer pool (cache) for a schema object. All blocks for the object are stored in the specified cache. If a buffer pool is defined 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.

Note: BUFFER_POOL is not a valid option for creating or altering tablespaces or rollback segments. For more information about using multiple buffer pools, see *Oracle8 Tuning*.

- KEEP — retains the schema object in memory to avoid I/O operations.
- RECYCLE — eliminates blocks from memory as soon as they are no longer needed, thus preventing an object from taking up unnecessary cache space.
- DEFAULT — always exists for objects not assigned to KEEP or RECYCLE.

Specifying Storage Parameters

The storage parameters affect both how long it takes to access data stored in the database and how efficiently space in the database is used. For a discussion of the effects of these parameters, see *Oracle8 Tuning*.

When you create a tablespace, you can specify values for the storage parameters. These values serve as default values for segments allocated in the tablespace.

When you create a cluster, index, rollback segment, snapshot, snapshot log, or table, you can specify values for the storage parameters for the segments allocated to these objects. If you omit any storage parameter, Oracle uses the value of that parameter specified for the tablespace. However, when creating a rollback segment,

you cannot specify PCTINCREASE (which is always 0) or MINEXTENTS (which is always 2).

When you alter a cluster, index, rollback segment, snapshot, snapshot log, or table, you can change the values of storage parameters. The new values only affect future extent allocations. For this reason, you cannot change the values of the INITIAL and MINEXTENTS parameter. If you change the value of the NEXT parameter, the next allocated extent will have the specified size, regardless of the size of the most recently allocated extent and the value of the PCTINCREASE parameter. If you change the value of the PCTINCREASE parameter, Oracle calculates the size of the next extent using this new value and the size of the most recently allocated extent.

When you alter a tablespace, you can change the values of storage parameters. The new values serve as default values only to subsequently allocated segments (or subsequently created objects).

Rollback Segments and MAXEXTENTS UNLIMITED

It is not good practice to create or alter a rollback segment to use MAXEXTENTS UNLIMITED. Rogue transactions containing inserts, updates, or deletes, that continue for a long time will continue to create new extents until a disk is full.

A rollback segment that you create without specifying the STORAGE option has the same storage parameters as the tablespace that the rollback segment is created in. Thus, if the tablespace is created with MAXEXTENT UNLIMITED, then the rollback segment would also have the same default.

Examples

Example I. The following statement creates a table and provides storage parameter values:

```
CREATE TABLE dept
    (deptno     NUMBER(2),
     dname      VARCHAR2(14),
     loc        VARCHAR2(13) )
     STORAGE  ( INITIAL 100K  NEXT     50K
                  MINEXTENTS 1  MAXEXTENTS 50  PCTINCREASE 5 );
```

Oracle allocates space for the table based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 1, so Oracle allocates 1 extent for the table upon creation.

- The INITIAL value is 100K, so the first extent's size is 100 kilobytes.
- If the table data grows to exceed the first extent, Oracle allocates a second extent. The NEXT value is 50K, so the second extent's size would be 50 kilobytes.
- If the table data subsequently grows to exceed the first two extents, Oracle allocates a third extent. The PCTINCREASE value is 5, so the calculated size of the third extent is 5% larger than the second extent, or 52.5 kilobytes. If the data block size is 2 kilobytes, Oracle rounds this value to 52 kilobytes.

  If the table data continues to grow, Oracle allocates more extents, each 5% larger than the previous one.
- The MAXEXTENTS value is 50, so Oracle can allocate as many as 50 extents for the table.

Example II. The following statement creates a rollback segment and provides storage parameter values:

```
CREATE ROLLBACK SEGMENT rsone
    STORAGE ( INITIAL  10K  NEXT 10K
              MINEXTENTS 2  MAXEXTENTS 25
               OPTIMAL 50K );
```

Oracle allocates space for the rollback segment based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 2, so Oracle allocates 2 extents for the rollback segment upon creation.
- The INITIAL value is 10K, so the first extent's size is 10 kilobytes.
- The NEXT value is 10K, so the second extent's size is 10 kilobytes.
- If the rollback data exceeds the first two extents, Oracle allocates a third extent. The PCTINCREASE value for rollback segments is always 0, so the third and subsequent extents are the same size as the second extent, 10 kilobytes.
- The MAXEXTENTS value is 25, so Oracle can allocate as many as 25 extents for the rollback segment.
- The OPTIMAL value is 50K, so Oracle deallocates extents if the rollback segment exceeds 50 kilobytes. Note that Oracle deallocates only extents that contain data for transactions that are no longer active.

Related Topics

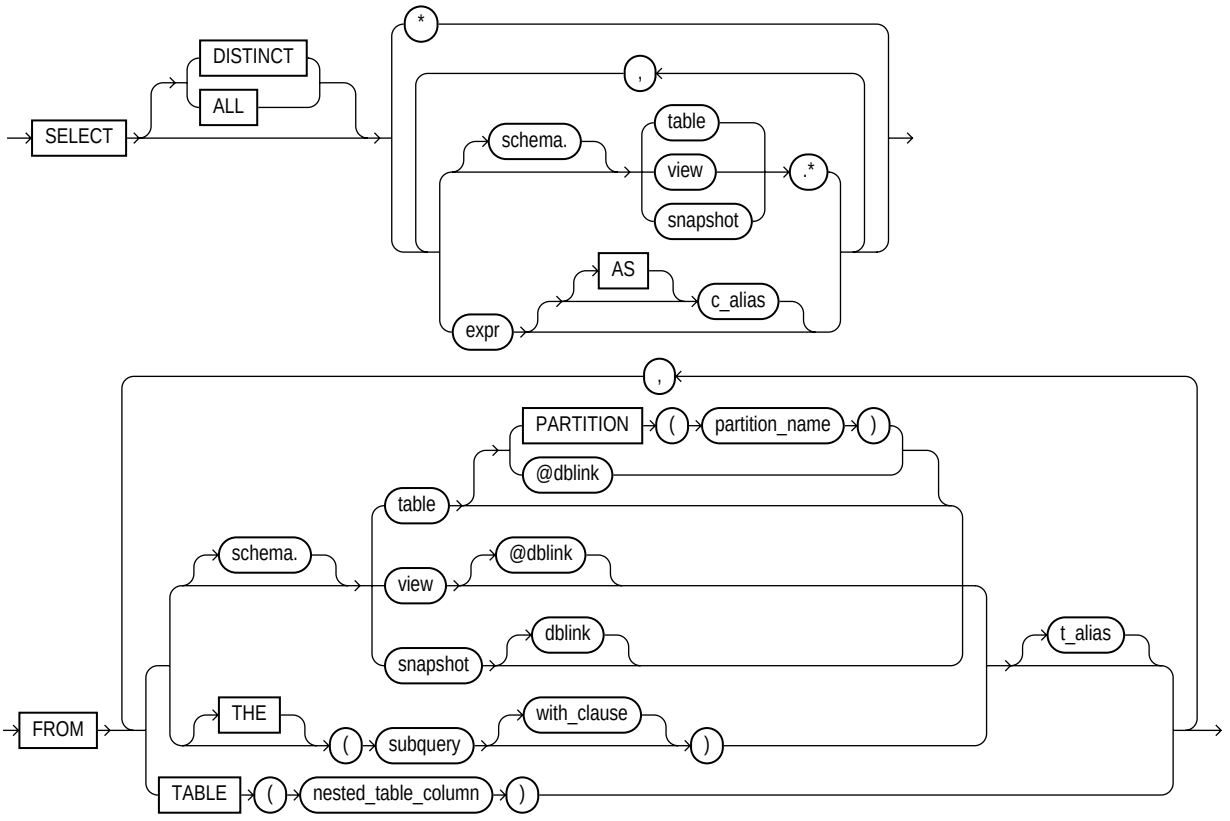
CREATE CLUSTER on page 4-207
CREATE INDEX on page 4-237
CREATE ROLLBACK SEGMENT on page 4-275
CREATE TABLE on page 4-306
CREATE TABLESPACE on page 4-328

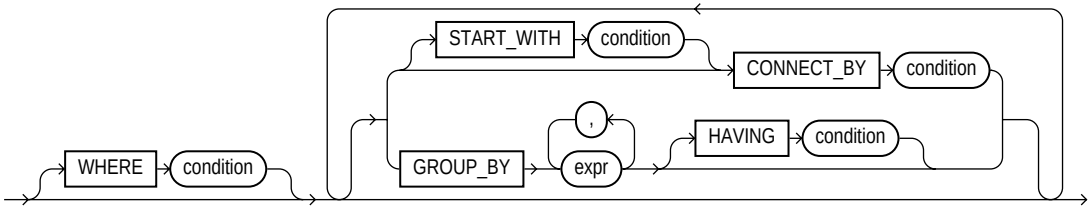
Subqueries

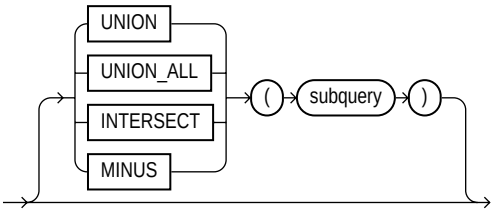
Purpose

A *subquery* is a form of the SELECT command that appears inside another SQL statement. A subquery is sometimes called a *nested query*. The statement containing a subquery is called the *parent statement*. The rows returned by the subquery are used by the parent statement. See also "Using Subqueries" on page 4-532.

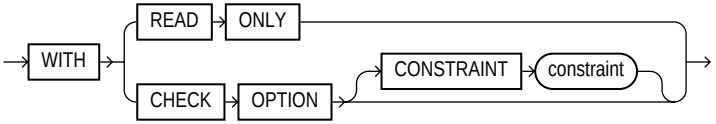
Syntax







WITH_clause::=



Keywords and Parameters

WITH READ ONLY — specifies that the subquery cannot be updated.

WITH CHECK OPTION — specifies that, if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, changes to that table that would produce rows excluded from the subquery are prohibited. In other words, the following statement:

```
INSERT INTO (SELECT ename, deptno FROM emp
             WHERE deptno < 10)
       VALUES ('Taylor', 20);
```

would be legal, but

```
INSERT INTO (SELECT ename, deptno FROM emp
             WHERE deptno < 10
               WITH CHECK OPTION)
       VALUES ('Taylor', 20);
```

would be rejected.

OBJ THE	informs Oracle that the column value returned by the subquery is a nested table, not a scalar value. A subquery prefixed by THE is called a *flattened subquery*. “Using Flattened Subqueries” on page 4-533.
OBJ TABLE *(nested_table_ column)*	identifies the nested table column correlated to the outer query.

Other keywords and parameters function as they are described in SELECT on page 4-489. For more information, see “Correlated Subqueries” on page 4-534, “Selecting from the DUAL Table” on page 4-535, “Using Sequences” on page 4-536, and “Distributed Queries” on page 4-536.

Using Subqueries

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 snapshot in a CREATE VIEW or CREATE SNAPSHOT statement
- to define one or more values to be assigned to existing rows in an UPDATE statement
- to provide values for conditions in WHERE, HAVING, and START WITH clauses 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, UDPATE, and DELETE statements. Subqueries so used can employ correlation variables, but only those defined within the subquery itself, not outer references.

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 is evaluated once for the entire parent statement, in contrast to a correlated subquery which is evaluated once per row processed by the parent statement.

A subquery can itself contain a subquery. Oracle places no limit on the level of query nesting.

Example I. To determine who works in Taylor's department, issue the following statement:

```
SELECT ename, deptno
    FROM emp
    WHERE deptno =
        (SELECT deptno
            FROM emp
            WHERE ename = 'TAYLOR');
```

Example II. To give all employees in the EMP table a 10% raise if they have not already been issued a bonus (if they do not appear in the BONUS table), issue the following statement:

```
UPDATE emp
    SET sal = sal * 1.1
    WHERE empno NOT IN (SELECT empno FROM bonus);
```

Example III. To create a duplicate of the DEPT table named NEWDEPT, issue the following statement:

```
CREATE TABLE newdept (deptno, dname, loc)
    AS SELECT deptno, dname, loc FROM dept;
```

Using Flattened Subqueries

OBJ To manipulate the individual rows of a nested table stored in a database column, use the keyword THE. You must prefix THE to a subquery that returns a single column value or an expression that yields a nested table. If the subquery returns more than a single column value, a run-time error results. Because the value is a nested table, not a scalar value, Oracle must be informed, which is what THE does.

The following example adds a new row to department 40's nested table stored in column PROJECTS:

```
INSERT INTO
  THE(SELECT projects FROM dept WHERE deptno = 40)
  VALUES(33, 'Install new email system', 14875);
```

This example increases the budgets for two projects assigned to department 70:

```
UPDATE
  THE(SELECT projects FROM dept WHERE deptno = 70)
    SET budget = budget + 1000
    WHERE projno IN (24, 25);
```

Correlated Subqueries

A *correlated subquery* is a subquery that is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement. 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);
```

This discussion focuses on correlated subqueries in SELECT statements; it also applies to correlated subqueries in UPDATE and DELETE statements.

You can use a correlated subquery to answer a multiple-part question whose answer depends on the value in each row processed by the parent statement. For example, a correlated subquery can be used 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.

Oracle performs a correlated subquery when the subquery references a column from a table from the parent statement.

Oracle resolves unqualified columns in the subquery by looking in the tables of the subquery, then in the tables of the parent statement, then in the tables of the next enclosing parent statement, and so on. Oracle resolves all unqualified columns in the subquery to the same table. If the tables in a subquery and parent query contain a column with the same name, a reference to the column of a table from the parent query must be prefixed by the table name or alias. To make your statements easier for you to read, always qualify the columns in a correlated subquery with the table, view, or snapshot name or alias.

In an UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table. For example, you could use a correlated subquery to roll up four quarterly sales tables into a yearly sales table.

In a DELETE statement, you can use a correlated query to delete only those rows that also exist in another table.

Example . The following statement returns data about employees whose salaries exceed the averages for their departments. The following statement assigns an alias to EMP, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal
    FROM emp x
    WHERE sal > (SELECT AVG(sal)
          FROM emp
          WHERE x.deptno = deptno)
    ORDER BY deptno;
```

For each row of the EMP table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs these steps for each row of the EMP table:

1. The DEPTNO of the row is determined.
2. The DEPTNO is then used to evaluate the parent query.
3. If that row's salary is greater than the average salary for that row's department, then the row is returned.

The subquery is evaluated once for each row of the EMP table.

Selecting from the DUAL Table

DUAL is a table automatically created by Oracle 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 command. Because DUAL has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table.

Example . The following statement returns the current date:

```
SELECT SYSDATE FROM DUAL;
```

You could select SYSDATE from the EMP table, but Oracle would return 14 rows of the same SYSDATE, one for every row of the EMP table. Selecting from DUAL is more convenient.

Using Sequences

The sequence pseudocolumns NEXTVAL and CURRVAL can also appear in the select list of a SELECT statement. For information on sequences and their use, see CREATE SEQUENCE on page 4-281 and "Pseudocolumns" on page 2-32.

Example . The following statement increments the ZSEQ sequence and returns the new value:

```
SELECT zseq.nextval
    FROM dual;
```

The following statement selects the current value of ZSEQ:

```
SELECT zseq.currval
    FROM dual;
```

Distributed Queries

Oracle's distributed database management system architecture allows you to access data in remote databases using Net8 and an Oracle server. You can identify a remote table, view, or snapshot 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 snapshot. For more information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-54.

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. For example, the following statement will cause an error:

```
SELECT emp_ny.*
    FROM emp_ny@ny, dept
    WHERE emp_ny.deptno = dept.deptno
    AND dept.dname = 'ACCOUNTING'
    FOR UPDATE OF emp_ny.sal;
```

The following statement fails because it selects LONG_COLUMN, a LONG value, from the EMP_REVIEW table on the NY database and locks the EMP table on the local database:

```
SELECT emp.empno, review.long_column, emp.sal
```

```
    FROM emp, emp_review@ny review
    WHERE emp.empno = emp_review.empno
    FOR UPDATE OF emp.sal;
```

Example . This example shows a query that joins the DEPT table on the local database with the EMP table on the HOUSTON database:

```
SELECT ename, dname
    FROM emp@houston, dept
    WHERE emp.deptno = dept.deptno;
```

Related Topics

DELETE on page 4-374
SET CONSTRAINT(S) on page 4-514
UPDATE on page 4-542

TRUNCATE

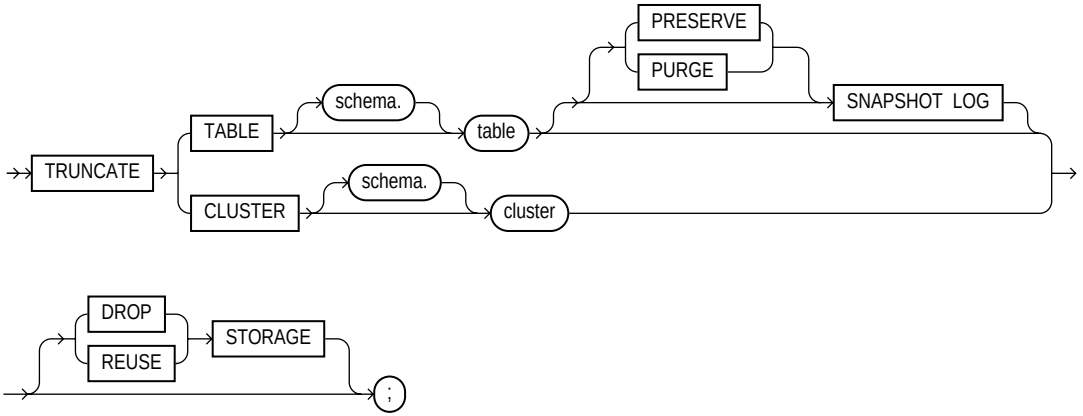
Purpose

To remove all rows from a table or cluster and reset the STORAGE parameters to the values when the table or cluster was created. See also "Truncating Tables and Clusters" on page 4-539. For illustrations, see "Examples" on page 4-540.

Prerequisites

The table or cluster must be in your schema or you must have DROP ANY TABLE system privilege. See also "Restrictions" on page 4-540.

Syntax



Keywords and Parameters

schema is the schema to contain the trigger. If you omit *schema*, Oracle creates the trigger in your own schema.

TABLE specifies the schema and name of the table to be truncated. You can truncate index-organized tables. This table cannot be part of a cluster.

When you truncate a table, Oracle also automatically deletes all data in the table's indexes.

SNAPSHOT LOG	specifies whether a snapshot log defined on the table is to be preserved or purged when the table is truncated. This clause allows snapshot master tables to be reorganized through export/import without affecting the ability of primary-key snapshots defined on the master to be fast refreshed. To support continued fast refresh of primary-key snapshots the snapshot log must record primary-key information. For more information about snapshot logs and the TRUNCATE command, see *Oracle8 Replication*.
PRESERVE	specifies that any snapshot log should be preserved when the master table is truncated. This is the default.
PURGE	specifies that any snapshot log should be purged when the master table is truncated.
CLUSTER	specifies the schema and name of the cluster to be truncated. You can only truncate an indexed cluster, not a hash cluster. When you truncate a cluster, Oracle also automatically deletes all data in the cluster's tables' indexes.
DROP STORAGE	deallocates the space from the deleted rows from the table or cluster. This space can subsequently be used by other objects in the tablespace. This is the default.
REUSE STORAGE	retains the space from the deleted rows allocated to the table or 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 table or cluster resulting from inserts or updates. The DROP STORAGE and REUSE STORAGE options also apply to the space freed by the data deleted from associated indexes.

Truncating Tables and Clusters

You can use the TRUNCATE command to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE command is faster than removing them with the DELETE command for the following reasons:

- The TRUNCATE command is a data definition language (DDL) command and generates no rollback information.
- Truncating a table does not fire the table's DELETE triggers.

The TRUNCATE command allows you to optionally deallocate the space freed by the deleted rows. The DROP STORAGE option deallocates all but the space specified by the table's MINEXTENTS parameter.

Deleting rows with the TRUNCATE command is also more convenient than dropping and re-creating a table because dropping and re-creating:

- invalidates the table's dependent objects, while truncating does not
- requires you to regrant object privileges on the table, while truncating does not

- requires you to re-create the table's indexes, integrity constraints, and triggers and respecify its STORAGE parameters, while truncating does not

> **Note:** When you truncate a table, the storage parameter NEXT is changed to be the size of the last extent deleted from the segment in the process of truncation.

Restrictions

When you truncate a table, NEXT is automatically reset to the last extent deleted.

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 referential integrity constraint. You must disable the constraint before truncating the table. (An exception is that you may truncate the table if the integrity constraint is self-referential.)

You cannot roll back a TRUNCATE statement.

Examples

Example I. The following statement deletes all rows from the EMP table and returns the freed space to the tablespace containing EMP:

```
TRUNCATE TABLE emp;
```

The above statement also deletes all data from all indexes on EMP and returns the freed space to the tablespaces containing them.

Example II. The following statement deletes all rows from all tables in the CUST cluster, but leaves the freed space allocated to the tables:

```
TRUNCATE CLUSTER cust
    REUSE STORAGE
```

The above statement also deletes all data from all indexes on the tables in CUST.

Example III. The following statements are examples of truncate statements that preserve snapshot logs:

```
TRUNCATE TABLE emp PRESERVE SNAPSHOT LOG;
TRUNCATE TABLE stock;
```

Related Topics

DELETE on page 4-374
DROP CLUSTER on page 4-386
DROP TABLE on page 4-405

UPDATE

Purpose

To change existing values in a table or in a view's base table.

> **Note:** Descriptions of commands and clauses preceded by OBJ are available only if the Oracle objects option is installed on your database server.

You can use comments in an UPDATE statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information, see *Oracle8 Tuning*.

You can place a parallel hint immediately after the UPDATE keyword to parallelize both the underlying scan and UPDATE operations. For detailed information about parallel DML, see *Oracle8 Tuning*, *Oracle8 Parallel Server Concepts and Administration*, and *Oracle8 Concepts*.

Prerequisites

For you to update values in a table, the table must be in your own schema or you must have UPDATE privilege on the table.

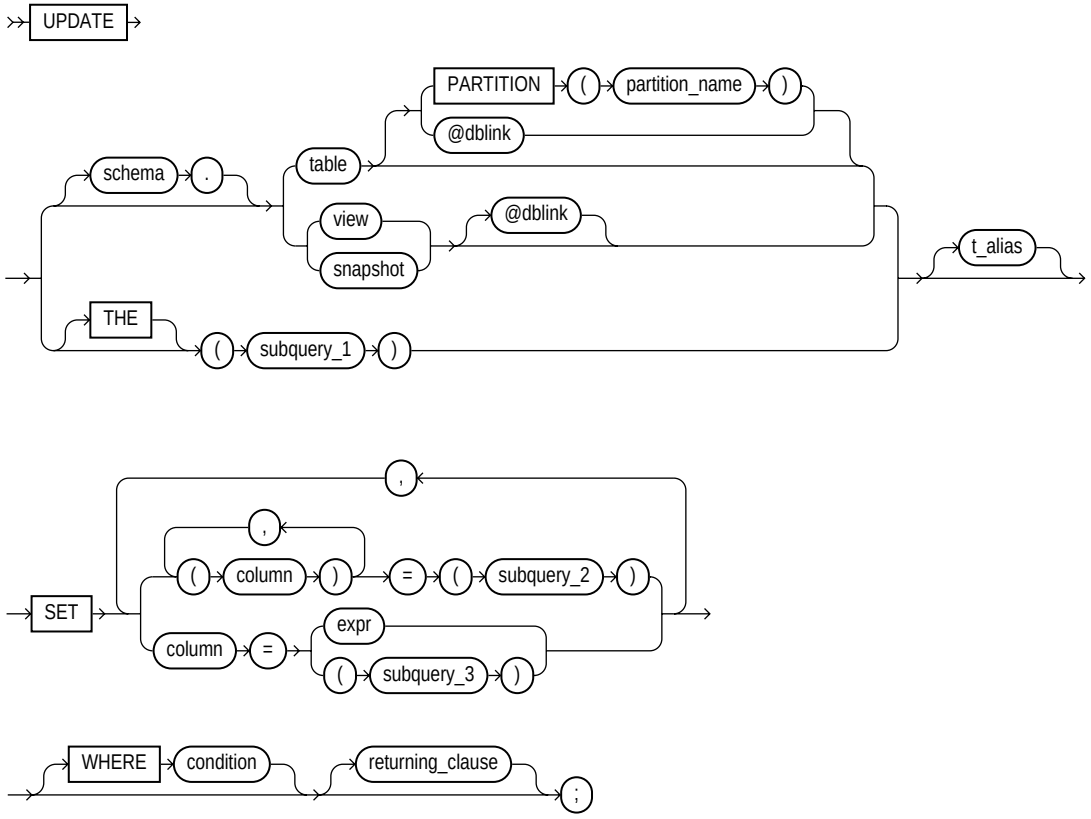
For you to update values in the base table of a view,

- You must have UPDATE privilege on the view, and
- Whoever owns the schema containing the view must have UPDATE privilege on the base table.

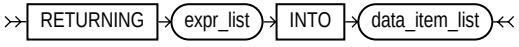
If the SQL92_SECURITY initialization parameter is set to TRUE, then you must have SELECT privilege on the table whose column values you are referencing (such as the columns in a WHERE clause) to perform an UPDATE.

The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

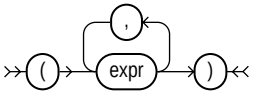
Syntax


UPDATE
schema
.
table
PARTITION
(
partition_name
)
@dblink
view
@dblink
snapshot
t_alias
THE
(
subquery_1
)
SET
,
(
column
)
=
(
subquery_2
)
column
=
expr
(
subquery_3
)
WHERE
condition
returning_clause
;

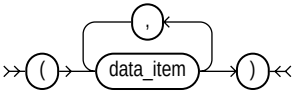
returning_clause ::=



expr_list ::=



data_item_list ::=



Keywords and Parameters

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table / view is the name of the table to be updated. Issuing an UPDATE statement against a table fires any UPDATE triggers associated with the table. If you specify *view*, Oracle updates the view's base table. See also "Updating Views" on page 4-545.

PARTITION *(partition_name)* specifies partition-level row updates for *table*. The *partition_name* parameter may be the name of the partition within table targeted for update, or a more complicated predicate restricting the update to just one partition. See also "Updating Partitioned Tables" on page 4-546.

dblink is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-54. You can use a database link to update a remote table or view only if you are using Oracle's distributed functionality.

If you omit *dblink*, Oracle assumes the table or view is on the local database.

OBJ THE informs Oracle that the column value returned by the subquery is a nested table, not a scalar value. A subquery prefixed by THE is called a *flattened subquery*. See also "Using Flattened Subqueries" on page 4-533.

subquery_1 is a subquery that Oracle treats in the same manner as a view. See also "Subqueries" on page 4-530.

t_alias provides a different name for the table, view, or subquery to be referenced elsewhere in the statement.

SET clause	determines which columns are updated and what new values are stored in them.
column	is the name of a column of the table or view that is to be updated. If you omit a column of the table from the SET clause, that column's value remains unchanged.
subquery_2	is a subquery that returns new values that are assigned to the corresponding columns. See also "Subqueries" on page 4-530.
expr	is the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables. See the syntax description in "Expressions" on page 3-78.
subquery_3	is a subquery that returns new values that are assigned to the corresponding columns. See also "Subqueries" on page 4-530 and "Correlated Update" on page 4-547.

If the SET clause contains a subquery, it must return exactly one row for each row updated. Each value in the subquery result is assigned respectively to the columns in the parenthesized list. If the subquery returns no rows, then the column is assigned a null. Subqueries may select from the table being updated.

The SET clause may mix assignments of expressions and subqueries.

WHERE	restricts the rows updated to those for which the specified *condition* is TRUE. If you omit this clause, Oracle updates all rows in the table or view. See the syntax description of "Conditions" on page 3-90. The WHERE clause determines the rows in which values are updated. If the WHERE clause is not specified, all rows are updated. For each row that satisfies the WHERE clause, the columns to the left of the equals (=) operator in the SET clause are set to the values of the corresponding expressions on the right. The expressions are evaluated as the row is updated.
returning_clause	retrieves the rows affected by the UPDATE statement. You can only retrieve scalar, LOB, ROWID, and REF types. See also "The RETURNING Clause" on page 4-549.
expr_list	is some of the syntax descriptions in "Expressions" on page 3-78. You must specify a column expression in the *expr_list* for each variable in the *data_item_list*.
INTO	indicates that the values of the changed rows are to be stored in the *data_item* variable(s) specified in *data_item_list*.
data_item	is a PL/SQL variable or bind variable which stores the retrieved *expr* value in the *expr_list*.

You cannot use the *returning_clause* with parallel DML or with remote objects.

Updating Views

If a view was created with the WITH CHECK OPTION, you can update the view only if the resulting data satisfies the view's defining query.

You cannot update a view if the view's defining query contains one of the following constructs:

- set operator
- GROUP BY clause
- group function
- DISTINCT operator
- flattened subqueries
- nested table columns
- CAST and MULTISET expressions

Updating Partitioned Tables

When you create a partitioned table, you specify an ordered list of columns that determines into which partition a row or index entry belongs. These columns are the *partitioning columns*. The values in the partitioning columns of a row are the *partitioning key* for that row.

```
CREATE TABLE emp
  (emp_no NUMBER(5),
   dept VARCHAR2(2),
   name VARCHAR2 (30))
  STORAGE (INITIAL 100K NEXT 50K) LOGGING
  PARTITION BY RANGE (emp_no)
   ( PARTITION acct VALUES LESS THAN (1000)
     TABLESPACE ts1,
    PARTITION sales VALUES LESS THAN (2000)
     TABLESPACE ts2
     PARTITION educ VALUES LESS THAN (3000) );

INSERT INTO EMP VALUES (1226, 'sa', 'smith');

INSERT INTO EMP VALUES (2100, 'ed', 'jones');
```

In the following example, employee SMITH is updated in the EMP table:

```
UPDATE emp SET emp_no = 1356
  WHERE name = 'SMITH';
```

The following statement is rejected because updating the row would cause JONES to move to another partition:

```
UPDATE emp SET emp_no = 1500
  WHERE name = 'JONES';
```

Attempting to change the value of one or more columns that are part of the partitioning key would cause the updated row to migrate to another partition, thereby generating an error.

Updating a Single Partition

You do not need to 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. To target a single partition of a partitioned table whose values you want to change, specify the PARTITION clause. This syntax can be less cumbersome than using a WHERE clause.

Example. The following example updates values in a single partition of the SALES table:

```
UPDATE sales PARTITION (feb96) s
   SET s.account_name = UPPER(s.account_name);
```

Correlated Update

If a subquery refers to columns from the updated table, Oracle evaluates the subquery once for each row, rather than once for the entire update. Such an update is called a correlated update. The reference to columns from the updated table is usually accomplished by means of a table alias.

Potentially, each row evaluated by an UPDATE statement could be updated with a different value as determined by the correlated subquery. Normal UPDATE statements update each row with the same value.

Example I. The following statement gives null commissions to all employees with the job TRAINEE:

```
UPDATE emp
    SET comm = NULL
    WHERE job = 'TRAINEE';
```

Example II. The following statement promotes JONES to manager of Department 20 with a $1,000 raise (assuming there is only one JONES):

```
UPDATE emp
    SET job = 'MANAGER', sal = sal + 1000, deptno = 20
    WHERE ename = 'JONES';
```

Example III. The following statement increases the balance of bank account number 5001 in the ACCOUNTS table on a remote database accessible through the database link BOSTON:

```
UPDATE accounts@boston
    SET balance = balance + 500
    WHERE acc_no = 5001;
```

Example IV. This example shows the following syntactic constructs of the UPDATE command:

- both forms of the SET clause together in a single statement
- a correlated subquery
- a WHERE clause to limit the updated rows

```
UPDATE emp a
    SET deptno =
        (SELECT deptno
            FROM dept
            WHERE loc = 'BOSTON'),
        (sal, comm) =
        (SELECT 1.1*AVG(sal), 1.5*AVG(comm)
          FROM emp b
          WHERE a.deptno = b.deptno)
    WHERE deptno IN
        (SELECT deptno
          FROM dept
          WHERE loc = 'DALLAS'
              OR loc = 'DETROIT');
```

The above UPDATE statement performs the following operations:

- updates only those employees who work in Dallas or Detroit
- sets DEPTNO for these employees to the DEPTNO of Boston
- 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

Example V. OBJ The following example updates particular rows of the PROJS table:

```
UPDATE THE(SELECT projs
           FROM dept d WHERE d.dno = 123)  p
  SET p.budgets = p.budgets + 1
```

```
  WHERE p.pno IN (123, 456);
```

The RETURNING Clause

You can use a RETURNING clause to return values from updated columns, and thereby eliminate the need to perform a SELECT following the UPDATE statement.

- When you are updating a single row, an UPDATE statement with a RETURNING clause can retrieve column expressions that use the updated columns of the row, ROWID, and REFs to the updated row and store them in PL/SQL variables or bind variables.
- When you are using an UPDATE statement with the RETURNING clause to update multiple rows, the values from expressions, ROWID, and REFs involving the updated rows are stored in bind arrays.

You can also use UPDATE with a RETURNING clause to update from views with single base tables.

Example. The following example returns values from the updated row and stores the result in PL/SQL variables BND1, BND2, BND3:

```
UPDATE emp
  SET job ='MANAGER', sal = sal + 1000, deptno = 20
  WHERE ename = 'JONES'
  RETURNING sal*0.25, ename, deptno INTO bnd1, bnd2, bnd3;
```

Related Topics

DELETE on page 4-374
INSERT on page 4-451

A

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. Operators, delimiters, and terminators appear inside circles.

If the syntax diagram has more than one path, you can choose any path to travel.

If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list.

Required Keywords and Parameters

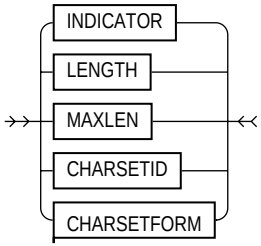
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *cursor* is a required parameter:

If there is a library named *HQ_LIB*, then, according to the diagram, the following statement is valid:

```
DROP LIBRARY hq_lib;
```

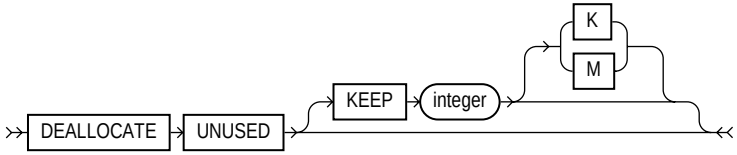
If multiple keywords or parameters appear in a vertical list that intersects the main path, one of them is required. That is, you must choose one of the keywords or

parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the five settings:



Optional Keywords and Parameters

If keywords and parameters appear in a vertical list *above* the main path, they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:

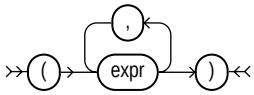


According to the diagram, all of the following statements are valid:

```
DEALLOCATE UNUSED;
DEALLOCATE UNUSED KEEP 1000;
DEALLOCATE UNUSED KEEP 10 M;
```

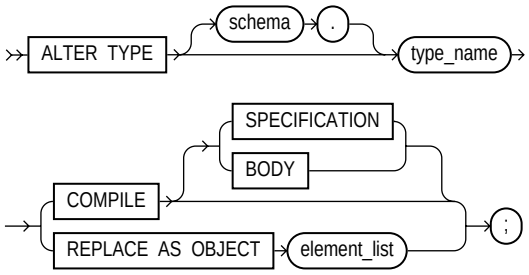
Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, after choosing one expression, you can go back repeatedly to choose another, separated by commas.



Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end to end. The following example is a two-part diagram:



According to the diagram, the following statement is valid:

```
ALTER TYPE type_name COMPILE BODY;
```

Database Objects

The names of Oracle identifiers, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an Oracle identifier is enclosed by double quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle identifiers are not case-sensitive except when enclosed by double quotation marks.

B

Oracle and Standard SQL

This appendix discusses the following topics:

- Oracle's conformance to the SQL standards established by industry standards governing bodies
- Oracle's extensions to standard SQL
- Locating extensions to standard SQL with the FIPS Flagger

Conformance with Standard SQL

This section declares Oracle's conformance to the SQL standards established by these organizations:

- American National Standards Institute (ANSI)
- International Standards Organization (ISO)
- United States Federal Government

Conformance with these standards is measured by the National Institute of Standards and Technology (NIST) "SQL Test Suite". NIST is an organization of the government of the United States of America.

ANSI and ISO Compliance

Oracle8 conforms to Entry level conformance defined in the ANSI document, X3.135-1992, "Database Language SQL." You can obtain a copy of the ANSI standard from this address:

American National Standards Institute
1430 Broadway
New York, NY 10018 USA

The ANSI and ISO SQL standards require conformance claims to state the type of conformance and the implemented facilities. The Oracle8 server, the Oracle Precompiler for Fortran Version 1.8.25, Oracle Precompilers for C / C++ Version 8.0.4, Oracle Precompiler for Cobol Version 8.0.4, and SQL*Module for ADA Version 8.0.4 provide conformance with the ANSI X3.135-1992 / ISO 9075-1992 standard:

- Compliance at Entry Level (including both SQL-DDL and SQL-DML)
- Module Language for ADA
- Embedded SQL C
- Embedded SQL COBOL
- Embedded SQL FORTRAN

FIPS Compliance

Oracle complies completely with FIPS PUB 127-2 for Entry SQL. In addition, the following information is provided for Section 16, "Special Procurement Considerations."

Section 16.2 Programming Language Interfaces

The Oracle precompilers support the use of embedded SQL in C, COBOL, and Fortran. SQL*Module supports the use of Module Language in ADA.

Section 16.3 Style of Language Interface

Oracle with SQL*Module supports Module Language for Ada. Oracle with the Oracle precompilers supports C, COBOL, and FORTRAN. The specific languages supported depend on your operating system.

Section 16.5 Interactive Direct SQL

Oracle8 with SQL*Plus Version 3.1 (as well as other Oracle tools) supports "direct invocation" of the following SQL commands, meeting the requirements of FIPS PUB 127-2:

- CREATE TABLE command
- CREATE VIEW command
- GRANT command
- INSERT command

- SELECT command, with ORDER BY clause but not INTO clause
- UPDATE command: searched
- DELETE command: searched
- COMMIT WORK command
- ROLLBACK WORK command

Most other SQL commands described in this guide are also supported interactively.

Section 16.6 Sizing for Database Constructs

Table B–1 lists requirements identified in FIPS PUB 127-1 and how they are met by Oracle8.

Table B–1 Sizing for Database Constructs

Database Constructs	FIPS	Oracle8
Length of an identifier (in bytes)	18	30
Length of CHARACTER datatype (in bytes)	240	2000
Decimal precision of NUMERIC datatype	15	38
Decimal precision of DECIMAL datatype	15	38
Decimal precision of INTEGER datatype	9	38
Decimal precision of SMALLINT datatype	4	38
Binary precision of FLOAT datatype	20	126
Binary precision of REAL datatype	20	63
Binary precision of DOUBLE PRECISION datatype	30	126
Columns in a table	100	1000
Values in an INSERT statement	100	1000
Set clauses in an UPDATE statement(a)	20	1000
Length of a row(b,c)	2,000	2,000,000
Columns in a UNIQUE constraint	6	16
Length of a UNIQUE constraint(b)	120	(d)
Length of foreign key column list(b)	120	(d)

Table B–1 (Cont.) Sizing for Database Constructs

Columns in a GROUP BY clause	6	255(e)
Length of GROUP BY column list	120	(e)
Sort specifications in ORDER BY clause	6	255(e)
Length of ORDER BY column list	120	(e)
Columns in a referential integrity constraint	6	16
Tables referenced in a SQL statement	15	No limit
Cursors simultaneously open	10	(f)
Items in a SELECT list	100	1000

(a) The number of set clauses in an UPDATE statement refers to the number items separated by commas following the SET keyword.

(b) 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.

(c) 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)).

(d) 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.

(e) 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 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.

(f) 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.

Section 16.7 Character Set Support

Oracle supports the ASCII character set (FIPS PUB 1-2) on most computers and the EBCDIC character set on IBM mainframe computers. Oracle supports both single-byte and multibyte character sets.

Extensions to Standard SQL

This section lists the additional features supported by Oracle that extend beyond standard SQL “Database Language SQL”. This section provides information on these parts of the SQL language:

- commands
- functions

- operators
- pseudocolumns
- datatypes
- names of schema objects
- values

For information on the extensions to standard embedded SQL "Database Language Embedded SQL" supported by the Oracle Precompilers, see *Pro*COBOL Precompiler Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *SQL*Module for Ada Programmer's Guide.*

Commands

This section describes these additional commands and additional syntax and functionality of standard commands. Oracle supports these commands that are not part of Entry SQL92:

```
ALTER CLUSTER
ALTER DATABASE
ALTER FUNCTION
ALTER INDEX
ALTER PACKAGE
ALTER PROCEDURE
ALTER PROFILE
ALTER RESOURCE COST
ALTER ROLLBACK SEGMENT
ALTER ROLE
ALTER SEQUENCE
ALTER SESSION
ALTER SNAPSHOT
ALTER SNAPSHOT LOG
ALTER SYSTEM
ALTER TABLE
ALTER TABLESPACE
ALTER TRIGGER
ALTER TYPE
ALTER USER
ALTER VIEW
ANALYZE
AUDIT

COMMENT
CREATE CONTROLFILE
CREATE CLUSTER
CREATE DATABASE
CREATE DATABASE LINK
CREATE DIRECTORY
CREATE FUNCTION
CREATE INDEX
CREATE LIBRARY
CREATE PACKAGE
CREATE PACKAGE BODY
CREATE PROCEDURE
CREATE PROFILE
CREATE ROLLBACK SEGMENT
CREATE ROLE
```

```
CREATE SEQUENCE
CREATE SNAPSHOT
CREATE SNAPSHOT LOG
CREATE SYNONYM
CREATE TABLE
CREATE TABLESPACE
CREATE TRIGGER
CREATE TYPE
CREATE TYPE BODY
CREATE USER
CREATE VIEW

DROP CLUSTER
DROP DATABASE LINK
DROP DIRECTORY
DROP FUNCTION
DROP INDEX
DROP LIBRARY
DROP PACKAGE
DROP PROCEDURE
DROP PROFILE
DROP ROLLBACK SEGMENT
DROP ROLE
DROP SEQUENCE
DROP SNAPSHOT
DROP SNAPSHOT LOG
DROP SYNONYM
DROP TABLE
DROP TABLESPACE
DROP TYPE
DROP TYPE BODY

EXPLAIN PLAN

NOAUDIT

RENAME
REVOKE

SAVEPOINT
SET CONSTRAINT
SET TRANSACTION

TRUNCATE
```

Additional Parts of Standard Commands

Oracle supports additional syntax for some commands that are part of Entry SQL92.

COMMIT

The COMMIT command supports these additional clauses:

- COMMENT clause
- FORCE clause

Also, Entry SQL92 requires a COMMIT statement to include the WORK keyword. Oracle allows your COMMIT statements to either include or omit this keyword. This keyword adds no functionality to the command.

CREATE TABLE

The CREATE TABLE command supports these additional parameters and clauses:

- AS clause
- ENABLE clause
- DISABLE clause
- CLUSTER clause
- INITRANS parameter
- MAXTRANS parameter
- ORGANIZATION clause
- PCTFREE parameter
- PCTTHRESHOLD parameter
- PCTUSED parameter
- STORAGE clause
- TABLESPACE parameter

CONSTRAINT Clause The CONSTRAINT clause of the CREATE TABLE command supports these additional options and identifiers:

- ON DELETE CASCADE option
- ENABLE option
- DISABLE option

- CONSTRAINT identifier

Column definitions in a CREATE TABLE command support these additional clauses:

- WITH ROWID
- SCOPE

In addition, columns may be defined using any Oracle predefined type, not just the Entry SQL92 datatypes. Oracle's extended datatypes are noted below. If a column's datatype is BLOB, CLOB, or NCLOB, then special LOB storage and index features can be specified in a CREATE TABLE command.

CREATE VIEW

The CREATE VIEW command supports this additional syntax:

- OR REPLACE option
- FORCE and NOFORCE options
- CONSTRAINT identifier with the WITH CHECK OPTION

If you omit column names from a CREATE VIEW statement, the column aliases that appear in the defining query are used for columns of the view.

DELETE

The DELETE command supports this additional syntax:

- Database links to delete rows from tables and views on remote databases
- Table aliases for use with correlated queries
- PARTITION clause
- RETURNING clause

Also, Entry SQL92 requires a DELETE statement to include the FROM keyword. Oracle allows your DELETE statements to either include or omit this keyword. Note that this keyword adds no functionality to the command.

Oracle allows a DELETE command against a modifiable join view with exactly one key-preserved table in the join; SQL92 does not allow DELETE against a join view.

GRANT

The GRANT command (System Privileges and Roles) is an extension to standard SQL.

The GRANT command (Object Privileges) supports the following other privileges on other objects in addition to the DELETE, INSERT, REFERENCES, SELECT, and UPDATE privileges on tables and views supported by Entry SQL92:

- ALTER
- EXECUTE
- INDEX
- READ

This command also supports granting object privileges to roles.

INSERT

The INSERT command supports the use of database links to insert rows into tables and views on remote databases. The INSERT command supports this additional syntax:

- PARTITION clause
- RETURNING clause

The INSERT command supports a subquery in the INTO clause, similar to inserting into a view.

The INSERT command can insert into a modifiable join view that does not specify the WITH CHECK OPTION provided that all columns to be inserted are in the same key-preserved table of the join.

ROLLBACK

The ROLLBACK command supports these additional clauses:

- TO clause
- FORCE clause

Also, Entry SQL92 requires a ROLLBACK statement to include the WORK keyword. Oracle allows your ROLLBACK statements to either include or omit this keyword. This keyword adds no functionality to the command.

SELECT

The SELECT command supports these additional clauses and syntax:

- START WITH clause
- CONNECT BY clause
- FOR UPDATE clause
- Database links for querying tables, views, and snapshots on remote databases
- Outer join operator (+) for performing outer joins
- NULL in the select list

GROUP BY Clause The GROUP BY clause of the SELECT command supports this additional syntax and functionality:

- A SELECT statement that selects from a view whose defining query contains group functions or a GROUP BY clause can contain group functions and GROUP BY, HAVING, and WHERE clauses.
- A SELECT statement can perform a join involving a view whose defining query contains a GROUP BY clause.

ORDER BY Clause The ORDER BY clause of the SELECT command supports this additional syntax and functionality:

- This clause can also specify any expression involving any columns in any tables or views that appear in the FROM clause, rather than only names, aliases, or position numbers of columns in the select list.
- This clause can qualify a column name with its table or view name, using the syntax *table.column* or *view.column*.

Subqueries Subqueries (i.e., forms of the SELECT command that appear inside other SQL statements), support this additional functionality:

- Subqueries can contain the GROUP BY clause.
- Subqueries can select from views whose defining queries contain the GROUP BY clause.

UPDATE

The UPDATE command supports this additional syntax:

- Database links to update data in tables and views on remote databases
- Table aliases for use with correlated subqueries

- Parenthesized lists of columns on the left side of the SET clause, rather than only single columns
- Subqueries on the right side of the SET clause, rather than only expressions
- PARTITION clause
- RETURNING clause

The UPDATE command also supports this additional functionality:

- An UPDATE statement may update a modifiable join view provided that all columns to be updated are in the same key-preserved table of the join. If the view specifies WITH CHECK OPTION, then join columns cannot be modified.
- A subquery within the SET clause or WHERE clause of an UPDATE statement can refer to the table or view being updated.
- A view containing columns that are defined as complex expressions (i.e., not simply as a column of a table in the FROM clause, but also containing functions or operators) can be updated.
- The UPDATE command supports updating a subquery.

Functions

This section describes additional functions and additional functionality of standard functions.

Additional Functions

The only Entry SQL92 functions are AVG, COUNT, MAX, MIN, and SUM. Oracle supports many additional functions that are not part of Entry SQL92. See “SQL Functions” on page 3-16.

Additional Functionality of Standard Functions

You can nest group functions in the select list of a SELECT statement, as in this example:

```
SELECT MIN(MAX(sal))
FROM emp
GROUP BY deptno;
```

The depth of nesting cannot be more than that shown in the example.

You can also use a group function in a SELECT statement that queries a view whose defining query contains group functions or a GROUP BY clause.

Operators

This section describes additional operators and additional functionality of standard operators.

Additional Operators

Oracle supports these operators that are not part of Entry SQL92:

- || character operator (character concatenation)
- !=, ^=, and ¬= comparison operators (inequality)
- MINUS set operator
- INTERSECT set operator
- (+) operator (outer join)
- PRIOR operator

Additional Functionality of Standard Operators

Oracle supports additional functionality for Entry SQL92 operators:

- The left member of an expression containing the IN operator can be a parenthesized list of expressions, rather than only a single expression.
- Any expression, rather than only a column, can be used with the comparison operators IS NULL and IS NOT NULL.
- The pattern used with the LIKE operator can be any expression of datatype CHAR or VARCHAR2, rather than only a text literal.

Pseudocolumns

Pseudocolumns are values that behave like columns of a table but are not actually stored in the table. Pseudocolumns are supported by Oracle, but are not part of Entry SQL92. For a list of pseudocolumns, see “Pseudocolumns” on page 2-32.

Datatypes

Oracle supports these additional datatypes that are not part of Entry SQL92:

- DATE (Note: Oracle’s DATE datatype is different from the DATE data type in Intermediate SQL92.)
- NUMBER
- VARCHAR2

- LONG
- RAW
- LONG RAW
- ROWID
- BLOB
- CLOB
- BFILE
- NCLOB

Additionally, Oracle supports the following user-defined types that are not part of Entry SQL92:

- object
- REF
- collection (VARRAY and nested table)

Oracle also supports automatic conversion of values from one datatype to another that is not part of Entry SQL92.

Names of Schema Objects

Oracle supports additional functionality for names of schema objects:

- Oracle supports names of maximum length 30 bytes, rather than 18 Oracle supports names that contain the special characters # and $ and repeated underscores (__).

Values

Oracle allows you to use either uppercase “E” or lowercase “e” for exponential notation of numeric values, rather than only “E”.

FIPS Flagger

In your Oracle applications, you can use the extensions listed in the previous sections just as you can use Entry SQL92. If you are concerned with the portability of your applications to other implementations of SQL, use Oracle’s FIPS Flagger to locate Oracle extensions to Entry SQL92 in your embedded SQL programs. The FIPS Flagger is part of the Oracle precompilers and the SQL*Module compiler. For information on how to use the FIPS Flagger, see *Pro*COBOL Precompiler*

Programmer's Guide, *Pro*C/C++ Precompiler Programmer's Guide*, and *SQL*Module for Ada Programmer's Guide.*

C

Oracle Reserved Words and Keywords

This appendix lists Oracle reserved words and keywords.

Table C–1, “Reserved Words” lists Oracle reserved words. Words followed by an asterisk (*) are also ANSI reserved words.

> **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.

Table C–1 Reserved Words

ACCESS	AUDIT	COMPRESS	DESC*
ADD*	BETWEEN*	CONNECT*	DISTINCT*
ALL*	BY*	CREATE*	DROP*
ALTER*	CHAR*	CURRENT*	ELSE*
AND*	CHECK*	DATE*	EXCLUSIVE
ANY*	CLUSTER	DECIMAL*	EXISTS
AS*	COLUMN	DEFAULT*	FILE
ASC*	COMMENT	DELETE*	FLOAT*

Table C–1 Reserved Words

FOR*	LONG	PCTFREE	SUCCESSFUL
FROM*	MAXEXTENTS	PRIOR*	SYNONYM
GRANT*	MINUS	PRIVILEGES*	SYSDATE
GROUP*	MODE	PUBLIC*	TABLE*
HAVING*	MODIFY	RAW	THEN*
IDENTIFIED	NETWORK	RENAME	TO*
IMMEDIATE*	NOAUDIT	RESOURCE	TRIGGER
IN*	NOCOMPRESS	REVOKE*	UID
INCREMENT	NOT*	ROW	UNION*
INDEX	NOWAIT	ROWID	UNIQUE*
INITIAL	NULL*	ROWNUM	UPDATE*
INSERT*	NUMBER	ROWS*	USER*
INTEGER*	OF*	SELECT*	VALIDATE
INTERSECT*	OFFLINE	SESSION*	VALUES*
INTO*	ON*	SET*	VARCHAR*
IS*	ONLINE	SHARE	VARCHAR2
LEVEL*	OPTION*	SIZE*	VIEW*
LIKE*	OR*	SMALLINT*	WHENEVER*
LOCK	ORDER*	START	WHERE
			WITH*

Table C–2, " Keywords" lists Oracle keywords. Keywords marked with asterisks (*) are also ANSI reserved words. For maximum portability to other implementations of SQL, do not use these words as schema object names.

Table C–2 Keywords

ACCOUNT	CACHE_INSTANCES	CONSTRAINT*	DEGREE
ACTIVATE	CANCEL	CONSTRAINTS*	DEREF
ADMIN	CASCADE*	CONTENTS	DIRECTORY
AFTER	CAST	CONTINUE*	DISABLE
ALLOCATE*	CFILE	CONTROLFILE	DISCONNECT
ALL_ROWS	CHAINED	CONVERT*	DISMOUNT
ANALYZE	CHANGE	COST	DISTRIBUTED
ARCHIVE	CHARACTER*	COUNT*	DML
ARCHIVELOG	CHAR_CS	CPU_PER_CALL	DOUBLE*
ARRAY	CHECKPOINT	CPU_PER_SESSION	DUMP
AT*	CHOOSE	CURRENT_SCHEMA	
AUTHENTICATED	CHUNK	CURRENT_USER*	EACH
AUTHORIZATION	CLEAR	CURSOR*	ENABLE
AUTOEXTEND	CLOB	CYCLE	END*
AUTOMATIC	CLONE		ENFORCE
	CLOSE*	DANGLING	ENTRY
BACKUP	CLOSED_CACHED_OPEN_ CURSORS	DATABASE	ESCAPE*
BECOME	COALESCE*	DATAFILE	ESTIMATE
BEFORE	COLUMNS	DATAFILES	EVENTS
BEGIN*	COMMIT*	DATAOBJNO	EXEMPT*
BFILE	COMMITTED	DBA	EXCEPTIONS
BITMAP	COMPATIBILITY	DEALLOCATE*	EXCHANGE
BLOB	COMPILE	DEBUG	EXCLUDING
BLOCK	COMPLETE	DEC*	EXECUTE*
BODY	COMPOSITE_LIMIT	DECLARE*	EXPIRE
	COMPUTE	DEFERRABLE	EXPLAIN
CACHE	CONNECT_TIME	DEFERRED	EXTENT

Table C–2 (Cont.) Keywords

EXTENTS	INDEXES	MANAGE	NLS_CALENDAR
EXTERNALLY	INDICATOR*	MASTER	NLS_CHARACTERSET
	IND_PARTITION	MAX*	NLS_ISO_CURRENCY
FAILED_LOGIN_ATTEMPTS	INITIALLY	MAXARCHLOGS	NLS_LANGUAGE
FALSE	INITRANS	MAXDATAFILES	NLS_NUMERIC_ CHARACTERS
FAST	INSTANCE	MAXINSTANCES	NLS_SORT
FIRST_ROWS	INSTANCES	MAXLOGFILES	NOS_SPECIAL_CHARS
FLAGGER	INSTEAD	MAXLOGHISTORY	NLS_TERRITORY
FLUSH	INT*	MAXLOGMEMBERS	NOARCHIVELOG
FORCE	INTERMEDIATE	MAXSIZE	NOCACHE
FOREIGN*	ISOLATION*	MAXTRANS	NOCYCLE
FREELIST	ISOLATION_LEVEL	MAXVALUE	NOFORCE
FREELISTS		MEMBER	NOLOGGING
FULL	KEEP	MIN*	NOMAXVALUE
FUNCTION	KEY*	MINEXTENTS	NOMINVALUE
	KILL	MINIMUM	NONE
GLOBAL*		MINVALUE	NOORDER
GLOBALLY	LAYER	MOUNT	NOOVERIDE
GLOBAL_NAME	LESS	MOVE	NOPARALLEL
GROUPS	LIBRARY	MTS_DISPATCHERS	NORESETLOGS
HASH	LIMIT	MULTISET	NOREVERSE
HASHKEYS	LINK		NORMAL
HEADER	LIST	NATIONAL*	NOSORT
INSTANCE	LOB	NCHAR*	NOTHING
HEAP	LOCAL*	NCHAR_CS	NUMERIC*
	LOG	NCLOB	NVARCHAR2
IDLE_TIME	LOGFILE	NEEDED	OBJECT
IF	LOGGING	NESTED	OBJNO
INCLUDING	LOGICAL_READS_PER_CALL	NEW	OBJNO_REUSE
INDEXED	LOGICAL_READS_PER_ SESSION	NEXT*	OFF

Table C–2 (Cont.) Keywords

OID	PLSQL_DEBUG	RESIZE	SHARED_POOL
OIDINDEX	POST_TRANSACTION	RESTRICTED	SHRINK
OLD	PRECISION*	RETURN	SKIM_UNUSABLE_INDEXES
ONLY*	PRESERVE*	RETURNING	SNAPSHOT
OPCODE	PRIMARY*	REUSE	SOME*
OPEN*	PRIVATE	REVERSE	SORT
OPTIMAL	PRIVATE_SGA	ROLE	SPECIFICATION
OPTIMIZER_GOAL	PRIVILEGE	ROLES	SPLIT
ORGANIZATION	PROCEDURE*	ROLLBACK*	SQLCODE*
OVERFLOW	PROFILE	ROWLABEL	SQLERROR*
OWN	PURGE	RULE	SQL_TRACE
			STANDBY
PACKAGE	QUEUE	SAMPLE	STATEMENT_ID
PARALLEL	QUOTA	SAVEPOINT	STATISTICS
PARTITION		SCAN_INSTANCES	STOP
PASSWORD	RANGE	SCHEMA*	STORAGE
PASSWORD_GRACE_TIME	RBA	SCN	STORE
PASSWORD_LIFE_TIME	READ*	SCOPE	STRUCTURE
PASSWORD_LOCK_TIME	REAL*	SD_ALL	SUM*
PASSWORD_REUSE_MAX	REBUILD	SD_INHIBIT	SWITCH
PASSWORD_REUSE_TIME	RECOVER	SD_SHOW	SYSDBA
PASSWORD_VERIFY_ FUNCTION	RECOVERABLE	SEGMENT	SYSOPER
PCTINCREASE	RECOVERY	SEG_BLOCK	SYSTEM
PCTTHRESHOLD	REF	SEG_FILE	
PCTUSED	REFERENCES*	SEQUENCE	TABLES
PCTVERSION	REFERENCING	SERIALIZABLE	TABLESPACE
PERCENT	REFRESH	SESSIONS_PER_USER	TABLESPACE_NO
PERMANENT	REPLACE	SESSION_CACHED_ CURSORS	TABNO
PLAN	RESET	SHARED	TEMPORARY*
	RESETLOGS		

Table C–2 (Cont.) Keywords

THAN	TRUE	UNLOCK	VALIDATION
THE	TRUNCATE	UNRECOVERABLE	VALUE
THREAD	TX	UNTIL*	VARRAY
TIME	TYPE	UNUSABLE	VARYING*
TIMESTAMP		UNUSED	WHEN
TOPLEVEL	UBA	UPDATABLE	WITHOUT
TRACE	UNARCHIVED	USAGE*	WORK*
TRACING	UNDER	USE	WRITE*
TRANSACTION*	UNDO	USING*	
TRANSITIONAL	UNLIMITED		XID
TRIGGERS			

Index

Symbols

!= comparison operator, 3-5
$ format element, 3-65
% character
  in pattern matching, 3-9
(+) outer join operator, 3-16
, (comma) format element, 3-65
. (period) number format element, 3-65
< comparison operator, 3-5
<= comparison operator, 3-5
= comparison operator, 3-5
> comparison operator, 3-5
>= comparison operator, 3-5
^= comparison operator, 3-5
_ character
  in pattern matching, 3-9
” double quotation marks
  with object names, 2-49

Numerics

0 format element, 3-65
9 format element, 3-65

A

ABS numeric function, 3-18
AD / A.D. format element, 3-72
ADD clause
  of ALTER SNAPSHOT LOG command, 4 - 86
  of ALTER TABLE command, 4 - 116
ADD DATAFILE clause
  of ALTER TABLESPACE command, 4 - 136
ADD LOGFILE clause
  of ALTER DATABASE command, 4 - 19
ADD LOGFILE MEMBER clause
  of ALTER DATABASE command, 4 - 20
ADD OVERFLOW option
  of ALTER TABLE command, 4 - 117
ADD PARTITION option
  of ALTER SNAPSHOT command, 4 - 79, 4 - 85
  of ALTER TABLE command, 4 - 121
add_column_options_clause
  of ALTER TABLE, 4 - 109
ADD_MONTHS date function, 3-36
add_overflow_clause
  of ALTER TABLE, 4 - 113
add_partition_clause
  of ALTER TABLE, 4 - 114
adding
  columns to tables, 4 - 106, 4 - 116, 4 - 122
  comments to objects, 4 - 183
  datafiles, 4 - 136
  datafiles to databases, 4 - 219, 4 - 223
  datafiles to tablespaces, 4 - 329
  integrity constraints to columns, 4 - 116, 4 - 123
  integrity constraints to tables, 4 - 116, 4 - 123
  members to redo log file groups, 4 - 20
  procedures to packages, 4 - 251
  redo log file groups to threads, 4 - 19
  redo log files to databases, 4 - 219, 4 - 221
  REF columns to tables, 4 - 127
  resource limits to profiles, 4 - 43, 4 - 265
  stored functions to packages, 4 - 251
  tables to clusters, 4 - 213, 4 - 318
  triggers to tables, 4 - 333
ADMIN OPTION

of GRANT command, 4 - 441
ADVISE clause
of ALTER SESSION command, 4 - 62
AFTER option
of CREATE TRIGGER command, 4 - 335
alias
table, 4 - 547
ALL comparison operator, 3-5
ALL option
of ARCHIVE LOG clause, 4 - 168
of SELECT command, 4 - 491
of SET ROLE command, 4 - 516
of SQL group functions, 3-57
ALL PRIVILEGES option
of GRANT command, 4 - 445
of REVOKE command, 4 - 479
ALL statement auditing short cut, 4 - 177
ALL TRIGGERS option
of DISABLE clause, 4 - 381
of ENABLE clause, 4 - 419
ALL_INDEXES view, 4 - 161
ALL_TAB_COLUMNS view, 4 - 163
ALL_TABLES view, 4 - 161
ALLOCATE EXTENT clause
of ALTER CLUSTER command, 4 - 12
of ALTER TABLE command, 4 - 118
allocate_extent_clause
of ALTER CLUSTER, 4 - 12
of ALTER INDEX, 4 - 31
of ALTER TABLE, 4 - 112
allocating
extents for tables, 4 - 106
ALTER CLUSTER command, 4 - 11
allocate_extent_clause, 4 - 12
deallocate_unused_clause. See DEALLOCATE UNUSED Clause.
physical_attributes_clause, 4 - 11
ALTER DATABASE command, 4 - 15
autoextend_clause, 4 - 18
examples, 4 - 23, 4 - 24, 4 - 25, 4 - 471
recover_clause. See RECOVER Clause.
ALTER FUNCTION command, 4 - 26
examples, 4 - 27
ALTER INDEX command, 4 - 28
allocate_extent_clause, 4 - 31
deallocate_unused_clause. See DEALLOCATE UNUSED clause.
examples, 4 - 35
index_physical_attributes_clause, 4 - 30
parallel_clause. See PARALLEL Clause.
partition_description_clause, 4 - 31
split_partition_clause, 4 - 31
storage_clause. See STORAGE clause.
ALTER object auditing option, 4 - 174
ALTER object privilege
on sequences, 4 - 448
on tables, 4 - 447
ALTER PACKAGE command
examples, 4 - 39, 4 - 40
ALTER PROCEDURE command, 4 - 41
examples, 4 - 42
ALTER PROFILE command, 4 - 43
examples, 4 - 46, 4 - 47
ALTER RESOURCE COST command, 4 - 48
examples, 4 - 49
ALTER ROLE command, 4 - 51
examples, 4 - 52
ALTER ROLLBACK SEGMENT command, 4 - 53
examples, 4 - 55
storage_clause. See STORAGE clause.
ALTER SEQUENCE command, 4 - 56
examples, 4 - 57
ALTER SESSION command, 4 - 58
examples, 4 - 67, 4 - 68, 4 - 69, 4 - 70, 4 - 73
ALTER SNAPSHOT command, 4 - 76
add_partition_clause. See ALTER TABLE command.
examples, 4 - 81
LOB_storage_clause. See ALTER TABLE command.
modify_default_attributes_clause. See ALTER TABLE command.
modify_LOB_storage_clause. See ALTER TABLE command.
modify_partition_clause. See ALTER TABLE command.
move_partition_clause. See ALTER TABLE command.
parallel_clause. See PARALLEL clause.
physical_attributes_clause. See ALTER TABLE

command.
rename_partition_clause. See ALTER TABLE command.
split_partition_clause. See ALTER TABLE command.
storage_clause. See STORAGE clause.
ALTER SNAPSHOT LOG command, 4 - 84
add_partition_clause. See ALTER TABLE command.
examples, 4 - 86
modify_default_attributes_clause. See ALTER TABLE command.
modify_partition_clause. See ALTER TABLE command.
move_partition_clause. See ALTER TABLE command.
physical_attributes_clause. See ALTER TABLE command.
rename_partition_clause. See ALTER TABLE command.
split_partition_clause. See ALTER TABLE command.
ALTER SYSTEM command, 4 - 88
archive_log_clause. See ARCHIVE LOG clause.
dispatch_clause, 4 - 92
examples, 4 - 98, 4 - 100, 4 - 102, 4 - 104
options_clause, 4 - 93
set_clause, 4 - 90
ALTER TABLE command, 4 - 106
add_column_options_clause, 4 - 109
add_overflow_clause, 4 - 113
add_partition_clause, 4 - 114
allocate_extent_clause, 4 - 112
column_constraint, table_constraint. See CONSTRAINT clause.
column_ref_clause, 4 - 109
deallocate_unused_clause. See DEALLOCATE UNUSED clause.
drop_clause. See DROP clause.
examples, 4 - 124
exchange_partition_clause, 4 - 115
index_organized_table_clauses, 4 - 112
LOB_index_clause, 4 - 111
LOB_parameters, 4 - 110
LOB_storage_clause, 4 - 110
modify_column_options_clause, 4 - 109
modify_LOB_index_clause, 4 - 112
modify_LOB_storage_clause, 4 - 111
modify_partition_clause, 4 - 114
move_partition_clause, 4 - 114, 4 - 115
nested_table_storage_clause, 4 - 112
overflow_clause, 4 - 113
parallel_clause. See PARALLEL clause.
partitioning_clauses, 4 - 113
physical_attributes_clause, 4 - 110
segment_partition_clause, 4 - 115
split_partition_clause, 4 - 115
storage_clause. See STORAGE clause.
table_ref_clause, 4 - 109
ALTER TABLESPACE command, 4 - 133
autoextend_clause, 4 - 136
examples, 4 - 139
filespec. See “Filespec”.
storage_clause. See STORAGE clause.
ALTER TRIGGER command, 4 - 141
examples, 4 - 142
ALTER TYPE command, 4 - 144
pragma_clause, 4 - 145
ALTER USER command, 4 - 150
examples, 4 - 152
ALTER VIEW command, 4 - 154
examples, 4 - 155
altering
costs of resources, 4 - 48
databases, 4 - 15
indexes, 4 - 31
profiles, 4 - 43
sequences, 4 - 56
snapshot logs, 4 - 84
snapshots, 4 - 76
tables, 4 - 106
AM / A.M. format element, 3-72
American National Standards Committee (ANSI), 1- 2
ANALYZE command, 4 - 156
examples, 4 - 163, 4 - 164
AND logical operator, 3-11
in a condition, 3-91
truth table, 3-12
ANSI

American National Standards Institute, 1- 2
datatypes, 2-20
X3.135-1992, 1- 2
ANY comparison operator, 3-5
application failover
See DISCONNECT SESSION clause of ALTER SYSTEM command.
ARCHIVE LOG clause, 4 - 167
examples, 4 - 169
of ALTER SYSTEM command, 4 - 96
ARCHIVELOG option
of ALTER DATABASE command, 4 - 19
of CREATE CONTROLFILE command, 4 - 217
of CREATE DATABASE command, 4 - 222
archiving redo log files
disabling, 4 - 19
enabling, 4 - 19
arithmetic
with DATE values, 2-14
operator, 3-3
AS clause
of CREATE SNAPSHOT command, 4 - 291
of CREATE TABLE command, 4 - 320
of CREATE VIEW command, 4 - 365
AS EXTERNAL clause
of CREATE FUNCTION command, 4 - 235
AS OBJECT option
of CREATE TYPE command, 4 - 349
AS TABLE option
of CREATE TYPE command, 4 - 349
AS VARRAY option
of CREATE TYPE command, 4 - 349
ASC option
of CREATE INDEX command, 4 - 240
of ORDER BY clause, 4 - 493
ASCII
and EBCDIC, 3-4
character function, 3-33
character set, 2-26
attribute reference
expression syntax, 3-88
attribute_name parameter
of ALTER TYPE command, 4 - 145
of CREATE TYPE command, 4 - 349
attributes, 2-23
AUDIT command, 4 - 170, 4 - 178
examples, 4 - 175, 4 - 181
AUDIT object auditing option, 4 - 174
AUDIT_TRAIL, 4 - 171
AUTHENTICATED BY clause
of CREATE DATABASE LINK command, 4 - 226
authenticated_clause
of CREATE DATABASE LINK, 4 - 226
AUTOEXTEND
datafile size in tablespace, 4 - 137, 4 - 329
AUTOEXTEND clause
of ALTER DATABASE command, 4 - 23
of CREATE DATABASE command, 4 - 223
autoextend_clause
of ALTER DATABASE, 4 - 18
of ALTER TABLESPACE, 4 - 136
of CREATE DATABASE, 4 - 220
of CREATE TABLESPACE, 4 - 329
AUTOMATIC option
of RECOVER clause, 4 - 470
AVG group function, 3-57

B

B format element, 3-65
BACKUP CONTROLFILE clause
of ALTER DATABASE command, 4 - 21
BC / B.C. format element, 3-72
BEFORE option
of CREATE TRIGGER command, 4 - 334
BEGIN BACKUP option
of ALTER TABLESPACE command, 4 - 138
BETWEEN comparison operator, 3-5
BFILE datatype, 2-17
BFILENAME function, 3-50
BFILEs
accessing, 4 - 231
BITMAP option
of CREATE INDEX command, 4 - 240
blank-padded comparison semantics, 2-25
BLOB datatype, 2-17
block size
effect on PCTINCREASE, 4 - 525
BODY option

of ALTER PACKAGE command, 4 - 38
of COMPILE clause
of ALTER TYPE command, 4 - 145
of DROP PACKAGE command, 4 - 394
BUFFER_POOL option
of STORAGE clause, 4 - 526
BY ACCESS option
of AUDIT command, 4 - 171, 4 - 179
BY clause
of AUDIT command, 4 - 171
of NOAUDIT command, 4 - 461
BY SESSION option
of AUDIT command, 4 - 171, 4 - 179

C

C format element, 3-65
CACHE option
of CREATE TABLE, 4 - 320
CACHE parameter
of CREATE SEQUENCE command, 4 - 282, 4 - 284
CACHE_INSTANCES parameter
of ALTER SYSTEM command, 4 - 95
CALLING STANDARD clause
of CREATE FUNCTION command, 4 - 235
of CREATE PROCEDURE command, 4 - 262
CANCEL option
of RECOVER clause, 4 - 471
capitalizing
date format elements, 3-74
Cartesian product, 4 - 507
CASCADE CONSTRAINT option
of REVOKE (Schema Object Privileges) command, 4 - 479
CASCADE option
of ANALYZE command, 4 - 159
of DISABLE clause, 4 - 381
of DROP clause, 4 - 384
case sensitivity
in pattern matching, 3-9
in SQL statements, 1- 5
CAST expression, 3-84
CAST operator
expression syntax, 3-83
CEIL number function, 3-19, 3-20
century, storing, 2-14
change display format
format model, 3-63
CHANGE parameter
of ARCHIVE LOG clause, 4 - 168
changing
goal for the cost-based optimization approach, 4 - 71
optimization approach, 4 - 71
passwords, 4 - 150
CHAR datatype, 2-8
comparing values of, 2-25
character
comparison rules, 2-25
datatypes, 2-7
literal, 2-2
operator, 3-4
SQL functions, 3-25
CHARACTER SET parameter
of CREATE DATABASE command, 4 - 222
character sets, ASCII and EBCDIC, 2-26
CHARTOROWID conversion function, 3-42
CHECK constraints, 3-40, 4 - 201
CHECKPOINT clause
of ALTER SYSTEM command, 4 - 94, 4 - 97
CHR character function, 3-25
CHUNK option
of LOB storage clause, 4 - 318
CHUNK parameter
of LOB storage clause
of ALTER TABLE command, 4 - 118
CLEAR LOGFILE clause
of ALTER DATABASE command, 4 - 20
CLOB datatype, 2-18
CLONE DATABASE option
of ALTER DATABASE command, 4 - 18
CLOSE DATABASE LINK option
of ALTER SESSION command, 4 - 62
CLOSE_OPEN_CACHED_CURSORS parameter
of ALTER SESSION command, 4 - 63
closing
database links, 4 - 72
CLUSTER clause
of CREATE SNAPSHOT command, 4 - 289

of CREATE TABLE command, 4 - 318
cluster keys, 4 - 210
distinct values of, 4 - 211
CLUSTER option
of ANALYZE command, 4 - 158
of CREATE INDEX command, 4 - 240
of TRUNCATE command, 4 - 539
clusters, 4 - 207
adding snapshots to, 4 - 289
adding tables to, 4 - 213, 4 - 318
altering, 4 - 11
cluster indexes, 4 - 245
creating, 4 - 207
definition, 4 - 210
dropping, 4 - 386
indexed clusters, 4 - 211
order of columns in, 4 - 211
physical storage of tables in, 4 - 210
removing tables from, 4 - 386
size, 4 - 212
specifying tablespaces for, 4 - 209
storage characteristics of, 4 - 11, 4 - 208, 4 - 209
truncating, 4 - 538
column constraints, 4 - 188
column_ref_clause
of ALTER TABLE, 4 - 109
of CREATE TABLE, 4 - 309
columns
adding comments to, 4 - 183
adding integrity constraints to, 4 - 116, 4 - 123
adding to tables, 4 - 106, 4 - 116, 4 - 122
changing datatypes of, 4 - 116, 4 - 123
changing default values of, 4 - 116, 4 - 123
defining, 4 - 312
maximum number in a table, 4 - 312
maximum number in indexes, 4 - 243
of cluster keys, 4 - 210
order in indexes, 4 - 243
qualifying names with tables and schemas, 4 - 493
redefining, 4 - 106, 4 - 116, 4 - 123
removing comments from, 4 - 183
removing integrity constraints from, 4 - 116, 4 - 123
renaming, 4 - 474
selecting from tables, 4 - 489
specifying datatypes for, 4 - 313
vs. pseudocolumns, 2-32
COMMENT clause
of COMMIT command, 4 - 185
COMMENT command, 4 - 183
examples, 4 - 183
COMMENT object auditing option, 4 - 174
comments
adding to objects, 4 - 183
examples, 2-39
removing from objects, 4 - 183
within SQL statements, 2-38
COMMIT command, 4 - 185
ending a transaction, 4 - 485, 4 - 521
examples, 4 - 186
summary, 4 - 8
COMMIT option
of ADVISE clause, 4 - 62
committing
transactions, 4 - 185
comparison operators, 3-5
comparison rules, 2-24
comparison semantics
blank-padded, 2-25
nonpadded, 2-25
COMPILE option
of ALTER FUNCTION command, 4 - 26, 4 - 38
of ALTER PROCEDURE command, 4 - 41
of ALTER TYPE command, 4 - 145
of ALTER VIEW command, 4 - 154
compiling
procedures, 4 - 41
stored functions, 4 - 26
type bodies, 4 - 145
type specifications, 4 - 145
COMPLETE option
of REFRESH clause, 4 - 80, 4 - 290
complex snapshots, 4 - 291
CONCAT character function, 3-4
condition
example, 3-90
multiple, 3-91
syntax, 3-90
CONNECT BY clause

examples, 4 - 497
of SELECT command, 4 - 492, 4 - 496, 4 - 497
CONNECT role, 4 - 273
CONNECT statement auditing short cut, 4 - 176
CONNECT TO clause
of CREATE DATABASE LINK command, 4 - 226
CONNECT_TIME parameter
of CREATE PROFILE command, 4 - 267
constant
literal, 2-2
CONSTRAINT clause
examples, 4 - 198, 4 - 200
foreign_key_clause, 4 - 189
index_physical_attributes_clause, 4 - 190
storage_clause. See STORAGE clause.
CONSTRAINT identifier
of CONSTRAINT clause, 4 - 190
of WITH CHECK OPTION clause, 4 - 366
CONSTRAINT option
of DISABLE clause, 4 - 381
of DROP clause, 4 - 384
constraint states, 4 - 205, 4 - 419
CONSTRAINT(S) DEFAULT option
of ALTER SESSION command, 4 - 63
CONSTRAINT(S) DEFERRED option
of ALTER SESSION command, 4 - 63
constraints
deferred, 4 - 204
ENABLE NOVALIDATE, 4 - 192, 4 - 419
ENABLE VALIDATE, 4 - 192, 4 - 418
enabling disabled constraints, 4 - 420
Integrity constraints, 4 - 188
on object type columns, 4 - 326
CONTINUE option
of RECOVER clause, 4 - 471
CONVERT conversion function, 3-42
CONVERT option
of ALTER DATABASE command, 4 - 18
converting
strings to dates, 3-77
converting values, 2-28
explicitly, 2-29, 2-30
implicitly, 2-28, 2-30
correlated subqueries, 4 - 492, 4 - 534
correlated update, 4 - 547
COSH number function, 3-20
cost
of executing SQL statements, 4 - 427
COUNT group function, 3-58
CPU_PER_CALL parameter
of CREATE PROFILE command, 4 - 267
create
object view, 4 - 363
CREATE CLUSTER command, 4 - 207
examples, 4 - 213
parallel_clause, 4 - 207
parallel_clause. See PARALLEL clause.
physical_attributes_clause, 4 - 208
storage clause. See STORAGE clause.
CREATE CONTROLFILE command, 4 - 215
examples, 4 - 218
filespec. See “FILESPEC”.
CREATE DATABASE command, 4 - 219
autoextend_clause, 4 - 220
CREATE DATABASE LINK command, 4 - 225
authenticated_clause, 4 - 226
CREATE DIRECTORY command, 4 - 230
CREATE FUNCTION command, 4 - 232
CREATE INDEX command, 4 - 237
global_index_clause, 4 - 238
global_partition_clause, 4 - 239
index_physical_attributes_clause, 4 - 239
local_index_clause, 4 - 239
parallel clause. See PARALLEL clause.
storage_clause. See STORAGE clause.
CREATE LIBRARY command, 4 - 248
filespec. See “Filespec”.
CREATE PACKAGE BODY command, 4 - 254
examples, 4 - 255
CREATE PACKAGE command, 4 - 250
examples, 4 - 252
CREATE PROCEDURE command, 4 - 259
CREATE PROFILE command, 4 - 265
examples, 4 - 269
CREATE ROLE command, 4 - 272
examples, 4 - 274
CREATE ROLLBACK SEGMENT command, 4 - 275
examples, 4 - 276

storage_clause. See STORAGE clause.
CREATE SCHEMA command, 4 - 278
examples, 4 - 279
CREATE SEQUENCE command, 4 - 281
examples, 4 - 285
CREATE SNAPSHOT command, 4 - 286
examples, 4 - 293
index_physical_attributes_clause. See ALTER TABLE command.
LOB_storage_clause. See CREATE TABLE command.
parallel_clause. See PARALLEL clause.
physical_attributes_clause. See ALTER TABLE command.
select_command. See SELECT command.
table_partition_clause. See CREATE TABLE command.
CREATE SNAPSHOT LOG command, 4 - 297
examples, 4 - 300
LOB_storage_clause. See CREATE TABLE command.
parallel_clause. See PARALLEL clause.
physical_attributes_clause. See CREATE TABLE command.
storage_clause. See STORAGE clause.
table_partition_clause. See CREATE TABLE command.
CREATE STANDBY CONTROLFILE option
of ALTER DATABASE command, 4 - 21
CREATE SYNONYM command, 4 - 302
examples, 4 - 304
CREATE TABLE command, 4 - 306
column_ref_clause, 4 - 309
disable_clause. See DISABLE clause.
enable_clause. See ENABLE clause.
examples, 4 - 321
index_organized_table_clause, 4 - 310
LOB_index_clause, 4 - 311
LOB_storage_clause, 4 - 310
nested_table_storage_clause, 4 - 312
parallel_clause. See PARALLEL clause.
physical_attributes_clause, 4 - 310
segment_attributes_clause, 4 - 309
storage_clause. See STORAGE clause.
subquery. See “Subqueries”.
table_partition_clause, 4 - 312
table_ref_clause, 4 - 309
CREATE TABLESPACE command, 4 - 328
autoextend_clause, 4 - 329
examples, 4 - 331
filespec, 4 - 328
storage_clause. See STORAGE clause.
CREATE TRIGGER command, 4 - 333
examples, 4 - 340
CREATE TYPE BODY command, 4 - 353
examples, 4 - 355
CREATE TYPE command, 4 - 345
examples, 4 - 351
pragma_clause, 4 - 347
CREATE USER command, 4 - 357
examples, 4 - 360
CREATE VIEW command, 4 - 363
examples, 4 - 370
subquery, 4 - 364
creating
clusters, 4 - 207
database links, 4 - 225
databases, 4 - 219
directories, 4 - 230
dispatcher processes (DISP), 4 - 100
external functions, 4 - 232
indexes, 4 - 237
object tables, 4 - 312
object views, 4 - 370
packages, 4 - 250, 4 - 254
partitioned tables, 4 - 319
procedures, 4 - 259
profiles, 4 - 265
REF columns in tables, 4 - 325
roles, 4 - 272
rollback segments, 4 - 275
savepoints, 4 - 487
schemas, 4 - 278
sequences, 4 - 281
shared server processes, 4 - 99
snapshot logs, 4 - 297
snapshots, 4 - 286
stored functions, 4 - 232
synonyms, 4 - 302
tables, 4 - 306

tablespaces, 4 - 328
triggers, 4 - 333
users, 4 - 357
views, 4 - 363
CURRENT option
of ARCHIVE LOG clause, 4 - 168
CURRENT_USER option of CONNECT TO clause
of CREATE DATABASE LINK command, 4 - 226
CURRVAL pseudocolumn, 2-33
examples, 2-35, 4 - 536
cursors
storing in session cache, 4 - 71
CYCLE option
of CREATE SEQUENCE command, 4 - 282

D

D format element, 3-65
DANGLING REFs, 2-23
data complexity
hiding with a view, 4 - 366
data definition language (DDL), 4 - 2
data independence
via synonyms, 4 - 304
data manipulation language (DML), 4 - 7
data_item parameter of RETURNING clause
of DELETE command, 4 - 376, 4 - 545
of INSERT command, 4 - 453
database
deleting rollback segments from, 4 - 400
database links
closing, 4 - 72, 4 - 73
creating, 4 - 225
definition, 4 - 225
dropping, 4 - 388
using in DELETE command, 4 - 376
using in INSERT command, 4 - 453
using in LOCK TABLE command, 4 - 458
using in SELECT command, 4 - 492, 4 - 536
using in UPDATE command, 4 - 544
using with synonyms, 4 - 228
database objects
definition of, 2-44
DATABASE option
of RECOVER clause, 4 - 471
DATABASE parameter
of CREATE CONTROLFILE command, 4 - 216
databases
adding datafiles to, 4 - 219, 4 - 223
adding redo log files to, 4 - 219, 4 - 221
altering, 4 - 15
archiving redo log files, 4 - 19, 4 - 222
creating, 4 - 219
maximum number
of datafiles, 4 - 219
of instances, 4 - 222
of redo log files, 4 - 219, 4 - 221
mounting and dismounting, 4 - 18
opening and closing, 4 - 18
DATAFILE clause
of ALTER DATABASE command, 4 - 22
of CREATE CONTROLFILE clause, 4 - 217
of CREATE DATABASE command, 4 - 223
of CREATE TABLESPACE command, 4 - 329
DATAFILE option
of RECOVER clause, 4 - 471
DATAFILE parameter
of ALLOCATE EXTENT clause, 4 - 12, 4 - 33, 4 - 119
datafiles
adding, 4 - 136
adding to databases, 4 - 219, 4 - 223
adding to tablespaces, 4 - 133, 4 - 329
backing up, 4 - 133, 4 - 138
maximum number
for databases, 4 - 219
renaming, 4 - 21, 4 - 133
specifying, 4 - 431
datatypes, 2-5
ANSI, 2-20
changing for columns, 4 - 123
converting between values of different, 2-28
converting between with SQL functions, 3-42
DB2, 2-20
explicit conversion, 2-29
implicit conversion, 2-28
of conditions, 3-90
of expressions, 3-78
specifying for columns, 4 - 208, 4 - 313

SQL/DS, 2-20
summary, 2-5
date
arithmetic, 2-14
comparison rules, 2-24
DATE datatype, 2-13
comparing values of, 2-24
julian, 2-14
DATE format element, 3-65
date format elements, 3-69
capitalizing, 3-74
date format model, 3-69
default, 3-69
examples, 3-64, 3-76
modifiers, 3-75
suffixes, 3-74
DAY format element, 3-72
DB2
datatypes, 2-20
DBA role, 4 - 273
DBA statement auditing short cut, 4 - 177
DBA_CLUSTERS view, 4 - 163
DBA_INDEXES view, 4 - 161
DBA_TAB_COLUMNS view, 4 - 163
DBA_TABLES view, 4 - 161
DBMS_SNAPSHOT.REFRESH() procedure, 4 - 292
DBMSSTDX.SQL, 4 - 232, 4 - 250, 4 - 254, 4 - 259, 4 - 333
DDL (data definition language), 4 - 2
DEALLOCATE UNUSED clause, 4 - 372
of ALTER TABLE command, 4 - 119
deallocating space, 4 - 119
decimal places
negative, 2-11
DECODE expression, 3-89
default
cluster key, 4 - 246
DEFAULT option
of ALTER PROFILE command, 4 - 48
of BUFFER_POOL option
of STORAGE clause, 4 - 526
of CREATE PROFILE command, 4 - 268
of CREATE TABLE command, 4 - 313
of NOAUDIT command, 4 - 464
of ROLLBACK SEGMENT clause, 4 - 289
of ALTER SNAPSHOT command, 4 - 80
default privilege domain, 4 - 517
DEFAULT profile, 4 - 269, 4 - 359, 4 - 398
DEFAULT ROLE clause
of ALTER USER command, 4 - 152
DEFERRABLE constraints, 4 - 204
DEFERRABLE option
of CONSTRAINT clause, 4 - 191
deferring constraints, 4 - 204
DELETE command, 4 - 374
returning_clause, 4 - 375
subquery, 4 - 375
summary, 4 - 7
DELETE object auditing option, 4 - 174
DELETE object privilege
on tables, 4 - 447
on views, 4 - 447
DELETE option
of CREATE TRIGGER command, 4 - 335, 4 - 338
DELETE_CATALOG_ROLE role, 4 - 274
deleting
from a single partition, 4 - 378
referenced key values, 4 - 197rows from tables, 4 - 405
rows from tables and views, 4 - 374
delimited names
quoted names, 2-49
DEREF function, 3-56
DESC option
of CREATE INDEX command, 4 - 240
of ORDER BY clause, 4 - 493
directories
creating, 4 - 230
definition, 3-50, 4 - 230
directory object
defined, 4 - 231
DIRECTORY option
of GRANT (Object Privileges), 4 - 445
directory parameter
of AUDIT (Schema Objects) command, 4 - 179
of NOAUDIT (Schema Objects) command, 4 - 464
DISABLE clause, 4 - 380
examples, 4 - 382
of ALTER DATABASE command, 4 - 22

of CREATE TABLE command, 4 - 319
DISABLE COMMIT IN PROCEDURE option
of ALTER SESSION command, 4 - 62
DISABLE option
of ALTER TRIGGER command, 4 - 141
of CONSTRAINT clause, 4 - 192
of PARALLEL DML clause
of ALTER SESSION command, 4 - 63
DISABLE STORAGE IN ROW option
of LOB storage clause
of ALTER TABLE command, 4 - 117
of CREATE TABLE command, 4 - 317
disabled constraints
enabling, 4 - 420
disabling
distributed recovery, 4 - 103
integrity constraints, 4 - 319, 4 - 380
redo log threads, 4 - 22
resource limits, 4 - 88, 4 - 98
roles for sessions, 4 - 516
SQL trace facility for sessions, 4 - 66, 4 - 67
triggers, 4 - 142
DISCONNECT SESSION clause
of ALTER SYSTEM command, 4 - 96
disconnecting
sessions, 4 - 96, 4 - 104
dispatch_clause
of ALTER SYSTEM, 4 - 92
dispatcher processes (DISP)
creating and terminating, 4 - 100
DISTINCT clause
with ORDER BY clause, 4 - 502
DISTINCT option
of SELECT command, 4 - 491
of SQL group functions, 3-57
distributed query, 4 - 536
examples, 4 - 537
restrictions on, 4 - 536
distributed recovery
disabling, 4 - 88, 4 - 103
enabling in a single-process environment, 4 - 88
enabling in single-process environments, 4 - 103
distributed transactions, 4 - 186, 4 - 486
DML (data manipulation language), 4 - 7
DROP clause, 4 - 384
examples, 4 - 385
of ALTER TABLE command, 4 - 118
DROP CLUSTER command, 4 - 386
examples, 4 - 387
DROP DATABASE LINK command, 4 - 388
examples, 4 - 388
DROP DIRECTORY command, 4 - 389
examples, 4 - 389
DROP FUNCTION command, 4 - 390
examples, 4 - 390
DROP INDEX command, 4 - 392
examples, 4 - 392
DROP LIBRARY command, 4 - 393
DROP LOGFILE clause
of ALTER DATABASE command, 4 - 20
DROP LOGFILE MEMBER clause
of ALTER DATABASE command, 4 - 20
DROP PACKAGE command, 4 - 394
DROP PARTITION option
of ALTER TABLE command, 4 - 121
DROP PROCEDURE command, 4 - 396
examples, 4 - 395, 4 - 396
DROP PROFILE command, 4 - 398
examples, 4 - 398
DROP ROLE command, 4 - 399
examples, 4 - 399
DROP ROLLBACK SEGMENT command, 4 - 400
examples, 4 - 400
DROP SEQUENCE command, 4 - 401
examples, 4 - 401
DROP SNAPSHOT command, 4 - 402
examples, 4 - 402
DROP SNAPSHOT LOG command, 4 - 403
examples, 4 - 403
DROP STORAGE option
of TRUNCATE command, 4 - 539
DROP SYNONYM command, 4 - 404
examples, 4 - 404
DROP TABLE command, 4 - 405
examples, 4 - 406
DROP TABLESPACE command, 4 - 407
examples, 4 - 408
DROP TRIGGER command, 4 - 409
examples, 4 - 409
DROP TYPE BODY command, 4 - 412

DROP TYPE command, 4 - 410
DROP USER command, 4 - 414
  examples, 4 - 415
DROP VIEW command
  examples, 4 - 416
dropping
  clusters, 4 - 386
  database links, 4 - 388
  indexes, 4 - 392, 4 - 393
  integrity constraints from tables, 4 - 118, 4 - 384
  members from redo log file groups, 4 - 20
  objects owned by users, 4 - 414
  package bodies, 4 - 394
  packages, 4 - 394
  procedures, 4 - 248, 4 - 396
  profiles, 4 - 398
  redo log file groups, 4 - 20
  referential integrity constraints, 4 - 479
  roles, 4 - 399
  sequences, 4 - 401
  snapshot logs, 4 - 403
  snapshots, 4 - 402
  stored functions, 4 - 390
  synonyms, 4 - 404
  tables, 4 - 405
  tablespaces, 4 - 407
  triggers from tables, 4 - 409
  users, 4 - 414
  views, 4 - 416
DUAL data dictionary table
  definition, 4 - 535
  example of selecting from, 4 - 535
DUMMY column
  of DUAL table, 4 - 535
DUMP function, 3-49
DY format element, 3-72
dynamic performance tables
  V$LOG, 4 - 221
  V$NLS_PARAMETERS, 4 - 67
dynamic performance views
  V$LOG, 4 - 19

E

EBCDIC
  and ASCII, 3-4
  character set, 2-26
EEEE format element, 3-65
embedded SQL, 1- 3
EMPTY_BLOB function, 3-50
EMPTY_CLOB function, 3-50
ENABLE clause, 4 - 417
  examples, 4 - 422
  exceptions_clause, 4 - 418
  of ALTER DATABASE command, 4 - 22
  of CREATE TABLE command, 4 - 319
  storage_clause. See STORAGE clause.
  using_index_clause, 4 - 418
ENABLE DISTRIBUTED RECOVERY option
  of ALTER SYSTEM command, 4 - 103
ENABLE NOVALIDATE, 4 - 420
ENABLE NOVALIDATE constraints, 4 - 192, 4 - 419
ENABLE NOVALIDATE option
  of CONSTRAINT clause, 4 - 192
ENABLE option
  of ALTER TRIGGER command, 4 - 141
  of PARALLEL DML clause
    of ALTER SESSION command, 4 - 63
ENABLE STORAGE IN ROW option
  of LOB storage clause
    of ALTER TABLE command, 4 - 117
    of CREATE TABLE command, 4 - 317
ENABLE VALIDATE, 4 - 420
ENABLE VALIDATE constraints, 4 - 192, 4 - 418
ENABLE VALIDATE option
  of CONSTRAINT clause, 4 - 192
enabling
  distributed recovery, 4 - 88, 4 - 103
  integrity constraints, 4 - 319, 4 - 417
  primary and unique key constraints, 4 - 205, 4 - 420
  redo log threads, 4 - 22
  resource limits, 4 - 88, 4 - 98, 4 - 269
  roles for sessions, 4 - 516
  SQL trace facility for sessions, 4 - 66, 4 - 67
  triggers, 4 - 142
enabling and diabling constraints, 4 - 205, 4 - 419
END BACKUP
  of ALTER DATABASE command, 4 - 23

END BACKUP option
of ALTER TABLESPACE command, 4 - 138
ending
transactions, 4 - 185
ENTRYID option
of USERENV function, 3-55
equijoins, 4 - 505
ESCAPE character
of LIKE operator, 3-8
examples
of comments, 2-39
EXCEPT clause
of SET ROLE command, 4 - 516
EXCEPTIONS INTO clause
of CONSTRAINT clause, 4 - 191
exceptions_clause
of ENABLE, 4 - 418
EXCHANGE PARTITION option
of ALTER TABLE command, 4 - 121
exchange_partition_clause
of ALTER TABLE, 4 - 115
EXECUTE object auditing option, 4 - 174
EXECUTE object privilege
on procedures, functions, and packages, 4 - 448
EXECUTE_CATALOG_ROLE role, 4 - 274
EXISTS comparison operator, 3-5
EXP number function, 3-20
EXPLAIN PLAN command, 4 - 425
analyzing partitioned tables, 4 - 427
examples, 4 - 427
summary, 4 - 7
expression, 3-78
examples, 3-78
use in condition, 3-90
expression syntax
attribute reference, 3-88
method invocation, 3-88
VALUE operator, 3-87
extents
allocating for tables, 4 - 106
deallocating space, 4 - 372
INITIAL size, 4 - 524
MAXEXTENTS limit, 4 - 525
EXTERNAL clause
of CREATE PROCEDURE command, 4 - 262
external functions
creating, 4 - 232
defined, 4 - 232
external procedures
definition, 4 - 259
EXTERNALLY option
of IDENTIFIED clause, 4 - 272, 4 - 359, 4 - 360

F

FAILED_LOGIN_ATTEMPTS option
of CREATE PROFILE command, 4 - 267
FALSE
result of a condition, 3-90
FAST option
of REFRESH clause, 4 - 80, 4 - 290
Federal Information Processing Standard (FIPS), 1- 2
filespec, 4 - 431
examples, 4 - 432
of CREATE TABLESPACE, 4 - 328
fill mode
trims trailing blanks, 3-75
FIPS, 4 - 63
Federal Information Processing Standard, 1- 2
flagging, 4 - 71
PUB 127-2, 1- 2
FIPS Flagger, B - 13
FLAGGER clause
of ALTER SESSION command, 4 - 63
flattened subqueries
definition, 4 - 532
in DELETE statements, 4 - 376
using, 4 - 533
using in UPDATE command, 4 - 544
FLOAT
ANSI datatype, 2-12
FLOOR number function, 3-20
FM date format element prefix
examples, 3-76
FM format model modifier, 3-75
FOR clause
of EXPLAIN PLAN command, 4 - 426
FOR EACH ROW option
of CREATE TRIGGER command, 4 - 336

FOR RECOVER option
of ALTER TABLESPACE command, 4 - 138
FOR UPDATE clause
of SELECT command, 4 - 493, 4 - 503
FOR UPDATE OF
example, 4 - 504
FOR UPDATE option
of CREATE SNAPSHOT command, 4 - 290
FORCE clause
of COMMIT command, 4 - 186
of ROLLBACK command, 4 - 484
FORCE option
of CREATE VIEW command, 4 - 364
of PARALLEL DML clause
of ALTER SESSION command, 4 - 63
of REFRESH clause, 4 - 80, 4 - 290
of REVOKE (Schema Object Privileges) command, 4 - 479
foreign key constraints, 4 - 196
FOREIGN KEY option
of CONSTRAINT clause, 4 - 197
foreign_key_clause
of CONSTRAINT clause, 4 - 189
format model
date, 3-69
definition, 3-63
examples, 3-63, 3-64, 3-76
number, 3-65
formatting
date values, 3-69
number values, 3-65
FROM clause
of REVOKE command, 4 - 475, 4 - 479
FROM parameter
of RECOVER clause, 4 - 470
function_specification parameter
of ALTER TYPE command, 4 - 147
of CREATE TYPE command, 4 - 350
functions
PL/SQL, 3-60
SQL, 3-16
stored functions, 4 - 232
user, 3-60
FX format model modifier, 3-75

G

G format element, 3-65
GLOBAL option
of CHECK DATAFILES clause, 4 - 94
of CHECKPOINT clause, 4 - 94
global_index_clause
of CREATE INDEX, 4 - 238
GLOBAL_NAMES, 4 - 99
global_partition_clause
of CREATE INDEX, 4 - 239
GLOBALLY AS option
of IDENTIFIED clause, 4 - 359
GLOBALLY option
of IDENTIFIED clause, 4 - 273
GRANT command, 4 - 434, 4 - 444
examples, 4 - 442, 4 - 449
part of CREATE SCHEMA command, 4 - 278
GRANT object auditing option, 4 - 174
GRANT OPTION
of GRANT command, 4 - 446
granting
object privileges to users and roles, 4 - 444
roles, 4 - 434
system privileges and roles to users, roles, 4 - 434
GRAPHIC datatype, 2-22
GREATEST function, 3-51
group
SQL functions, 3-57
GROUP BY clause
group SQL functions and, 3-17
of SELECT command, 4 - 492, 4 - 499
GROUP parameter
of ADD LOGFILE MEMBER clause, 4 - 20
of DROP LOGFILE clause, 4 - 20

H

HASH parameter
of CREATE CLUSTER command, 4 - 209
HAVING clause
of SELECT command, 4 - 493, 4 - 500
HEXTORAW conversion function, 3-43
hints, 2-40

in DELETE statements, 4 - 377
in SELECT statements, 4 - 494
in UPDATE statements, 4 - 542

I

I date format element, 3-72
IDENTIFIED clause
of CREATE ROLE command, 4 - 272
of CREATE USER command, 4 - 358
identifiers
names, 2-47
ORACLE, how to form, A-3
IDLE_TIME parameter
of CREATE PROFILE command, 4 - 267
IEC (International Electrotechnical Commission), 1- 2
IMMEDIATE option
of ALTER TABLESPACE command, 4 - 138
implies, 4 - 191
IN comparison operator, 3-5
definition, 3-5
INCLUDING clause
of ALTER TABLE command, 4 - 117
of CREATE TABLE command, 4 - 317
INCREMENT BY clause
of CREATE SEQUENCE command, 4 - 282
incrementing
sequence values, 2-33, 4 - 536
INDEX object auditing option, 4 - 174
INDEX object privilege
on tables, 4 - 447
INDEX option
of ANALYZE command, 4 - 158
of CREATE CLUSTER command, 4 - 209
INDEX parameter
of LOB storage clause, 4 - 318
index_organized_table_clause
of ALTER TABLE, 4 - 112
of CREATE TABLE, 4 - 310
index_physical_attributes_clause
of ALTER INDEX, 4 - 30
of CONSTRAINT clause, 4 - 190
of CREATE INDEX, 4 - 239
INDEX_STATS view, 4 - 164
indexed clusters, 4 - 211
indexes
altering, 4 - 28
and LIKE operator, 3-9
bitmap indexes, 4 - 246
cluster indexes, 4 - 245
creating, 4 - 237
definition, 4 - 237
dropping, 4 - 392, 4 - 393, 4 - 405
LONG RAW datatypes prohibit, 2-15
multiple per table, 4 - 244
nested table columns, 4 - 247
partitioned indexes, 4 - 246
storage characteristics of, 4 - 28, 4 - 241
indexing
specifying tablespaces for, 4 - 241
index-organized tables, 4 - 323
creating, 4 - 316
examples, 4 - 323
INITCAP character function, 3-26
INITIAL parameter
of STORAGE clause, 4 - 524, 4 - 527
initialization parameters
AUDIT_TRAIL, 4 - 171
GLOBAL_NAMES, 4 - 99
LOG_FILES, 4 - 221
MAX_ENABLED_ROLES, 4 - 517
MTS_MAX_DISPATCHERS, 4 - 100
MTS_MAX_SERVERS, 4 - 99
MTS_SERVERS, 4 - 99
NLS_DATE_FORMAT, 3-69
NLS_DATE_LANGUAGE, 3-72
NLS_LANGUAGE, 3-72
NLS_TERRITORY, 3-68, 3-69, 3-72
OPEN_LINKS, 4 - 72, 4 - 227
OPTIMIZER_MODE, 4 - 71
OS_AUTHENT_PREFIX, 4 - 361
OS_ROLES, 4 - 442
ROLLBACK_SEGMENTS, 4 - 54
SNAPSHOT_REFRESH_INTERVAL, 4 - 293
SNAPSHOT_REFRESH_KEEP_CONNECTIONS, 4 - 293
SNAPSHOT_REFRESH_PROCESSES, 4 - 293
SQL_TRACE, 4 - 67
THREAD, 4 - 22

INITIALLY DEFERRABLE option
of CONSTRAINT clause, 4 - 191
INITIALLY IMMEDIATE option
of CONSTRAINT clause, 4 - 191
INITRANS parameter
of ALTER CLUSTER command, 4 - 12
of ALTER INDEX command, 4 - 32
of ALTER SNAPSHOT command, 4 - 79
of ALTER TABLE command, 4 - 85, 4 - 117
of CREATE CLUSTER command, 4 - 209
of CREATE INDEX command, 4 - 241
of CREATE SNAPSHOT command, 4 - 289, 4 - 299
of CREATE TABLE command, 4 - 315
inline LOB storage, 4 - 117, 4 - 317
INSERT command, 2-12, 4 - 451
examples, 2-35, 4 - 455
summary, 4 - 7
INSERT object auditing option, 4 - 174
INSERT object privilege
on tables, 4 - 447
on views, 4 - 447
INSERT option
of CREATE TRIGGER command, 4 - 335
inserting
rows into tables and views, 4 - 451
into views, 4 - 454
INSTANCE clause
of ALTER SESSION command, 4 - 64
INSTANCE parameter
of ALLOCATE EXTENT clause, 4 - 13, 4 - 33, 4 - 119
instances
maximum number
for databases, 4 - 222
INSTEAD OF option
of CREATE TRIGGER command, 4 - 335
instead of triggers
using, 4 - 342
INSTR character function, 3-34
INTEGER datatype, 2-3
integrity constraints
adding to columns, 4 - 116, 4 - 123
adding to tables, 4 - 116, 4 - 123
CHECK, 4 - 191, 4 - 201
column definition, 4 - 188
creating as parts of tables, 4 - 313
defining, 4 - 188
definition, 4 - 188
disabling, 4 - 319, 4 - 380
dropping from tables, 4 - 118
enabling, 4 - 319, 4 - 417
NOT NULL, 4 - 193
PRIMARY KEY, 4 - 190, 4 - 195
referential, 4 - 190, 4 - 196
removing from columns, 4 - 116, 4 - 123
table definition, 4 - 116, 4 - 188, 4 - 192, 4 - 313
UNIQUE, 4 - 190, 4 - 193
International Electrotechnical Commission (IEC), 1- 2
International Standards Organization (ISO), 1- 2
INTERSECT set operator, 4 - 493
examples, 3-15
INTO clause
of ANALYZE command, 4 - 159
of EXPLAIN PLAN command, 4 - 426
IS DANGLING, 2-23
IS NOT DANGLING, 2-23
IS NOT NULL comparison operator, 3-5
IS NULL comparison operator, 3-5
ISDBA option
of USERENV function, 3-55
ISO
International Standards Organization, 1- 2
ISO / IEC 9075
1992, 1- 2
ISOLATION LEVEL clause
of SET TRANSACTION command, 4 - 520
of ALTER SESSION command, 4 - 64
IW date format element, 3-72
IY date format element, 3-72
IYY date format element, 3-72
IYYY date format element, 3-72

J

job queue processes, 4 - 293
join view, 4 - 368
joins
and clusters, 4 - 210

examples, 4 - 506, 4 - 509
simple, 4 - 505
julian dates, 2-14

K

KEEP option
of BUFFER_POOL option
of STORAGE clause, 4 - 526
KILL SESSION clause
of ALTER SYSTEM command, 4 - 96

L

L format element, 3-65
LANGUAGE clause
of CREATE FUNCTION command, 4 - 235
of CREATE PROCEDURE command, 4 - 262
LANGUAGE option
of USERENV function, 3-55
LAST_DAY date function, 3-37
LEAST function, 3-51
LENGTH character function, 3-35
LENGTHB character function, 3-35
LEVEL pseudocolumn, 2-35
lexical conventions
SQL, 1- 4
LIBRARY clause
of CREATE FUNCTION command, 4 - 235
of CREATE PROCEDURE command, 4 - 262
LIKE comparison operator, 3-5
definition, 3-8
links
database links, 4 - 225
LIST CHAINED ROWS clause
of ANALYZE command, 4 - 159
literal
character, 2-2
definition of, 2-2
numeric, 2-2
LN number function, 3-21
LOB columns
adding to tables, 4 - 125
LOB datatypes, 2-15
LOB functions, 3-50
LOB storage
in row, 4 - 117, 4 - 317
out of line, 4 - 117, 4 - 317
LOB storage clause
of ALTER SNAPSHOT command, 4 - 79
of CREATE TABLE command, 4 - 317
LOB storage parameters
specifying in ALTER TABLE command, 4 - 118
LOB_index_clause
of ALTER TABLE, 4 - 111
of CREATE TABLE, 4 - 311
LOB_parameters
of ALTER TABLE, 4 - 110
LOB_storage_clause
of ALTER TABLE, 4 - 110
of CREATE TABLE, 4 - 310
LOCAL option
of CHECK DATAFILES clause, 4 - 94
of CHECKPOINT clause, 4 - 94
of ROLLBACK SEGMENT clause, 4 - 289
of ALTER SNAPSHOT command, 4 - 81
LOCAL parameter
of ANALYZE command, 4 - 159
local_index_clause
of CREATE INDEX, 4 - 239
location transparency
through synonyms, 4 - 304
lock
and queries, 4 - 459
exclusive, 4 - 459
multiple, 4 - 459
released by ROLLBACK statement, 4 - 485
releasing with COMMIT command, 4 - 185
table, 4 - 459
types of, 4 - 459
LOCK object auditing option, 4 - 174
LOCK TABLE command, 4 - 458
examples, 4 - 459
summary, 4 - 7
locking
tables and views, 4 - 458
LOG number function, 3-21
LOG_FILES, 4 - 221
logarithms
LN number function, 3-21

LOG number function, 3-21
LOGFILE clause
of CREATE CONTROLFILE command, 4 - 216
of CREATE DATABASE command, 4 - 221
LOGFILE parameter
of ARCHIVE LOG clause, 4 - 168
of RECOVER clause, 4 - 471
LOGGING option
of ALTER INDEX command, 4 - 33
of ALTER TABLE command, 4 - 120
of ALTER TABLESPACE command, 4 - 136
of CREATE INDEX command, 4 - 241
of CREATE TABLE command, 4 - 316
of CREATE TABLESPACE command, 4 - 329
logical operator
definition, 3-11
use in condition, 3-90
LONG datatype, 2-12
maximum length, 2-12
restrictions on, 2-12
LONG RAW datatype, 2-15
indexing prohibited on, 2-15
similarity to LONG datatype, 2-15
LONG VARGRAPHIC datatype, 2-22
LOWER character function, 3-27
lowercase
significance in SQL statements, 1- 5
lowercase and uppercase
of schema object names, 2-47
significance in pattern matching, 3-9
LPAD character function, 3-27
LTRIM character function, 3-27

M

MAKE_REF function, 3-56
MAP MEMBER clause
of ALTER TYPE command, 4 - 146
of CREATE TYPE command, 4 - 350
MAP method
object value comparisons, 2-28
MASTER option
of ROLLBACK SEGMENT clause, 4 - 289
of ALTER SNAPSHOT command, 4 - 81
MAX group function, 3-58
MAX_DUMP_FILE_SIZE option
of ALTER SESSION command, 4 - 64
of ALTER SYSTEM command, 4 - 96
MAX_ENABLE_ROLES, 4 - 517
MAXDATAFILES parameter
of CREATE CONTROLFILE command, 4 - 217
of CREATE DATABASE command, 4 - 222
MAXEXTENTS parameter
of STORAGE clause, 4 - 525, 4 - 527
MAXINSTANCES parameter
of CREATE CONTROLFILE command, 4 - 217
of CREATE DATABASE command, 4 - 222
MAXLOGFILES parameter
of CREATE CONTROLFILE command, 4 - 217
of CREATE DATABASE command, 4 - 221
MAXLOGHISTORY parameter
of CREATE DATABASE command, 4 - 222
MAXLOGMEMBERS parameter
of CREATE DATABASE command, 4 - 222
MAXSIZE clause
of ALTER DATABASE command, 4 - 23
MAXTRANS parameter
of ALTER CLUSTER command, 4 - 12
of ALTER INDEX command, 4 - 32
of ALTER SNAPSHOT command, 4 - 79
of ALTER TABLE command, 4 - 85, 4 - 117
of CREATE CLUSTER command, 4 - 209
of CREATE INDEX command, 4 - 241
of CREATE SNAPSHOT command, 4 - 289, 4 - 299
of CREATE TABLE command, 4 - 315
MAXVALUE parameter
of CREATE SEQUENCE command, 4 - 282
of partitioning clause, 4 - 319
MEMBER clause
of CREATE TYPE command, 4 - 146
MEMBER clause of CREATE TYPE command, 4 - 350
method invocation
expression syntax, 3-88
method_name parameter
of CREATE TYPE command, 4 - 147
method_name parameter of CREATE TYPE command, 4 - 351
methods, 2-23

MI format element, 3-65
MIN group function, 3-59
MINEXTENTS parameter
of STORAGE clause, 4 - 525
MINIMUM EXTENT parameter
of ALTER TABLESPACE command, 4 - 137
of CREATE TABLESPACE command, 4 - 329
MINUS set operator, 4 - 493
examples, 3-15
MINVALUE parameter
of CREATE SEQUENCE command, 4 - 282
miscellaneous operators, 3-16
MLSLABEL datatype, 2-20
MODIFY clause
of ALTER TABLE command, 4 - 116
MODIFY DEFAULT ATTRIBUTES
of ALTER INDEX command, 4 - 34, 4 - 86
MODIFY DEFAULT ATTRIBUTES option
of ALTER SNAPSHOT command, 4 - 79
of ALTER TABLE command, 4 - 116
modify LOB storage clause
of ALTER SNAPSHOT command, 4 - 79
MODIFY PARTITION option
of ALTER SNAPSHOT command, 4 - 79, 4 - 85
of ALTER TABLE command, 4 - 120
modify_column_options_clause
of ALTER TABLE, 4 - 109
modify_LOB_index_clause
of ALTER TABLE, 4 - 112
modify_LOB_storage_clause
of ALTER TABLE, 4 - 111
modify_partition_clause
of ALTER TABLE, 4 - 114
modifying
column definitions, 4 - 106, 4 - 116, 4 - 123
resource limits, 4 - 43
MON format element, 3-72
MONTH format element, 3-72
MONTHS_BETWEEN date function, 3-38
MOUNT option
of ALTER DATABASE command, 4 - 18
mounting
databases, 4 - 18
MOVE PARTITION option
of ALTER SNAPSHOT command, 4 - 79, 4 - 85
of ALTER TABLE command, 4 - 121
move_partition_clause
of ALTER TABLE, 4 - 114, 4 - 115
MTS_DISPATCHERS parameter
of ALTER SYSTEM command, 4 - 95, 4 - 99
MTS_MAX_DISPATCHERS, 4 - 100
MTS_MAX_SERVERS, 4 - 99
MTS_SERVERS parameter
of ALTER SYSTEM command, 4 - 95, 4 - 99
multithreaded server
managing processes for, 4 - 95, 4 - 99

N

NAME clause
of CREATE FUNCTION command, 4 - 235
of CREATE PROCEDURE command, 4 - 262
names
for schema objects, 2-47
lowercase and uppercase, 2-47
quoted, 2-49
namespaces
for schema objects, 2-48
naming
of database objects, A-3
of schema objects, 2-47
NATIONAL CHARACTER SET parameter
of CREATE DATABASE command, 4 - 223
National Institute for Standards and Technology (NIST), 1- 2
National Language Support (NLS)
session settings, 4 - 64, 4 - 67
natural logarithms, 3-21
navigation
automatic, 1- 2
NCHAR datatype, 2-8
NCLOB datatype, 2-18
negative scale, 2-11
nested CURSOR
expression syntax, 3-85
nested table columns
adding to tables, 4 - 126
identifying in a subquery, 4 - 532
indexes, 4 - 247
nested table storage

creating, 4 - 325
NESTED TABLE storage clause
of ALTER TABLE command, 4 - 118
of CREATE TABLE command, 4 - 318
nested table types, 2-24
nested_table_storage_clause
of ALTER TABLE, 4 - 112
of CREATE TABLE, 4 - 312
NEW_TIME date function, 3-38
NEXT clause
of ALTER DATABASE command, 4 - 23
NEXT option
of ARCHIVE LOG clause, 4 - 168
NEXT parameter
of REFRESH clause, 4 - 80, 4 - 290
of STORAGE clause, 4 - 525
NEXT_DAY date function, 3-39
NEXTVAL pseudocolumn, 2-33
examples, 2-35, 4 - 456, 4 - 536
NIST
National Institute for Standards and Technology, 1- 2
NLS_CHARSET_DECL_LEN function, 3-52
NLS_CHARSET_ID function, 3-52
NLS_CHARSET_NAME function, 3-53
NLS_CURRENCY parameter
of ALTER SESSION command, 4 - 65
NLS_DATE_FORMAT parameter
of ALTER SESSION command, 4 - 65
NLS_DATE_LANGUAGE parameter
of ALTER SESSION command, 4 - 65
NLS_INITCAP character function, 3-28
NLS_ISO_CURRENCY parameter
of ALTER SESSION command, 4 - 65
NLS_LANGUAGE parameter
of ALTER SESSION command, 4 - 64
NLS_LOWER character function, 3-23, 3-26, 3-28
NLS_NUMERIC_CHARACTERS parameter
of ALTER SESSION command, 4 - 65
NLS_SORT parameter
of ALTER SESSION command, 4 - 65
NLS_TERRITORY parameter
of ALTER SESSION command, 4 - 65
NLS_UPPER character function, 3-29
NLSSORT character function, 3-36
NOARCHIVELOG option
of ALTER DATABASE command, 4 - 19
of CREATE DATABASE command, 4 - 222
NOAUDIT command, 4 - 461, 4 - 463
examples, 4 - 462, 4 - 464
NOCACHE option
of ALTER TABLE command, 4 - 119
of CREATE SEQUENCE command, 4 - 283
of CREATE TABLE, 4 - 320
NOCYCLE option
of CREATE SEQUENCE command, 4 - 282
NOFORCE option
of CREATE VIEW command, 4 - 364
NOLOGGING option
of ALTER INDEX command, 4 - 33
of ALTER TABLE command, 4 - 120
of ALTER TABLESPACE command, 4 - 136
of CREATE INDEX command, 4 - 245
of CREATE TABLE command, 4 - 316
of CREATE TABLESPACE command, 4 - 329
NOMAXVALUE option
of CREATE SEQUENCE command, 4 - 282
NOMINVALUE option
of CREATE SEQUENCE command, 4 - 282
NONE option
of SET ROLE command, 4 - 517
non-padded comparison semantics, 2-25
NOORDER option
of CREATE SEQUENCE command, 4 - 283
NORESETLOGS option
of CREATE CONTROLFILE command, 4 - 216
NOREVERSE option
of ALTER INDEX command, 4 - 32
normal exit from Oracle7, 4 - 187
NORMAL option
of ALTER TABLESPACE command, 4 - 137
NOSORT option
of CREATE INDEX command, 4 - 241, 4 - 244
NOT DEFERRABLE option
of CONSTRAINT clause, 4 - 191
NOT IN comparison operator, 3-5
examples, 3-7
NOT LIKE comparison operator, 3-5
NOT logical operator, 3-8
truth table, 3-12

NOT NULL clause
ALTER TABLE command, 4 - 124
NOT NULL constraints, 4 - 193
NOT option
of WHENEVER clause, 4 - 171, 4 - 179, 4 - 180, 4 - 461, 4 - 464
NOVALIDATE option
of ENABLE clause, 4 - 419
of CONSTRAINT clause, 4 - 192
NOWAIT option
of FOR UPDATE clause, 4 - 493
of LOCK TABLE command, 4 - 459
NULL
constraint, example, 4 - 193
in a condition, 3-94
null, 2-31
in a bitmap index, 4 - 245
in an index, 4 - 245
NULL value
of OPTIMAL parameter, 4 - 276, 4 - 526
number
comparison rules, 2-24
literal, 2-2
NUMBER datatype, 2-10
comparing values of, 2-24
number format elements, 3-65
number format models, 3-65
examples, 3-63
numeric
literal, 2-2
NVARCHAR2 datatype, 2-9
NVL function, 2-31, 3-53

O

object auditing options, 4 - 174, 4 - 179
object auditing shortcuts, 4 - 180
object privileges, 4 - 446
granting to users and roles, 4 - 444
on synonyms, 4 - 448
on tables, 4 - 447
revoking from users and roles, 4 - 478
object reference
functions, 3-56
object tables, 4 - 324
creating, 4 - 312
index on object identifier column, 4 - 314
object type bodies
creating, 4 - 353
dropping, 4 - 412
object type column constraints, 4 - 326
object types, 2-23
comparing values of, 2-27
creating, 4 - 345
dropping, 4 - 410
revoking privileges, 4 - 479
object views
creating, 4 - 363, 4 - 370
example, 4 - 371
OF datatype clause
of CREATE TYPE command, 4 - 349
OFFLINE option
of ALTER ROLLBACK SEGMENT command, 4 - 54
of ALTER TABLESPACE command, 4 - 137
of CREATE TABLESPACE command, 4 - 330
of DATAFILE clause, 4 - 23
OIDINDEX clause
of CREATE TABLE command, 4 - 314
ON clause
of CREATE TRIGGER command, 4 - 335, 4 - 338
of GRANT command, 4 - 445
of NOAUDIT command, 4 - 463
of REVOKE command, 4 - 479
ON DELETE CASCADE option
of CONSTRAINT clause, 4 - 197
ONLINE option
of ALTER ROLLBACK SEGMENT command, 4 - 54
of ALTER TABLESPACE command, 4 - 137
of CREATE TABLESPACE command, 4 - 330
of DATAFILE clause, 4 - 23
OPEN option
of ALTER DATABASE command, 4 - 18
OPEN_LINKS, 4 - 227
opening
databases, 4 - 18
operator
arithmetic, 3-3
character, 3-4

definition, 3-1
logical, 3-5, 3-11
miscellaneous, 3-16
NOT IN, 3-7
set, 3-12
use in expression, 3-78
optimal size
of rollback segments, 4 - 276, 4 - 526
optimizer
hints, 2-40
SQL, 1- 3
OPTIMIZER_MODE parameter, 4 - 71
of ALTER SESSION command, 4 - 66
options_clause
of ALTER SYSTEM, 4 - 93
OR logical operator
truth table, 3-12
OR REPLACE option
of CREATE FUNCTION command, 4 - 234
of CREATE PACKAGE BODY command, 4 - 255
of CREATE PACKAGE command, 4 - 250
of CREATE PROCEDURE command, 4 - 261
of CREATE TRIGGER command, 4 - 334
of CREATE TYPE command, 4 - 348
of CREATE VIEW command, 4 - 364
ORACLE identifiers
how to form, A-3
ORDER BY clause
and ROWNUM pseudocolumn, 2-38
of SELECT command, 4 - 493, 4 - 501
ORDER MEMBER clause of CREATE TRIGGER command, 4 - 350
ORDER method
object value comparisons, 2-28
ORDER option
of CREATE SEQUENCE command, 4 - 283
ORGANIZATION HEAP option
of CREATE TABLE command, 4 - 317
ORGANIZATION INDEX option
of CREATE TABLE command, 4 - 316
OS_AUTHENT_PREFIX
initialization parameter, 4 - 361
OS_ROLES, 4 - 442
out-of-line LOB storage, 4 - 117, 4 - 317
outer join, 3-16, 4 - 508
examples, 4 - 509
of SELECT, 4 - 508
OVERFLOW clause
of ALTER TABLE command, 4 - 117
of CREATE TABLE command, 4 - 317
overflow_clause
of ALTER TABLE, 4 - 113
overloading
procedures and stored functions, 2-49, 4 - 251

P

package bodies
dropping, 4 - 394
PACKAGE option
of ALTER PACKAGE command, 4 - 38
package specifications, 4 - 250
packages, 4 - 250, 4 - 251, 4 - 254
adding procedures to, 4 - 251
adding stored functions to, 4 - 251
creating, 4 - 250, 4 - 254
creating package bodies for, 4 - 254
creating package specifications for, 4 - 250
dropping, 4 - 394
redefining, 4 - 250, 4 - 255
removing procedures from, 4 - 396
removing stored functions from, 4 - 390
packages bodies, 4 - 254
PARALLEL clause, 4 - 465
of RECOVER clause, 4 - 471
parallel DML
enabling and disabling for sessions, 4 - 62
PARALLEL DML option
of ALTER SESSION command, 4 - 62
parallel query clause, 4 - 13
of CREATE TABLE command, 4 - 319
parallel server
setting the instance, 4 - 72
parallel_clause
of ALTER SNAPSHOT command, 4 - 80, 4 - 86
of ALTER TABLE command, 4 - 122
of CREATE CLUSTER, 4 - 207
PARALLEL_DEFAULT_SCANSIZE parameter, 4 - 466

PARAMETERS clause
of CREATE FUNCTION command, 4 - 235
of CREATE PROCEDURE command, 4 - 262
parentheses
around expressions, 3-90
overriding operator precedence, 3-3
PARTITION BY RANGE clause
of CREATE TABLE command, 4 - 319
PARTITION clause
of UPDATE command, 4 - 544
PARTITION option
of ANALYZE command, 4 - 158
partition views, 4 - 370
partition_description_clause
of ALTER INDEX, 4 - 31
partition_start column
of EXPLAIN PLAN, 4 - 428
partition_stop column
of EXPLAIN PLAN, 4 - 428
partitioned indexes
altering, 4 - 34
definition, 4 - 237
partitioned snapshots, 4 - 83, 4 - 296
partitioned table analysis
EXPLAIN PLAN, 4 - 427
partitioned table update, 4 - 546
partitioned tables
creating, 4 - 319
partitioning
partitioned indexes, 4 - 246
partitioning_clauses
of ALTER TABLE, 4 - 113
partitions
efficient deletes from, 4 - 378
password history parameters, 4 - 45
PASSWORD_GRACE_TIME option
of CREATE PROFILE command, 4 - 267
PASSWORD_LIFE_TIME option
of CREATE PROFILE command, 4 - 267
PASSWORD_LOCK_TIME option
of CREATE PROFILE command, 4 - 267
PASSWORD_REUSE_MAX option
of CREATE PROFILE command, 4 - 267
PASSWORD_REUSE_TIME option
of CREATE PROFILE command, 4 - 267
PASSWORD_VERIFY_FUNCTION option
of CREATE PROFILE command, 4 - 268
passwords
changing, 4 - 150
establishing for users, 4 - 272, 4 - 359
pattern matching
definition, 3-8
PCTFREE parameter
of ALTER CLUSTER command, 4 - 12
of ALTER SNAPSHOT command, 4 - 79
of ALTER TABLE command, 4 - 85, 4 - 117
of CREATE CLUSTER command, 4 - 208
of CREATE INDEX command, 4 - 241
of CREATE SNAPSHOT command, 4 - 289, 4 - 299
of CREATE TABLE command, 4 - 314
PCTINCREASE parameter
of STORAGE clause, 4 - 525
PCTTHRESHOLD option
of ALTER TABLE command, 4 - 117
of CREATE TABLE command, 4 - 317
PCTUSED parameter
of ALTER CLUSTER command, 4 - 12
of ALTER SNAPSHOT command, 4 - 79
of ALTER TABLE command, 4 - 85, 4 - 117
of CREATE CLUSTER command, 4 - 208
of CREATE SNAPSHOT command, 4 - 289, 4 - 299
of CREATE TABLE command, 4 - 314
PCTVERSION parameter
of LOB storage clause, 4 - 318
physical_attributes_clause
of ALTER CLUSTER, 4 - 11
of ALTER TABLE, 4 - 110
of CREATE CLUSTER, 4 - 208
of CREATE TABLE, 4 - 310
PL/SQL
functions, 3-60
PLSQL_V2_COMPATABILITY option
of ALTER SESSION command, 4 - 66
of ALTER SYSTEM command, 4 - 96
PM / P.M. format element, 3-72
POST_TRANSACTION option
of DISCONNECT SESSION clause
of ALTER SYSTEM command, 4 - 96

POWER number function, 3-22
PR format element, 3-65
PRAGMA RESTRICT_REFERENCES clause
  of ALTER TYPE command, 4 - 147
  of CREATE TYPE command, 4 - 351
pragma_clause
  of ALTER TYPE, 4 - 145
  of CREATE TYPE, 4 - 347
precedence, 3-2
precision
  of NUMBER columns, 2-10
primary key
  snapshots, 4 - 295
PRIMARY KEY option
  of ADD clause
    of ALTER SNAPSHOT LOG command, 4 - 86
  of DISABLE clause, 4 - 381
  of DROP clause, 4 - 384
  of ENABLE clause, 4 - 419
  of REFRESH clause, 4 - 80, 4 - 290
primary keys, 4 - 190, 4 - 195
PRIOR operator, 3-16
PRIVATE_SGA parameter
  of ALTER RESOURCE COST command, 4 - 48
privilege domain
  changing, 4 - 517
privileges, 4 - 446
procedure_specification parameter
  of ALTER TYPE command, 4 - 146
  of CREATE TYPE command, 4 - 350
procedures
  adding to packages, 4 - 251
  creating, 4 - 259
  definition, 4 - 259, 4 - 262
  dropping, 4 - 248, 4 - 396
  granting object privileges on, 4 - 448
  overloading, 2-49, 4 - 251
  recompiling, 4 - 41
  redefining, 4 - 259
  removing from packages, 4 - 396
PROFILE clause
  of CREATE USER command, 4 - 359
profiles
  adding resource limits to, 4 - 43, 4 - 265
  altering, 4 - 43
  assigning to users, 4 - 150, 4 - 359
  creating, 4 - 265
  DEFAULT profile, 4 - 269, 4 - 359, 4 - 398
  definition, 4 - 265, 4 - 268
  dropping, 4 - 398
  modifying resource limits in, 4 - 43
  PUBLIC_DEFAULT profile, 4 - 46, 4 - 48
  removing resource limits from, 4 - 43
pseudocolumns, 2-32
PUBLIC option
  of CREATE DATABASE LINK command, 4 - 226
  of CREATE ROLLBACK SEGMENT command, 4 - 275
  of CREATE SYNONYM command, 4 - 302
  of DROP DATABASE LINK command, 4 - 388
  of DROP SYNONYM command, 4 - 404
  of ENABLE clause, 4 - 22
  of FROM clause, 4 - 475, 4 - 479
  of TO clause, 4 - 435, 4 - 446
public rollback segments, 4 - 275
PUBLIC_DEFAULT profile, 4 - 46, 4 - 48
punctuation
  in date format models, 3-76

Q

queries, 4 - 530
  examples, 4 - 533
  SELECT, 4 - 489
QUOTA clause
  multiple in CREATE USER statement, 4 - 360
  of CREATE USER command, 4 - 359, 4 - 360
quote marks
  using in text literals, 2-2
quoted names, 2-49

R

RAW datatype, 2-15
RAWTOHEX conversion function, 3-43
RDBMS (relational database management system), 1- 1
read consistency

default, 4 - 520
READ ONLY option
of SET TRANSACTION command, 4 - 519
READ WRITE option
of SET TRANSACTION command, 4 - 519
REBUILD UNUSABLE LOCAL INDEXES option
of ALTER SNAPSHOT command, 4 - 80
of ALTER TABLE command, 4 - 120
recompiling
procedures, 4 - 41
stored functions, 4 - 26
RECOVER clause, 4 - 469
examples, 4 - 471
of ALTER DATABASE command, 4 - 19
parallel_clause. See PARALLEL clause.
recoverability
of tables, 4 - 316
recovery
disabling for distributed transactions, 4 - 88, 4 - 103
enabling for distributed transactions, 4 - 88, 4 - 103
RECYCLE option
of BUFFER_POOL option
of STORAGE clause, 4 - 526
redefining
columns, 4 - 116
packages, 4 - 250, 4 - 255
procedures, 4 - 261
stored functions, 4 - 234
redo log file groups
adding members to, 4 - 20
adding to threads, 4 - 19
assigning to redo log threads, 4 - 216
dropping, 4 - 20
dropping members from, 4 - 20
maximum number
of members, 4 - 222
redo log file members
adding to redo log file groups, 4 - 20
dropping from redo log file groups, 4 - 20
maximum number
for redo log file groups, 4 - 222
for redo log files, 4 - 219
renaming, 4 - 21
specifying, 4 - 431
redo log files
adding to databases, 4 - 219, 4 - 221
archiving, 4 - 19, 4 - 219, 4 - 222
maximum number
for databases, 4 - 219, 4 - 221
of members, 4 - 219
specifying, 4 - 431
switching, 4 - 88, 4 - 102
redo log threads
adding redo log file groups to, 4 - 19
assigning redo log file groups to, 4 - 216
disabling, 4 - 22
dropping redo log file groups from, 4 - 20
enabling, 4 - 22
REF clause
of CREATE TYPE command, 4 - 349
REF columns
adding to tables, 4 - 127
creating in tables, 4 - 325
REF constructor
expression syntax, 3-87
REFERENCES object privilege
on tables, 4 - 447
REFERENCES options
of CONSTRAINT clause, 4 - 197
REFERENCING clause
of CREATE TRIGGER command, 4 - 335
referential integrity constraints, 4 - 190, 4 - 196
defining, 4 - 198
dropping, 4 - 479
maintaining, 4 - 199
REFRESH clause
of ALTER SNAPSHOT command, 4 - 80
of CREATE SNAPSHOT command, 4 - 290
refresh modes
for snapshots, 4 - 292
refresh times
for snapshots, 4 - 293
refreshing
snapshots, 4 - 80, 4 - 290
snapshots with snapshot logs, 4 - 299
REFs, 2-23
DANGLING, 2-23
scoped, 4 - 325

REFTOHEX function, 3-56
remote query, 4 - 536
remote table
identifying, 4 - 536
removing
comments from objects, 4 - 183
integrity constraints from columns, 4 - 116, 4 - 123
procedures from packages, 4 - 396
resource limits from profiles, 4 - 43
stored functions from packages, 4 - 390
RENAME command, 4 - 473
examples, 4 - 473
RENAME FILE clause
of ALTER DATABASE command, 4 - 21
RENAME object auditing option, 4 - 174
RENAME option
of ALTER TABLE command, 4 - 120
RENAME PARTITION option
of ALTER SNAPSHOT command, 4 - 79, 4 - 85
of ALTER TABLE command, 4 - 121
renaming
datafiles, 4 - 21, 4 - 133
objects, 4 - 473
redo log file members, 4 - 21
REPLACE character function, 3-29
REPLACE option
of ALTER TYPE command, 4 - 145
RESETLOGS option
of ALTER DATABASE command, 4 - 18
of CREATE CONTROLFILE command, 4 - 216
RESIZE clause
of ALTER DATABASE command, 4 - 23
resource limits
adding to profiles, 4 - 43, 4 - 265
assigning to users, 4 - 150
costs of resources, 4 - 48
disabling, 4 - 88, 4 - 98
enabling, 4 - 88, 4 - 98, 4 - 269
exceeding, 4 - 268
modifying, 4 - 43
removing profiles from, 4 - 43
RESOURCE role, 4 - 273
RESOURCE statement auditing short cut, 4 - 176
RESOURCE_LIMIT option
of ALTER SYSTEM command, 4 - 98
RETURNING clause
of DELETE command, 4 - 376, 4 - 545
of INSERT command, 4 - 453
of the DELETE command, 4 - 378
of the UPDATE command, 4 - 549
retrieving deleted rows, 4 - 378
retrieving inserted rows, 4 - 455
retrieving updated rows, 4 - 549
returning_clause
of DELETE, 4 - 375
of INSERT, 4 - 452
of UPDATE, 4 - 544
REUSE option
of BACKUP CONTROLFILE clause, 4 - 21
of CREATE CONTROLFILE command, 4 - 216
of filespec, 4 - 432
REUSE STORAGE option
of TRUNCATE command, 4 - 539
REVERSE option
of ALTER INDEX command, 4 - 32
of CREATE INDEX command, 4 - 241
REVOKE command, 4 - 475, 4 - 478
examples, 4 - 477, 4 - 481
revoking
object privileges from users and roles, 4 - 478
object type privileges, 4 - 479
system privileges and roles from users, roles, 4 - 475
RN format element, 3-65
RNDS parameter
of ALTER TYPE command, 4 - 147
of CREATE TYPE command, 4 - 351
RNPS parameter
of ALTER TYPE command, 4 - 147
of CREATE TYPE command, 4 - 351
roles
CONNECT role, 4 - 273
creating, 4 - 272
DBA role, 4 - 273
definition, 4 - 272
DELETE_CATALOG_ROLE role, 4 - 274
dropping, 4 - 399
enabling or disabling for sessions, 4 - 516
establishing default roles for users, 4 - 150, 4 -

152
EXECUTE_CATALOG_ROLE role, 4 - 274
granting, 4 - 435
granting object privileges to, 4 - 444
granting system privileges and roles to, 4 - 434
granting to users and roles, 4 - 434
predefined by Oracle8, 4 - 273
RESOURCE role, 4 - 273
revoking from users and roles, 4 - 475
revoking object privileges from, 4 - 478
revoking system privileges and roles from, 4 - 475
SELECT_CATALOG_ROLE role, 4 - 274
roll back
to the same savepoint multiple times, 4 - 485
transactions, 4 - 484
ROLLBACK command, 4 - 484
ending a transaction, 4 - 485
examples, 4 - 485
summary, 4 - 8
ROLLBACK option
of ADVISE clause, 4 - 62
ROLLBACK SEGMENT option
of USING INDEX clause, 4 - 289
rollback segments
altering, 4 - 53
creating, 4 - 275
definition, 4 - 275
dropping, 4 - 400
optimal size of, 4 - 276, 4 - 526
shrinking size, 4 - 54, 4 - 276, 4 - 526
snapshots, 4 - 294
specifying tablespaces for, 4 - 275, 4 - 276
storage characteristics of, 4 - 276
SYSTEM rollback segment, 4 - 328
taking online and offline, 4 - 54, 4 - 400
ROLLBACK_SEGMENTS, 4 - 54
ROUND date function, 3-39, 3-40
format models for, 3-40
ROUND number function, 3-22
rounding numeric data, 2-11
by using scale, 2-11
row address
ROWID, 2-18
row exclusive locks, 4 - 459
row share locks, 4 - 459
ROWID
description of, 2-18
pseudocolumn, 2-36
ROWID option
of ADD clause
of ALTER SNAPSHOT LOG command, 4 - 86
ROWIDTOCHAR conversion function, 3-44
ROWNUM pseudocolumn, 2-37
and ORDER BY clause, 2-38
rows
accessing via ROWID, 2-36
deleting from tables and views, 4 - 374
identifying with ROWID values, 2-37
inserting into tables and views, 4 - 451
ordering, 4 - 501
selecting from tables, 4 - 489
updating, 4 - 542
RPAD character function, 3-29
RR date format element, 3-72
RTRIM character function, 3-30
RX locks, 4 - 459

S

S format element, 3-65
SAMPLE parameter
of ANALYZE command, 4 - 158
SAVEPOINT command, 4 - 487
examples, 4 - 488
summary, 4 - 8
savepoints
creating, 4 - 487
erasing with COMMIT command, 4 - 185
scalar
definition, 4 - 146, 4 - 350
scale
negative, 2-11
of NUMBER columns, 2-10
SCAN_INSTANCES parameter
of ALTER SYSTEM command, 4 - 94
schema object name
qualifiers, 2-47
schema objects

definition of, 2-44
namespaces for, 2-48
naming rules, 2-47
schemas
creating, 4 - 278
SCOPE clause
defined, 4 - 325
SCOPE IS clause
of CREATE TABLE command, 4 - 313, 4 - 314
searching
for rows with an index, 4 - 243
security
provided by views, 4 - 366
security domains, 4 - 152
segment_attributes_clause
of CREATE TABLE, 4 - 309
segment_partition_clause
of ALTER TABLE, 4 - 115
SELECT
outer_join, 4 - 508
SELECT clause
INSERT command, 4 - 454
UPDATE command, 4 - 545, 4 - 548
SELECT command, 4 - 489
examples, 2-35, 4 - 494, 4 - 497, 4 - 500, 4 - 502, 4 - 503, 4 - 504, 4 - 506, 4 - 533, 4 - 537
summary, 4 - 7
WITH_clause, 4 - 491
select list, 4 - 493
SELECT object auditing option, 4 - 174
SELECT object privilege
on sequences, 4 - 448
on tables, 4 - 447
on views, 4 - 447
SELECT_CATALOG_ROLE role, 4 - 274
self joins, 4 - 507
SEQ parameter
of ARCHIVE LOG clause, 4 - 168
SEQUEL (Structured English Query Language), 1-1
sequences, 4 - 281
accessing values of, 2-33, 4 - 536
altering, 4 - 56
creating, 4 - 281
cycling values of, 4 - 284
dropping, 4 - 401
increment between values, 4 - 56, 4 - 282
incrementing values of, 2-33, 4 - 536
limiting values of, 4 - 284
losing values of, 4 - 285
performance benefits of, 4 - 283
renaming, 4 - 473
restarting, 4 - 401
skipping values of, 4 - 283
SERIALIZABLE option
of ISOLATION_LEVEL parameter of ALTER SESSION command, 4 - 64
serializing transactions, 4 - 64
session control commands, 4 - 8
session cursors, 4 - 72
SESSION_CACHED_CURSORS parameter, 4 - 72
of ALTER SESSION command, 4 - 66
SESSIONID option
of USERENV function, 3-55
sessions
altering, 4 - 58
closing database links for, 4 - 72
disconnecting, 4 - 96, 4 - 104
enabling and disabling error reporting of unusable indexes, 4 - 66
enabling and disabling roles for, 4 - 516
enabling and disabling SQL trace facility for, 4 - 66, 4 - 67
National Language Support (NLS) settings for, 4 - 64, 4 - 67
terminating, 4 - 96, 4 - 103
SET clause
of EXPLAIN PLAN command, 4 - 426
of UPDATE command, 4 - 545
SET CONSTRAINT(S) command, 4 - 514
SET DATABASE parameter
of CREATE CONTROLFILE command, 4 - 216
set operators, 4 - 501
SET ROLE command, 4 - 516
examples, 4 - 518
SET TRANSACTION command, 4 - 519
examples, 4 - 521
summary, 4 - 8
set_clause
of ALTER SYSTEM, 4 - 90

share locks, 4 - 459
share row exclusive locks, 4 - 459
share update locks, 4 - 459
SHARED option
of CREATE DATABASE LINK command, 4 - 226
shared pool
clearing, 4 - 97
shared server processes
creating and terminating, 4 - 99
shared SQL area
session cursors, 4 - 71
SHRINK clause
of ALTER ROLLBACK SEGMENT command, 4 - 54
shrinking
rollback segments, 4 - 276, 4 - 526
simple join
example, 4 - 506
simple snapshots, 4 - 291
simultaneous update and query on tables, 4 - 520
SIN number function, 3-23, 3-24
SINH number function, 3-23
SIZE parameter
of ALLOCATE EXTENT clause, 4 - 12, 4 - 33, 4 - 118
of ALTER CLUSTER command, 4 - 12
of CREATE CLUSTER command, 4 - 209, 4 - 212
of filespec, 4 - 432
SKIP_UNUSABLE_INDEXES option
of ALTER SESSION command, 4 - 66
snapshot logs
altering, 4 - 84
creating, 4 - 297
definition, 4 - 297
dropping, 4 - 403
storage characteristics of, 4 - 299
snapshot refresh, job queue processes, 4 - 293
SNAPSHOT_REFRESH_INTERVAL, 4 - 293
SNAPSHOT_REFRESH_KEEP_CONNECTIONS, 4 - 293
SNAPSHOT_REFRESH_PROCESSES, 4 - 293
snapshots
adding comments to, 4 - 183
adding to clusters, 4 - 289
altering, 4 - 76
complex, 4 - 291
creating, 4 - 286
definition, 4 - 286, 4 - 291
dropping, 4 - 402
partitioned, 4 - 83, 4 - 296
primary key, 4 - 295
refresh modes, 4 - 292
refresh times, 4 - 293
refreshing, 4 - 80, 4 - 290
refreshing with snapshot logs, 4 - 299
removing comments from, 4 - 183
rollback segments, 4 - 294
rowid, 4 - 295
simple, 4 - 291
storage characteristics of, 4 - 79, 4 - 289
types of, 4 - 291
SOME comparison operator, 3-5
SOUNDEX character function, 3-30
space
deallocating, 4 - 372
SPECIFICATION option
of COMPILE clause of ALTER TYPE command, 4 - 145
SPLIT PARTITION option
of ALTER SNAPSHOT command, 4 - 79, 4 - 86
of ALTER TABLE command, 4 - 121
split_partition_clause
of ALTER INDEX, 4 - 31
of ALTER TABLE, 4 - 115
SQL (Structured Query Language), 1- 1
conversion functions, 3-42
embedded, 1- 3
functions, 3-16, 3-78
history of, 1- 1
lexical conventions, 1- 4
optimizer, 1- 3
standards, 1- 2
summary of commands, 4 - 2
unified language, 1- 3
SQL functions
character, 3-25
group, 3-57
SQL trace facility
enabling and disabling for sessions, 4 - 66, 4 - 67

SQL/DS
datatypes, 2-20
SQL_TRACE, 4 - 67
SQL2, 1- 2
SQL92, 1- 2
SRX locks, 4 - 459
standard deviation, 3-59
standby database
RECOVER clause, 4 - 470
START WITH clause
of CREATE SEQUENCE command, 4 - 282
of SELECT command, 4 - 492, 4 - 496, 4 - 497
START WITH parameter
of REFRESH clause, 4 - 80, 4 - 290
statement auditing options, 4 - 172
statement auditing shortcuts, 4 - 176
states of constraints, 4 - 205, 4 - 419
STDDEV group function, 3-59
storage
ALTER TABLESPACE, 4 - 133
storage characteristics
of clusters, 4 - 11, 4 - 208, 4 - 209
of indexes, 4 - 28, 4 - 241
of rollback segments, 4 - 276
of snapshot logs, 4 - 299
of snapshots, 4 - 79, 4 - 289
of tables, 4 - 314, 4 - 315
STORAGE clause, 4 - 523
examples, 4 - 527
of ALTER CLUSTER command, 4 - 12
of ALTER INDEX command, 4 - 32
of ALTER TABLE command, 4 - 117
of CREATE CLUSTER command, 4 - 209
of CREATE INDEX command, 4 - 241
of CREATE ROLLBACK SEGMENT command, 4 - 276
of CREATE SNAPSHOT command, 4 - 289, 4 - 299
of CREATE TABLE command, 4 - 315
stored functions
adding to packages, 4 - 251
creating, 4 - 232
definition, 4 - 232
dropping, 4 - 390
overloading, 2-49, 4 - 251
PL/SQL, 3-60
recompiling, 4 - 26
redefining, 4 - 234
removing from packages, 4 - 390
stored procedures, 4 - 259
storing
LOBs, 4 - 117, 4 - 317
nested tables, 4 - 118
string to date conversion, 3-77
subqueries, 4 - 530
correlated, 4 - 534
flattened, 4 - 533
of CREATE VIEW, 4 - 364
of DELETE, 4 - 375
WITH_clause, 4 - 531
SUBSTR character function, 3-31
SUM group function, 3-59, 3-60
suppressing blank padding
in date format models, 3-75
SWITCH LOGFILE option
of ALTER SYSTEM command, 4 - 102
switching
redo log files, 4 - 88, 4 - 102
SYEAR date format element, 3-72
synonyms
auditing, 4 - 179
creating, 4 - 302
definition, 4 - 302
dropping, 4 - 404
granting object privileges on, 4 - 448
renaming, 4 - 473
scope of, 4 - 304
using with database links, 4 - 228
SYSDATE date function, 3-40
system change numbers
specifying for forced transactions, 4 - 186
system control commands, 4 - 9
system privileges
granting, 4 - 435
granting to users and roles, 4 - 434
revoking from users and roles, 4 - 475
SYSTEM rollback segment, 4 - 328
SYSTEM tablespace, 4 - 328, 4 - 331

T

table alias, 4 - 547
table constraints, 4 - 188
  example, 4 - 203
TABLE option
  of ANALYZE command, 4 - 158
  of subquery, 4 - 532
  of TRUNCATE command, 4 - 538
table partitions
  modifying, 4 - 128
table_partition_clause
  of CREATE TABLE, 4 - 312
table_ref_clause
  of ALTER TABLE, 4 - 109
  of CREATE TABLE, 4 - 309
tables
  adding columns to, 4 - 106, 4 - 116, 4 - 122
  adding comments to, 4 - 183
  adding integrity constraints to, 4 - 116, 4 - 123
  adding LOB columns to, 4 - 125
  adding to clusters, 4 - 318
  adding triggers to, 4 - 333
  aliases for, 4 - 492
  allocating extents for, 4 - 106
  allowing writes, 4 - 106
  altering, 4 - 106
  creating, 4 - 306
  creating snapshot logs for, 4 - 297
  creating views on, 4 - 363
  definition, 4 - 306
  deleting rows from, 4 - 374, 4 - 405, 4 - 538
  disallowing writes, 4 - 106
  dropping, 4 - 405
  dropping integrity constraints from, 4 - 384
  dropping triggers from, 4 - 409
  granting object privileges on, 4 - 447
  index-organized, creating, 4 - 306
  inserting rows into, 4 - 451
  LOB storage characteristics of, 4 - 117, 4 - 317
  locking, 4 - 458
  recoverability, 4 - 316
  redefining columns of, 4 - 106
  removing comments from, 4 - 183
  removing from clusters, 4 - 386
  renaming, 4 - 473
  selecting data from, 4 - 489
  specifying tablespaces for, 4 - 315
  storage characteristics of, 4 - 106, 4 - 117, 4 - 306, 4 - 314, 4 - 315
  truncating, 4 - 538
  updating rows in, 4 - 542
TABLESPACE option
  of CREATE CLUSTER command, 4 - 209
  of CREATE INDEX command, 4 - 241
  of CREATE ROLLBACK SEGMENT command, 4 - 275
  of CREATE SNAPSHOT command, 4 - 289, 4 - 299
  of CREATE TABLE command, 4 - 315
  of RECOVER clause, 4 - 471
tablespaces, 4 - 328
  altering, 4 - 133
  assigning to users, 4 - 150
  backing up, 4 - 133, 4 - 138
  changing future storage allocations, 4 - 133
  creating, 4 - 328, 4 - 331
  creating clusters in, 4 - 209
  creating indexes in, 4 - 241
  creating rollback segments in, 4 - 275, 4 - 276
  datafiles of, 4 - 133, 4 - 136, 4 - 329
  dropping, 4 - 407
  establishing default tablespaces for users, 4 - 150
  establishing tablespace quotas for users, 4 - 359, 4 - 360
  establishing temporary tablespaces for users, 4 - 150
  specifying for tables, 4 - 315
  SYSTEM tablespace, 4 - 328, 4 - 331
  taking online and offline, 4 - 133, 4 - 137, 4 - 330
TAN number function, 3-24
TANH number function, 3-24
TEMPORARY option
  of ALTER TABLESPACE command, 4 - 137
TERMINAL option
  of USERENV function, 3-55
terminating
  dispatcher processes (DISP), 4 - 100
  sessions, 4 - 96, 4 - 103

shared shadow processes, 4 - 99
text
definition of, 2-2
THE keyword
in flattened subqueries, 4 - 533
of SELECT command, 4 - 492
of subquery, 4 - 532
THREAD parameter
of ADD LOGFILE clause, 4 - 19
of ARCHIVE LOG clause, 4 - 168
threads
redo log threads, 4 - 221
TO clause
of GRANT command, 4 - 446
of ROLLBACK command, 4 - 484
TO parameter
of ARCHIVE LOG clause, 4 - 168
TO_CHAR conversion function, 3-44
examples, 3-63, 3-64, 3-76
TO_DATE conversion function, 3-46
TO_MULTI_BYTE conversion function, 3-47
TO_NUMBER conversion function, 3-47
TO_SINGLE_BYTE conversion function, 3-47
transaction control commands, 4 - 8
transactions, 4 - 186
committing, 4 - 185
distributed, 4 - 186, 4 - 486
establishing as read-only, 4 - 519
read consistency, 4 - 520
read only, 4 - 521
rolling back, 4 - 484
serializing, 4 - 64
TRANSLATE character function, 3-32
TRANSLATE USING conversion function, 3-48
triggered action, 4 - 339
triggers
creating, 4 - 333
definition, 4 - 333
dropping from tables, 4 - 409
enabling and disabling, 4 - 142
executing, 4 - 336
firing, 4 - 336
INSTEAD OF, 4 - 342
LONG datatype, 2-13
types of, 4 - 339
TRUE
result of a condition, 3-90
TRUNC date function, 3-40
format models for, 3-40
TRUNC number function, 3-25
TRUNCATE command, 4 - 538
examples, 4 - 540
TRUNCATE PARTITION option
of ALTER TABLE command, 4 - 121
truncating
clusters, 4 - 538
tables, 4 - 538
truth tables, 3-12
type bodies
compiling, 4 - 145
type constructor
expression syntax, 3-82
type specifications
compiling, 4 - 145
types
user-defined, 2-22

U

UID function, 3-54
UNARCHIVED option
of CLEAR LOGFILE clause, 4 - 20
undo a transaction, 4 - 484
UNION ALL set operator, 4 - 493
examples, 3-14
UNION set operator, 4 - 493
examples, 3-14
unique keys, 4 - 190, 4 - 193
UNIQUE option
of DISABLE clause, 4 - 381
of ENABLE clause, 4 - 419
UNLIMITED clause
of ALTER DATABASE command, 4 - 23
UNLIMITED option
of ALTER PROFILE command, 4 - 44
of CREATE PROFILE command, 4 - 268
of QUOTA clause, 4 - 359
UNTIL CANCEL option
of RECOVER clause, 4 - 471
UNTIL CHANGE parameter

of RECOVER clause, 4 - 471
UNTIL TIME parameter
of RECOVER clause, 4 - 471
UNUSABLE LOCAL INDEXES option
of ALTER SNAPSHOT command, 4 - 80
of ALTER TABLE command, 4 - 120
UPDATE command, 4 - 542, 4 - 548
examples, 4 - 547
returning_clause, 4 - 544
subquery, 4 - 545
UPDATE object auditing options, 4 - 174
UPDATE object privilege
on tables, 4 - 447
on views, 4 - 447
UPDATE option
of CREATE TRIGGER command, 4 - 335
updating
rows in tables and views, 4 - 542
simple snapshots, 4 - 290
UPPER character function, 3-33
uppercase
significance in SQL statements, 1- 5
uppercase and lowercase
significance in pattern matching, 3-9
user access verification
security domains, 4 - 152
user authentication
changing, 4 - 150
USER function, 3-54
user functions, 3-60, 4 - 232
expression syntax, 3-81
USER_CLUSTERS view, 4 - 163
USER_INDEXES view, 4 - 161
USER_TAB_COLUMNS view, 4 - 163
USER_TABLES view, 4 - 161
user-defined types, 2-22
nested tables, 2-24
object types, 2-23
REFs, 2-23
VARRAYs, 2-23
USERENV function, 3-54
users
altering, 4 - 150
assigning profiles to, 4 - 150, 4 - 359
assigning resource limits to, 4 - 150
assigning tablespaces to, 4 - 150
changing authentication mechanism, 4 - 150
changing passwords, 4 - 150
creating, 4 - 357
definition, 4 - 357
dropping, 4 - 414
establishing default roles for, 4 - 150, 4 - 152
establishing default tablespaces for, 4 - 150
establishing passwords for, 4 - 272, 4 - 359
establishing tablespace quotas for, 4 - 359, 4 - 360
establishing temporary tablespaces for, 4 - 150
granting object privileges to, 4 - 444
granting system privileges and roles to, 4 - 434
revoking object privileges from, 4 - 478
revoking system privileges and roles from, 4 - 475
USING clause
of CREATE DATABASE LINK command, 4 - 226
USING INDEX option
of ALTER SNAPSHOT command, 4 - 80
of CONSTRAINT clause, 4 - 191
of CREATE SNAPSHOT command, 4 - 289
of ENABLE clause, 4 - 419
USING MASTER ROLLBACK SEGMENT clause
of ALTER SNAPSHOT command, 4 - 80
using_index_clause
of ENABLE clause, 4 - 418
ENABLE clause
using_index_clause, 4 - 418
UTLEXCPT.SQL, 4 - 421
UTLSAMPL.SQL, xxii
UTLXPLAN.SQL, 4 - 426

V

V format element, 3-65
V$LOG table, 4 - 19, 4 - 221
V$NLS_PARAMETERS table, 4 - 67
VALIDATE option
of ENABLE clause, 4 - 418
of CONSTRAINT clause, 4 - 192
VALIDATE REF UPDATE option
of ANALYZE command, 4 - 158

validating constraints, 4 - 192
value
use in expression, 3-78
VALUE operator
expression syntax, 3-87
VALUES clause
of INSERT command, 4 - 453, 4 - 454
VALUES LESS THAN clause
of CREATE TABLE command, 4 - 319
VARCHAR datatype, 2-10
VARCHAR2 datatype, 2-9
similarity to RAW datatype, 2-15
VARGRAPHIC datatype, 2-22
variable length
date format models, 3-75
VARRAYs, 2-23
views
adding comments to, 4 - 183
and DML commands, 4 - 367
creating, 4 - 363
definition, 4 - 363
deleting rows from, 4 - 374
dropping, 4 - 416
inherently updatable, 4 - 367
inserting rows into, 4 - 451
join views, 4 - 368
updatable, 4 - 368
locking, 4 - 458
partition, 4 - 370
redefining, 4 - 416
removing comments from, 4 - 183
renaming, 4 - 473
updating rows in, 4 - 542
uses of, 4 - 366
VSIZE function, 3-55

W

WHEN clause
of CREATE TRIGGER command, 4 - 336
WHENEVER SUCCESSFUL clause
of AUDIT (Schema Objects) command, 4 - 179
WHERE clause
of SELECT command, 4 - 492
of UPDATE command, 4 - 545
wildcard characters
in pattern matching, 3-9
pattern matching, 3-8
WITH CONTEXT option
of CREATE FUNCTION command, 4 - 235
of CREATE PROCEDURE command, 4 - 262
WITH GRANT OPTION, 4 - 446
WITH_clause
of SELECT command, 4 - 491
of subqueries, 4 - 531
WNDS parameter
of ALTER TYPE command, 4 - 147
of CREATE TYPE command, 4 - 351
WNPS parameter
of ALTER TYPE command, 4 - 147
of CREATE TYPE command, 4 - 351
WORK option
of COMMIT command, 4 - 185
of ROLLBACK command, 4 - 484

Y

year
storing, 2-14
YEAR date format element, 3-72