Contents

Preface

Audience xxxvi
Documentation Accessibility xxxvi
Related Documents xxxvii
Conventions xxxvii
Syntax Descriptions xxxviii

1 Changes in This Release for Oracle Database PL/SQL Language Reference

1.1 New Features in Release 20c for Oracle Database PL/SQL Language Reference 1-1
   1.1.1 PL/SQL Extended Iterators 1-1
   1.1.2 PL/SQL Qualified Expressions Enhancements 1-2
   1.1.3 SQL Macros 1-2
   1.1.4 New JSON Data Type 1-3
   1.1.5 New Pragma SUPPRESSES_WARNING_6009 1-3
   1.1.6 PL/SQL Type Attributes in Non Persistable User Defined Types 1-4
1.2 Deprecated Features 1-4
1.3 Desupported Features 1-4

2 Overview of PL/SQL

2.1 Advantages of PL/SQL 2-1
   2.1.1 Tight Integration with SQL 2-1
   2.1.2 High Performance 2-2
   2.1.3 High Productivity 2-2
   2.1.4 Portability 2-2
   2.1.5 Scalability 2-3
   2.1.6 Manageability 2-3
   2.1.7 Support for Object-Oriented Programming 2-3
2.2 Main Features of PL/SQL 2-3
   2.2.1 Error Handling 2-4
3.3.4 Initial Values of Variables and Constants 3-17
3.3.5 Declaring Items using the %TYPE Attribute 3-18
3.4 References to Identifiers 3-20
3.5 Scope and Visibility of Identifiers 3-21
3.6 Assigning Values to Variables 3-25
  3.6.1 Assigning Values to Variables with the Assignment Statement 3-25
  3.6.2 Assigning Values to Variables with the SELECT INTO Statement 3-26
  3.6.3 Assigning Values to Variables as Parameters of a Subprogram 3-27
  3.6.4 Assigning Values to BOOLEAN Variables 3-28
3.7 Expressions 3-28
  3.7.1 Concatenation Operator 3-29
  3.7.2 Operator Precedence 3-29
  3.7.3 Logical Operators 3-32
  3.7.4 Short-Circuit Evaluation 3-37
  3.7.5 Comparison Operators 3-37
    3.7.5.1 IS [NOT] NULL Operator 3-38
    3.7.5.2 Relational Operators 3-38
    3.7.5.3 LIKE Operator 3-40
    3.7.5.4 BETWEEN Operator 3-42
    3.7.5.5 IN Operator 3-42
  3.7.6 BOOLEAN Expressions 3-43
  3.7.7 CASE Expressions 3-44
    3.7.7.1 Simple CASE Expression 3-44
    3.7.7.2 Searched CASE Expression 3-46
  3.7.8 SQL Functions in PL/SQL Expressions 3-47
  3.7.9 Static Expressions 3-48
    3.7.9.1 PLS_INTEGER Static Expressions 3-51
    3.7.9.2 BOOLEAN Static Expressions 3-52
    3.7.9.3 VARCHAR2 Static Expressions 3-53
    3.7.9.4 Static Constants 3-53
3.8 Error-Reporting Functions 3-54
3.9 Conditional Compilation 3-55
  3.9.1 How Conditional Compilation Works 3-55
    3.9.1.1 Preprocessor Control Tokens 3-56
    3.9.1.2 Selection Directives 3-56
    3.9.1.3 Error Directives 3-57
    3.9.1.4 Inquiry Directives 3-57
    3.9.1.5 DBMS_DB_VERSION Package 3-61
  3.9.2 Conditional Compilation Examples 3-61
  3.9.3 Retrieving and Printing Post-Processed Source Text 3-63
4 PL/SQL Data Types

4.1 SQL Data Types
4.1.1 Different Maximum Sizes
4.1.2 Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE
4.1.3 Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE
4.1.4 CHAR and VARCHAR2 Variables
  4.1.4.1 Assigning or Inserting Too-Long Values
  4.1.4.2 Declaring Variables for Multibyte Characters
  4.1.4.3 Differences Between CHAR and VARCHAR2 Data Types
4.1.5 LONG and LONG RAW Variables
4.1.6 ROWID and UROWID Variables
4.2 BOOLEAN Data Type
4.3 PLS_INTEGER and BINARY_INTEGER Data Types
  4.3.1 Preventing PLS_INTEGER Overflow
  4.3.2 Predefined PLS_INTEGER Subtypes
  4.3.3 SIMPLE_INTEGER Subtype of PLS_INTEGER
    4.3.3.1 SIMPLE_INTEGER Overflow Semantics
    4.3.3.2 Expressions with Both SIMPLE_INTEGER and Other Operands
    4.3.3.3 Integer Literals in SIMPLE_INTEGER Range
4.4 User-Defined PL/SQL Subtypes
  4.4.1 Unconstrained Subtypes
  4.4.2 Constrained Subtypes
  4.4.3 Subtypes with Base Types in Same Data Type Family

5 PL/SQL Control Statements

5.1 Conditional Selection Statements
  5.1.1 IF THEN Statement
  5.1.2 IF THEN ELSE Statement
  5.1.3 IF THEN ELSIF Statement
  5.1.4 Simple CASE Statement
  5.1.5 Searched CASE Statement
5.2 LOOP Statements
  5.2.1 Basic LOOP Statement
  5.2.2 FOR LOOP Statement Overview
    5.2.2.1 FOR LOOP Iterand
    5.2.2.2 Iterand Mutability
6.10.3   EXTEND Collection Method 6-37
6.10.4   EXISTS Collection Method 6-39
6.10.5   FIRST and LAST Collection Methods 6-39
   6.10.5.1   FIRST and LAST Methods for Associative Array 6-40
   6.10.5.2   FIRST and LAST Methods for Varray 6-41
   6.10.5.3   FIRST and LAST Methods for Nested Table 6-42
6.10.6   COUNT Collection Method 6-44
   6.10.6.1   COUNT Method for Varray 6-44
   6.10.6.2   COUNT Method for Nested Table 6-44
6.10.7   LIMIT Collection Method 6-45
6.10.8   PRIOR and NEXT Collection Methods 6-46
6.11 Collection Types Defined in Package Specifications 6-49
6.12 Record Variables 6-50
   6.12.1   Initial Values of Record Variables 6-51
   6.12.2   Declaring Record Constants 6-51
6.12.3 RECORD Types 6-52
6.12.4 Declaring Items using the %ROWTYPE Attribute 6-55
   6.12.4.1 Declaring a Record Variable that Always Represents Full Row 6-56
   6.12.4.2 Declaring a Record Variable that Can Represent Partial Row 6-57
   6.12.4.3 %ROWTYPE Attribute and Virtual Columns 6-59
   6.12.4.4 %ROWTYPE Attribute and Invisible Columns 6-60
6.13 Assigning Values to Record Variables 6-61
   6.13.1 Assigning One Record Variable to Another 6-62
6.13.2 Assigning Full or Partial Rows to Record Variables 6-64
   6.13.2.1 Using SELECT INTO to Assign a Row to a Record Variable 6-64
   6.13.2.2 Using FETCH to Assign a Row to a Record Variable 6-65
   6.13.2.3 Using SQL Statements to Return Rows in PL/SQL Record Variables 6-66
6.13.3 Assigning NULL to a Record Variable 6-67
6.14 Record Comparisons 6-67
6.15 Inserting Records into Tables 6-68
6.16 Updating Rows with Records 6-69
6.17 Restrictions on Record Inserts and Updates 6-70

7 PL/SQL Static SQL

7.1 Description of Static SQL 7-1
   7.1.1 Statements 7-1
   7.1.2 Pseudocolumns 7-3
      7.1.2.1 CURRVAL and NEXTVAL in PL/SQL 7-3
7.2 Cursors Overview 7-5
   7.2.1 Implicit Cursors 7-6
7.2.1.1 SQL%ISOPEN Attribute: Is the Cursor Open? 7-7
7.2.1.2 SQL%FOUND Attribute: Were Any Rows Affected? 7-7
7.2.1.3 SQL%NOTFOUND Attribute: Were No Rows Affected? 7-8
7.2.1.4 SQL%ROWCOUNT Attribute: How Many Rows Were Affected? 7-8
7.2.2 Explicit Cursors 7-9
  7.2.2.1 Declaring and Defining Explicit Cursors 7-10
  7.2.2.2 Opening and Closing Explicit Cursors 7-10
  7.2.2.3 Fetching Data with Explicit Cursors 7-11
  7.2.2.4 Variables in Explicit Cursor Queries 7-13
  7.2.2.5 When Explicit Cursor Queries Need Column Aliases 7-15
  7.2.2.6 Explicit Cursors that Accept Parameters 7-16
  7.2.2.7 Explicit Cursor Attributes 7-20
7.3 Processing Query Result Sets 7-24
  7.3.1 Processing Query Result Sets With SELECT INTO Statements 7-25
    7.3.1.1 Handling Single-Row Result Sets 7-25
    7.3.1.2 Handling Large Multiple-Row Result Sets 7-26
  7.3.2 Processing Query Result Sets With Cursor FOR LOOP Statements 7-26
  7.3.3 Processing Query Result Sets With Explicit Cursors, OPEN, FETCH, and CLOSE 7-29
  7.3.4 Processing Query Result Sets with Subqueries 7-29
7.4 Cursor Variables 7-30
  7.4.1 Creating Cursor Variables 7-31
  7.4.2 Opening and Closing Cursor Variables 7-33
  7.4.3 Fetching Data with Cursor Variables 7-33
  7.4.4 Assigning Values to Cursor Variables 7-35
  7.4.5 Variables in Cursor Variable Queries 7-36
  7.4.6 Querying a Collection 7-38
  7.4.7 Cursor Variable Attributes 7-39
  7.4.8 Cursor Variables as Subprogram Parameters 7-39
  7.4.9 Cursor Variables as Host Variables 7-41
7.5 CURSOR Expressions 7-43
7.6 Transaction Processing and Control 7-44
  7.6.1 COMMIT Statement 7-45
  7.6.2 ROLLBACK Statement 7-47
  7.6.3 SAVEPOINT Statement 7-48
  7.6.4 Implicit Rollbacks 7-50
  7.6.5 SET TRANSACTION Statement 7-50
  7.6.6 Overriding Default Locking 7-51
    7.6.6.1 LOCK TABLE Statement 7-52
    7.6.6.2 SELECT FOR UPDATE and FOR UPDATE Cursors 7-52
    7.6.6.3 Simulating CURRENT OF Clause with ROWID Pseudocolumn 7-53
10 PL/SQL Triggers

10.1 Overview of Triggers
10.2 Reasons to Use Triggers
10.3 DML Triggers
   10.3.1 Conditional Predicates for Detecting Triggering DML Statement
   10.3.2 INSTEAD OF DML Triggers
   10.3.3 Compound DML Triggers
      10.3.3.1 Compound DML Trigger Structure
      10.3.3.2 Compound DML Trigger Restrictions
      10.3.3.3 Performance Benefit of Compound DML Triggers
      10.3.3.4 Using Compound DML Triggers with Bulk Insertion
      10.3.3.5 Using Compound DML Triggers to Avoid Mutating-Table Error
   10.3.4 Triggers for Ensuring Referential Integrity
      10.3.4.1 Foreign Key Trigger for Child Table
      10.3.4.2 UPDATE and DELETE RESTRICT Trigger for Parent Table
      10.3.4.3 UPDATE and DELETE SET NULL Trigger for Parent Table
      10.3.4.4 DELETE CASCADE Trigger for Parent Table
      10.3.4.5 UPDATE CASCADE Trigger for Parent Table
      10.3.4.6 Triggers for Complex Constraint Checking
      10.3.4.7 Triggers for Complex Security Authorizations
      10.3.4.8 Triggers for Transparent Event Logging
      10.3.4.9 Triggers for Deriving Column Values
      10.3.4.10 Triggers for Building Complex Updatable Views
      10.3.4.11 Triggers for Fine-Grained Access Control
   10.4 Correlation Names and Pseudorecords
      10.4.1 OBJECT_VALUE Pseudocolumn
   10.5 System Triggers
      10.5.1 SCHEMA Triggers
      10.5.2 DATABASE Triggers
      10.5.3 INSTEAD OF CREATE Triggers
10.6 Subprograms Invoked by Triggers 10-36
10.7 Trigger Compilation, Invalidation, and Recompilation 10-37
10.8 Exception Handling in Triggers 10-37
10.9 Trigger Design Guidelines 10-39
10.10 Trigger Restrictions 10-41
  10.10.1 Trigger Size Restriction 10-41
  10.10.2 Trigger LONG and LONG RAW Data Type Restrictions 10-41
  10.10.3 Mutating-Table Restriction 10-42
10.11 Order in Which Triggers Fire 10-45
10.12 Trigger Enabling and Disabling 10-46
10.13 Trigger Changing and Debugging 10-47
10.14 Triggers and Oracle Database Data Transfer Utilities 10-47
10.15 Triggers for Publishing Events 10-49
  10.15.1 Event Attribute Functions 10-50
  10.15.2 Event Attribute Functions for Database Event Triggers 10-54
  10.15.3 Event Attribute Functions for Client Event Triggers 10-55
10.16 Views for Information About Triggers 10-60

11 PL/SQL Packages

11.1 What is a Package? 11-1
11.2 Reasons to Use Packages 11-2
11.3 Package Specification 11-3
  11.3.1 Appropriate Public Items 11-4
  11.3.2 Creating Package Specifications 11-5
11.4 Package Body 11-6
11.5 Package Instantiation and Initialization 11-7
11.6 Package State 11-7
11.7 SERIALLY_REUSABLE Packages 11-8
  11.7.1 Creating SERIALLY_REUSABLE Packages 11-9
  11.7.2 SERIALLY_REUSABLE Package Work Unit 11-10
  11.7.3 Explicit Cursors in SERIALLY_REUSABLE Packages 11-11
11.8 Package Writing Guidelines 11-12
11.9 Package Example 11-15
11.10 How STANDARD Package Defines the PL/SQL Environment 11-18

12 PL/SQL Error Handling

12.1 Compile-Time Warnings 12-2
  12.1.1 DBMS_WARNING Package 12-4
12.2 Overview of Exception Handling 12-5
12.2.1 Exception Categories 12-6
12.2.2 Advantages of Exception Handlers 12-7
12.2.3 Guidelines for Avoiding and Handling Exceptions 12-9
12.3 Internally Defined Exceptions 12-10
12.4 Predefined Exceptions 12-11
12.5 User-Defined Exceptions 12-14
12.6 Redeclared Predefined Exceptions 12-14
12.7 Raising Exceptions Explicitly 12-16
   12.7.1 RAISE Statement 12-16
      12.7.1.1 Raising User-Defined Exception with RAISE Statement 12-16
      12.7.1.2 Raising Internally Defined Exception with RAISE Statement 12-17
      12.7.1.3 Reraising Current Exception with RAISE Statement 12-17
   12.7.2 RAISE_APPLICATION_ERROR Procedure 12-18
12.8 Exception Propagation 12-20
   12.8.1 Propagation of Exceptions Raised in Declarations 12-23
   12.8.2 Propagation of Exceptions Raised in Exception Handlers 12-24
12.9 Unhandled Exceptions 12-27
12.10 Retrieving Error Code and Error Message 12-27
12.11 Continuing Execution After Handling Exceptions 12-29
12.12 Retrying Transactions After Handling Exceptions 12-30
12.13 Handling Errors in Distributed Queries 12-31

13 PL/SQL Optimization and Tuning

13.1 PL/SQL Optimizer 13-1
   13.1.1 Subprogram Inlining 13-2
13.2 Candidates for Tuning 13-4
13.3 Minimizing CPU Overhead 13-5
   13.3.1 Tune SQL Statements 13-5
   13.3.2 Tune Function Invocations in Queries 13-6
   13.3.3 Tune Subprogram Invocations 13-7
   13.3.4 Tune Loops 13-9
   13.3.5 Tune Computation-Intensive PL/SQL Code 13-9
      13.3.5.1 Use Data Types that Use Hardware Arithmetic 13-9
      13.3.5.2 Avoid Constrained Subtypes in Performance-Critical Code 13-10
      13.3.5.3 Minimize Implicit Data Type Conversion 13-10
   13.3.6 Use SQL Character Functions 13-11
   13.3.7 Put Least Expensive Conditional Tests First 13-11
13.4 Bulk SQL and Bulk Binding 13-12
   13.4.1 FORALL Statement 13-13
      13.4.1.1 Using FORALL Statements for Sparse Collections 13-16
PL/SQL Language Elements

14.43  Iterator 14-3
14.52  Qualified Expression 14-10
14.64  SQL_MACRO Clause 14-14
14.1  ACCESSIBLE BY Clause 14-22
14.2  AGGREGATE Clause 14-27
14.3  Assignment Statement 14-28
14.4  AUTONOMOUS_TRANSACTION Pragma 14-31
14.5  Basic LOOP Statement 14-32
14.6  Block 14-35
14.7  Call Specification 14-44
14.8  CASE Statement 14-47
14.9  CLOSE Statement 14-50
14.10  Collection Method Invocation 14-51
14.11  Collection Variable Declaration 14-54
14.12  Comment 14-59
14.13  COMPILE Clause 14-60
14.14  Constant Declaration 14-63
14.15  CONTINUE Statement 14-64
14.16  COVERAGE Pragma 14-66
14.17  Cursor FOR LOOP Statement 14-70
14.18  Cursor Variable Declaration 14-72
14.19  Datatype Attribute 14-74
14.20  DEFAULT COLLATION Clause 14-75
14.21  DELETE Statement Extension 14-78
14.22  DEPRECATE Pragma 14-78
14.23  DETERMINISTIC Clause 14-88
14.24  Element Specification 14-90
14.25  EXCEPTION_INIT Pragma 14-95
14.26  Exception Declaration 14-97
14.27  Exception Handler 14-98
14.28  EXECUTE IMMEDIATE Statement 14-100
14.29  EXIT Statement 14-103
14.30  Explicit Cursor Declaration and Definition 14-105
14.31  Expression 14-109
14.32  FETCH Statement 14-119
14.33  FOR LOOP Statement 14-121
14.34  FORALL Statement 14-124
14.35  Formal Parameter Declaration 14-127
14.36  Function Declaration and Definition 14-130
SQL Statements for Stored PL/SQL Units

15.1 ALTER FUNCTION Statement
15.2 ALTER LIBRARY Statement
15.3 ALTER PACKAGE Statement
15.4 ALTER PROCEDURE Statement
15.5 ALTER TRIGGER Statement
15.6 ALTER TYPE Statement
| 15.7 CREATE FUNCTION Statement | 15-25 |
| 15.8 CREATE LIBRARY Statement | 15-31 |
| 15.9 CREATE PACKAGE Statement | 15-35 |
| 15.10 CREATE PACKAGE BODY Statement | 15-38 |
| 15.11 CREATE PROCEDURE Statement | 15-42 |
| 15.12 CREATE TRIGGER Statement | 15-45 |
| 15.13 CREATE TYPE Statement | 15-66 |
| 15.14 CREATE TYPE BODY Statement | 15-77 |
| 15.15 DROP FUNCTION Statement | 15-81 |
| 15.16 DROP LIBRARY Statement | 15-83 |
| 15.17 DROP PACKAGE Statement | 15-84 |
| 15.18 DROP PROCEDURE Statement | 15-85 |
| 15.19 DROP TRIGGER Statement | 15-86 |
| 15.20 DROP TYPE Statement | 15-87 |
| 15.21 DROP TYPE BODY Statement | 15-90 |

### A
**PL/SQL Source Text Wrapping**

- A.1 PL/SQL Source Text Wrapping Limitations A-2
- A.4 Wrapping PL/SQL Source Text with DBMS_DDL Subprograms A-8

### B
**PL/SQL Name Resolution**

- B.1 Qualified Names and Dot Notation B-1
- B.2 Column Name Precedence B-3
- B.3 Differences Between PL/SQL and SQL Name Resolution Rules B-5
- B.4 Resolution of Names in Static SQL Statements B-6
- B.5 What is Capture? B-7
  - B.5.1 Outer Capture B-7
  - B.5.2 Same-Scope Capture B-7
  - B.5.3 Inner Capture B-7
- B.6 Avoiding Inner Capture in SELECT and DML Statements B-8
  - B.6.1 Qualifying References to Attributes and Methods B-9
  - B.6.2 Qualifying References to Row Expressions B-10

### C
**PL/SQL Program Limits**
D  PL/SQL Reserved Words and Keywords

E  PL/SQL Predefined Data Types

Index
List of Examples

2-1 PL/SQL Block Structure 2-5
2-2 Processing Query Result Rows One at a Time 2-9
3-1 Valid Case-Insensitive Reference to Quoted User-Defined Identifier 3-8
3-2 Invalid Case-Insensitive Reference to Quoted User-Defined Identifier 3-9
3-3 Reserved Word as Quoted User-Defined Identifier 3-9
3-4 Neglecting Double Quotation Marks 3-9
3-5 Neglecting Case-Sensitivity 3-10
3-6 Single-Line Comments 3-13
3-7 Multiline Comments 3-14
3-8 Whitespace Characters Improving Source Text Readability 3-15
3-9 Variable Declaration with NOT NULL Constraint 3-16
3-10 Variables Initialized to NULL Values 3-16
3-11 Scalar Variable Declarations 3-16
3-12 Constant Declarations 3-17
3-13 Variable and Constant Declarations with Initial Values 3-18
3-14 Variable Initialized to NULL by Default 3-18
3-15 Declaring Variable of Same Type as Column 3-19
3-16 Declaring Variable of Same Type as Another Variable 3-19
3-17 Scope and Visibility of Identifiers 3-21
3-18 Qualifying Redeclared Global Identifier with Block Label 3-22
3-19 Qualifying Identifier with Subprogram Name 3-22
3-20 Duplicate Identifiers in Same Scope 3-23
3-21 Declaring Same Identifier in Different Units 3-23
3-22 Label and Subprogram with Same Name in Same Scope 3-24
3-23 Block with Multiple and Duplicate Labels 3-24
3-24 Assigning Values to Variables with Assignment Statement 3-26
3-25 Assigning Value to Variable with SELECT INTO Statement 3-26
3-26 Assigning Value to Variable as IN OUT Subprogram Parameter 3-27
3-27 Assigning Value to BOOLEAN Variable 3-28
3-28 Concatenation Operator 3-29
3-29 Concatenation Operator with NULL Operands 3-29
3-30 Controlling Evaluation Order with Parentheses 3-30
3-31 Expression with Nested Parentheses 3-30
3-32 Improving Readability with Parentheses 3-30
3-33 Operator Precedence 3-31
3-34 Procedure Prints BOOLEAN Variable
3-35 AND Operator
3-36 OR Operator
3-37 NOT Operator
3-38 NULL Value in Unequal Comparison
3-39 NULL Value in Equal Comparison
3-40 NOT NULL Equals NULL
3-41 Changing Evaluation Order of Logical Operators
3-42 Short-Circuit Evaluation
3-43 Relational Operators in Expressions
3-44 LIKE Operator in Expression
3-45 Escape Character in Pattern
3-46 BETWEEN Operator in Expressions
3-47 IN Operator in Expressions
3-48 IN Operator with Sets with NULL Values
3-49 Equivalent BOOLEAN Expressions
3-50 Simple CASE Expression
3-51 Simple CASE Expression with WHEN NULL
3-52 Searched CASE Expression
3-53 Searched CASE Expression with WHEN ... IS NULL
3-54 Static Constants
3-55 Predefined Inquiry Directives
3-56 Displaying Values of PL/SQL Compilation Parameters
3-57 PLSQL_CCFLAGS Assigns Value to Itself
3-58 Code for Checking Database Version
3-59 Compiling Different Code for Different Database Versions
3-60 Displaying Post-Processed Source Text
3-61 Using Conditional Compilation Directive in the Definition of a Package Specification
3-62 Using Conditional Compilation Directive in the Formal Parameter List of a Subprogram
4-1 CHAR and VARCHAR2 Blank-Padding Difference
4-2 Printing BOOLEAN Values
4-3 SQL Statement Invokes PL/SQL Function with BOOLEAN Parameter
4-4 PLS_INTEGER Calculation Raises Overflow Exception
4-5 Preventing Overflow
4-6 Violating Constraint of SIMPLE_INTEGER Subtype
4-7 User-Defined Unconstrained Subtypes Show Intended Use
4-8 User-Defined Constrained Subtype Detects Out-of-Range Values
6-42 Identically Defined Package and Local RECORD Types
6-43 %ROWTYPE Variable Represents Full Database Table Row
6-44 %ROWTYPE Variable Does Not Inherit Initial Values or Constraints
6-45 %ROWTYPE Variable Represents Partial Database Table Row
6-46 %ROWTYPE Variable Represents Join Row
6-47 Inserting %ROWTYPE Record into Table (Wrong)
6-48 Inserting %ROWTYPE Record into Table (Right)
6-49 %ROWTYPE Affected by Making Invisible Column Visible
6-50 Assigning Record to Another Record of Same RECORD Type
6-51 Assigning %ROWTYPE Record to RECORD Type Record
6-52 Assigning Nested Record to Another Record of Same RECORD Type
6-53 SELECT INTO Assigns Values to Record Variable
6-54 FETCH Assigns Values to Record that Function Returns
6-55 UPDATE Statement Assigns Values to Record Variable
6-56 Assigning NULL to Record Variable
6-57 Initializing Table by Inserting Record of Default Values
6-58 Updating Rows with Record
7-1 Static SQL Statements
7-2 CURRVAL and NEXTVAL Pseudocolumns
7-3 SQL%FOUND Implicit Cursor Attribute
7-4 SQL%ROWCOUNT Implicit Cursor Attribute
7-5 Explicit Cursor Declaration and Definition
7-6 FETCH Statements Inside LOOP Statements
7-7 Fetching Same Explicit Cursor into Different Variables
7-8 Variable in Explicit Cursor Query—No Result Set Change
7-9 Variable in Explicit Cursor Query—Result Set Change
7-10 Explicit Cursor with Virtual Column that Needs Alias
7-11 Explicit Cursor that Accepts Parameters
7-12 Cursor Parameters with Default Values
7-13 Adding Formal Parameter to Existing Cursor
7-14 %ISOPEN Explicit Cursor Attribute
7-15 %FOUND Explicit Cursor Attribute
7-16 %NOTFOUND Explicit Cursor Attribute
7-17 %ROWCOUNT Explicit Cursor Attribute
7-18 Implicit Cursor FOR LOOP Statement
7-19 Explicit Cursor FOR LOOP Statement
7-20 Passing Parameters to Explicit Cursor FOR LOOP Statement
<table>
<thead>
<tr>
<th>Page</th>
<th>Section Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-21</td>
<td>Cursor FOR Loop References Virtual Columns</td>
</tr>
<tr>
<td>7-22</td>
<td>Subquery in FROM Clause of Parent Query</td>
</tr>
<tr>
<td>7-23</td>
<td>Correlated Subquery</td>
</tr>
<tr>
<td>7-24</td>
<td>Cursor Variable Declarations</td>
</tr>
<tr>
<td>7-25</td>
<td>Cursor Variable with User-Defined Return Type</td>
</tr>
<tr>
<td>7-26</td>
<td>Fetching Data with Cursor Variables</td>
</tr>
<tr>
<td>7-27</td>
<td>Fetching from Cursor Variable into Collections</td>
</tr>
<tr>
<td>7-28</td>
<td>Variable in Cursor Variable Query—No Result Set Change</td>
</tr>
<tr>
<td>7-29</td>
<td>Variable in Cursor Variable Query—Result Set Change</td>
</tr>
<tr>
<td>7-30</td>
<td>Querying a Collection with Static SQL</td>
</tr>
<tr>
<td>7-31</td>
<td>Procedure to Open Cursor Variable for One Query</td>
</tr>
<tr>
<td>7-32</td>
<td>Opening Cursor Variable for Chosen Query (Same Return Type)</td>
</tr>
<tr>
<td>7-33</td>
<td>Opening Cursor Variable for Chosen Query (Different Return Types)</td>
</tr>
<tr>
<td>7-34</td>
<td>Cursor Variable as Host Variable in Pro*C Client Program</td>
</tr>
<tr>
<td>7-35</td>
<td>CURSOR Expression</td>
</tr>
<tr>
<td>7-36</td>
<td>COMMIT Statement with COMMENT and WRITE Clauses</td>
</tr>
<tr>
<td>7-37</td>
<td>ROLLBACK Statement</td>
</tr>
<tr>
<td>7-38</td>
<td>SAVEPOINT and ROLLBACK Statements</td>
</tr>
<tr>
<td>7-39</td>
<td>Reusing SAVEPOINT with ROLLBACK</td>
</tr>
<tr>
<td>7-40</td>
<td>SET TRANSACTION Statement in Read-Only Transaction</td>
</tr>
<tr>
<td>7-41</td>
<td>FETCH with FOR UPDATE Cursor After COMMIT Statement</td>
</tr>
<tr>
<td>7-42</td>
<td>Simulating CURRENT OF Clause with ROWID Pseudocolumn</td>
</tr>
<tr>
<td>7-43</td>
<td>Declaring Autonomous Function in Package</td>
</tr>
<tr>
<td>7-44</td>
<td>Declaring Autonomous Standalone Procedure</td>
</tr>
<tr>
<td>7-45</td>
<td>Declaring Autonomous PL/SQL Block</td>
</tr>
<tr>
<td>7-46</td>
<td>Autonomous Trigger Logs INSERT Statements</td>
</tr>
<tr>
<td>7-47</td>
<td>Autonomous Trigger Uses Native Dynamic SQL for DDL</td>
</tr>
<tr>
<td>7-48</td>
<td>Invoking Autonomous Function</td>
</tr>
<tr>
<td>8-1</td>
<td>Invoking Subprogram from Dynamic PL/SQL Block</td>
</tr>
<tr>
<td>8-2</td>
<td>Dynamically Invoking Subprogram with BOOLEAN Formal Parameter</td>
</tr>
<tr>
<td>8-3</td>
<td>Dynamically Invoking Subprogram with RECORD Formal Parameter</td>
</tr>
<tr>
<td>8-4</td>
<td>Dynamically Invoking Subprogram with Assoc. Array Formal Parameter</td>
</tr>
<tr>
<td>8-5</td>
<td>Dynamically Invoking Subprogram with Nested Table Formal Parameter</td>
</tr>
<tr>
<td>8-6</td>
<td>Dynamically Invoking Subprogram with Varray Formal Parameter</td>
</tr>
<tr>
<td>8-7</td>
<td>Uninitialized Variable Represents NULL in USING Clause</td>
</tr>
<tr>
<td>8-8</td>
<td>Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements</td>
</tr>
<tr>
<td>8-9</td>
<td>Querying a Collection with Native Dynamic SQL</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>9-26</td>
<td>Overloaded Subprogram</td>
</tr>
<tr>
<td>9-27</td>
<td>Overload Error Causes Compile-Time Error</td>
</tr>
<tr>
<td>9-28</td>
<td>Overload Error Compiles Successfully</td>
</tr>
<tr>
<td>9-29</td>
<td>Invoking Subprogram in Causes Compile-Time Error</td>
</tr>
<tr>
<td>9-30</td>
<td>Correcting Overload Error in</td>
</tr>
<tr>
<td>9-31</td>
<td>Invoking Subprogram in</td>
</tr>
<tr>
<td>9-32</td>
<td>Package Specification Without Overload Errors</td>
</tr>
<tr>
<td>9-33</td>
<td>Improper Invocation of Properly Overloaded Subprogram</td>
</tr>
<tr>
<td>9-34</td>
<td>Implicit Conversion of Parameters Causes Overload Error</td>
</tr>
<tr>
<td>9-35</td>
<td>Recursive Function Returns n Factorial (n!)</td>
</tr>
<tr>
<td>9-36</td>
<td>Recursive Function Returns nth Fibonacci Number</td>
</tr>
<tr>
<td>9-37</td>
<td>Declaring and Defining Result-Cached Function</td>
</tr>
<tr>
<td>9-38</td>
<td>Result-Cached Function Returns Configuration Parameter Setting</td>
</tr>
<tr>
<td>9-39</td>
<td>Result-Cached Function Handles Session-Specific Settings</td>
</tr>
<tr>
<td>9-40</td>
<td>Result-Cached Function Handles Session-Specific Application Context</td>
</tr>
<tr>
<td>9-41</td>
<td>Caching One Name at a Time (Finer Granularity)</td>
</tr>
<tr>
<td>9-42</td>
<td>Caching Translated Names One Language at a Time (Coarser Granularity)</td>
</tr>
<tr>
<td>9-43</td>
<td>Database Link in a DR Unit</td>
</tr>
<tr>
<td>9-44</td>
<td>PL/SQL Anonymous Block Invokes External Procedure</td>
</tr>
<tr>
<td>9-45</td>
<td>PL/SQL Standalone Procedure Invokes External Procedure</td>
</tr>
<tr>
<td>10-1</td>
<td>Trigger Uses Conditional Predicates to Detect Triggering Statement</td>
</tr>
<tr>
<td>10-2</td>
<td>INSTEAD OF Trigger</td>
</tr>
<tr>
<td>10-3</td>
<td>INSTEAD OF Trigger on Nested Table Column of View</td>
</tr>
<tr>
<td>10-4</td>
<td>Compound Trigger Logs Changes to One Table in Another Table</td>
</tr>
<tr>
<td>10-5</td>
<td>Compound Trigger Avoids Mutating-Table Error</td>
</tr>
<tr>
<td>10-6</td>
<td>Foreign Key Trigger for Child Table</td>
</tr>
<tr>
<td>10-7</td>
<td>UPDATE and DELETE RESTRICT Trigger for Parent Table</td>
</tr>
<tr>
<td>10-8</td>
<td>UPDATE and DELETE SET NULL Trigger for Parent Table</td>
</tr>
<tr>
<td>10-9</td>
<td>DELETE CASCADE Trigger for Parent Table</td>
</tr>
<tr>
<td>10-10</td>
<td>UPDATE CASCADE Trigger for Parent Table</td>
</tr>
<tr>
<td>10-11</td>
<td>Trigger Checks Complex Constraints</td>
</tr>
<tr>
<td>10-12</td>
<td>Trigger Enforces Security Authorizations</td>
</tr>
<tr>
<td>10-13</td>
<td>Trigger Derives New Column Values</td>
</tr>
<tr>
<td>10-14</td>
<td>Trigger Logs Changes to EMPLOYEES.SALARY</td>
</tr>
<tr>
<td>10-15</td>
<td>Conditional Trigger Prints Salary Change Information</td>
</tr>
<tr>
<td>10-16</td>
<td>Trigger Modifies CLOB Columns</td>
</tr>
<tr>
<td>10-17</td>
<td>Trigger with REFERENCING Clause</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>10-18</td>
<td>Trigger References OBJECT_VALUE Pseudocolumn</td>
</tr>
<tr>
<td>10-19</td>
<td>BEFORE Statement Trigger on Sample Schema HR</td>
</tr>
<tr>
<td>10-20</td>
<td>AFTER Statement Trigger on Database</td>
</tr>
<tr>
<td>10-21</td>
<td>Trigger Monitors Logons</td>
</tr>
<tr>
<td>10-22</td>
<td>INSTEAD OF CREATE Trigger on Schema</td>
</tr>
<tr>
<td>10-23</td>
<td>Trigger Invokes Java Subprogram</td>
</tr>
<tr>
<td>10-24</td>
<td>Trigger Cannot Handle Exception if Remote Database is Unavailable</td>
</tr>
<tr>
<td>10-25</td>
<td>Workaround for</td>
</tr>
<tr>
<td>10-26</td>
<td>Trigger Causes Mutating-Table Error</td>
</tr>
<tr>
<td>10-27</td>
<td>Update Cascade</td>
</tr>
<tr>
<td>10-28</td>
<td>Viewing Information About Triggers</td>
</tr>
<tr>
<td>11-1</td>
<td>Simple Package Specification</td>
</tr>
<tr>
<td>11-2</td>
<td>Passing Associative Array to Standalone Subprogram</td>
</tr>
<tr>
<td>11-3</td>
<td>Matching Package Specification and Body</td>
</tr>
<tr>
<td>11-4</td>
<td>Creating SERIALLY_REUSABLE Packages</td>
</tr>
<tr>
<td>11-5</td>
<td>Effect of SERIALLY_REUSABLE Pragma</td>
</tr>
<tr>
<td>11-6</td>
<td>Cursor in SERIALLY_REUSABLE Package Open at Call Boundary</td>
</tr>
<tr>
<td>11-7</td>
<td>Separating Cursor Declaration and Definition in Package</td>
</tr>
<tr>
<td>11-8</td>
<td>ACCESSIBLE BY Clause</td>
</tr>
<tr>
<td>11-9</td>
<td>Creating emp_admin Package</td>
</tr>
<tr>
<td>12-1</td>
<td>Setting Value of PLSQL_WARNINGS Compilation Parameter</td>
</tr>
<tr>
<td>12-2</td>
<td>Displaying and Setting PLSQL_WARNINGS with DBMS_WARNING Subprograms</td>
</tr>
<tr>
<td>12-3</td>
<td>Single Exception Handler for Multiple Exceptions</td>
</tr>
<tr>
<td>12-4</td>
<td>Locator Variables for Statements that Share Exception Handler</td>
</tr>
<tr>
<td>12-5</td>
<td>Naming Internally Defined Exception</td>
</tr>
<tr>
<td>12-6</td>
<td>Anonymous Block Handles ZERO_DIVIDE</td>
</tr>
<tr>
<td>12-7</td>
<td>Anonymous Block Avoids ZERO_DIVIDE</td>
</tr>
<tr>
<td>12-8</td>
<td>Anonymous Block Handles ROWTYPE_MISMATCH</td>
</tr>
<tr>
<td>12-9</td>
<td>Redeclared Predefined Identifier</td>
</tr>
<tr>
<td>12-10</td>
<td>Declaring, Raising, and Handling User-Defined Exception</td>
</tr>
<tr>
<td>12-11</td>
<td>Explicitly Raising Predefined Exception</td>
</tr>
<tr>
<td>12-12</td>
<td>Reraising Exception</td>
</tr>
<tr>
<td>12-13</td>
<td>Raising User-Defined Exception with RAISE_APPLICATION_ERROR</td>
</tr>
<tr>
<td>12-14</td>
<td>Exception that Propagates Beyond Scope is Handled</td>
</tr>
<tr>
<td>12-15</td>
<td>Exception that Propagates Beyond Scope is Not Handled</td>
</tr>
<tr>
<td>12-16</td>
<td>Exception Raised in Declaration is Not Handled</td>
</tr>
<tr>
<td>12-17</td>
<td>Exception Raised in Declaration is Handled by Enclosing Block</td>
</tr>
</tbody>
</table>

ORACLE
14-28  Deprecation Only Applies to Top Level Subprogram 14-86
14-29  Misplaced DEPRECATE Pragma 14-87
14-30  Mismatch of the Element Name and the DEPRECATE Pragma Argument 14-87
14-31  Basic LOOP Statement with EXIT Statement 14-104
14-32  Basic LOOP Statement with EXIT WHEN Statement 14-104
14-33  EXIT WHEN Statement in FOR LOOP Statement 14-123
14-34  EXIT WHEN Statement in Inner FOR LOOP Statement 14-123
14-35  CONTINUE WHEN Statement in Inner FOR LOOP Statement 14-124
14-36  GOTO Statement 14-133
14-37  Incorrect Label Placement 14-133
14-38  GOTO Statement Goes to Labeled NULL Statement 14-134
14-39  GOTO Statement Transfers Control to Enclosing Block 14-134
14-40  GOTO Statement Cannot Transfer Control into IF Statement 14-135
14-41  Enabling the PLW-6009 Warning 14-185
14-42  SUPPRESSES_WARNING_6009 Pragma in a Procedure 14-185
14-43  SUPPRESSES_WARNING_6009 Pragma in a Function 14-185
14-44  SUPPRESSES_WARNING_6009 Pragma in an Overloaded Subprogram in a Package Specification 14-186
14-45  SUPPRESSES_WARNING_6009 Pragma in a Forward Declaration in a Package Body 14-187
14-46  SUPPRESSES_WARNING_6009 Pragma in Object Type Methods 14-188
14-47  WHILE LOOP Statements 14-193
15-1  Recompiling a Function 15-4
15-2  Recompiling a Library 15-5
15-3  Recompiling a Package 15-7
15-4  Recompiling a Procedure 15-9
15-5  Disabling Triggers 15-12
15-6  Enabling Triggers 15-12
15-7  Adding a Member Function 15-21
15-8  Adding a Collection Attribute 15-22
15-9  Increasing the Number of Elements of a Collection Type 15-22
15-10 Increasing the Length of a Collection Type 15-22
15-11 Recompiling a Type 15-22
15-12 Recompiling a Type Specification 15-22
15-13 Evolving and Resetting an ADT 15-23
15-14 Creating a Function 15-28
15-15 Creating Aggregate Functions 15-29
15-16 Package Procedure in a Function 15-30
List of Figures

2-1  PL/SQL Engine  2-11
6-1  Varray of Maximum Size 10 with 7 Elements  6-10
6-2  Array and Nested Table  6-15
7-1  Transaction Control Flow  7-55
9-1  How PL/SQL Compiler Resolves Invocations  9-29
12-1 Exception Does Not Propagate  12-20
12-2 Exception Propagates from Inner Block to Outer Block  12-21
12-3 PL/SQL Returns Unhandled Exception Error to Host Environment  12-21
# List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>PL/SQL I/O-Processing Packages</td>
<td>2-6</td>
</tr>
<tr>
<td>2-2</td>
<td>PL/SQL Compilation Parameters</td>
<td>2-12</td>
</tr>
<tr>
<td>3-1</td>
<td>Punctuation Characters in Every Database Character Set</td>
<td>3-2</td>
</tr>
<tr>
<td>3-2</td>
<td>PL/SQL Delimiters</td>
<td>3-5</td>
</tr>
<tr>
<td>3-3</td>
<td>Operator Precedence</td>
<td>3-29</td>
</tr>
<tr>
<td>3-4</td>
<td>Logical Truth Table</td>
<td>3-32</td>
</tr>
<tr>
<td>3-5</td>
<td>Relational Operators</td>
<td>3-38</td>
</tr>
<tr>
<td>3-6</td>
<td>Operators Allowed in Static Expressions</td>
<td>3-49</td>
</tr>
<tr>
<td>4-1</td>
<td>Data Types with Different Maximum Sizes in PL/SQL and SQL</td>
<td>4-2</td>
</tr>
<tr>
<td>4-2</td>
<td>Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants</td>
<td>4-3</td>
</tr>
<tr>
<td>4-3</td>
<td>Predefined Subtypes of PLS_INTEGER Data Type</td>
<td>4-12</td>
</tr>
<tr>
<td>6-1</td>
<td>PL/SQL Collection Types</td>
<td>6-2</td>
</tr>
<tr>
<td>6-2</td>
<td>Collection Methods</td>
<td>6-32</td>
</tr>
<tr>
<td>9-1</td>
<td>PL/SQL Subprogram Parameter Modes</td>
<td>9-15</td>
</tr>
<tr>
<td>9-2</td>
<td>PL/SQL Subprogram Parameter Modes Characteristics</td>
<td>9-15</td>
</tr>
<tr>
<td>9-3</td>
<td>PL/SQL Actual Parameter Notations</td>
<td>9-27</td>
</tr>
<tr>
<td>9-4</td>
<td>Finer and Coarser Caching Granularity</td>
<td>9-46</td>
</tr>
<tr>
<td>10-1</td>
<td>Conditional Predicates</td>
<td>10-5</td>
</tr>
<tr>
<td>10-2</td>
<td>Compound Trigger Timing-Point Sections</td>
<td>10-11</td>
</tr>
<tr>
<td>10-3</td>
<td>Constraints and Triggers for Ensuring Referential Integrity</td>
<td>10-16</td>
</tr>
<tr>
<td>10-4</td>
<td>OLD and NEW Pseudorecord Field Values</td>
<td>10-29</td>
</tr>
<tr>
<td>10-5</td>
<td>System-Defined Event Attributes</td>
<td>10-51</td>
</tr>
<tr>
<td>10-6</td>
<td>Database Event Triggers</td>
<td>10-55</td>
</tr>
<tr>
<td>10-7</td>
<td>Client Event Triggers</td>
<td>10-56</td>
</tr>
<tr>
<td>12-1</td>
<td>Compile-Time Warning Categories</td>
<td>12-2</td>
</tr>
<tr>
<td>12-2</td>
<td>Exception Categories</td>
<td>12-7</td>
</tr>
<tr>
<td>12-3</td>
<td>PL/SQL Predefined Exceptions</td>
<td>12-11</td>
</tr>
<tr>
<td>13-1</td>
<td>Profiling and Tracing Tools Summary</td>
<td>13-68</td>
</tr>
<tr>
<td>14-1</td>
<td>Iterand Implicit Type Defaults</td>
<td>14-6</td>
</tr>
<tr>
<td>14-2</td>
<td>Summary of Possible Sharing Attributes by Application Common Object Type</td>
<td>14-177</td>
</tr>
<tr>
<td>C-1</td>
<td>PL/SQL Compiler Limits</td>
<td>C-1</td>
</tr>
<tr>
<td>D-1</td>
<td>PL/SQL Reserved Words</td>
<td>D-1</td>
</tr>
<tr>
<td>D-2</td>
<td>PL/SQL Keywords</td>
<td>D-2</td>
</tr>
</tbody>
</table>
Preface

Oracle Database PL/SQL Language Reference describes and explains how to use PL/SQL, the Oracle procedural extension of SQL.

Topics

• Audience
• Documentation Accessibility
• Related Documents
• Conventions
• Syntax Descriptions

Audience

Oracle Database PL/SQL Language Reference is intended for anyone who is developing PL/SQL-based applications for either an Oracle Database or an Oracle TimesTen In-Memory Database, including:

• Programmers
• Systems analysts
• Project managers
• Database administrators

To use this document effectively, you need a working knowledge of:

• Oracle Database
• Structured Query Language (SQL)
• Basic programming concepts such as IF-THEN statements, loops, procedures, and functions

Documentation Accessibility

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

Access to Oracle Support

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

For more information, see these documents in the Oracle Database documentation set:

- Oracle Database SQL Language Reference
- Oracle Database PL/SQL Packages and Types Reference
- Oracle Database JSON Developer's Guide
- Oracle Database SODA for PL/SQL Developer's Guide
- Oracle Database Development Guide
- Oracle Database Administrator's Guide
- Oracle Database SecureFiles and Large Objects Developer's Guide
- Oracle Database Object-Relational Developer's Guide
- Oracle Database Concepts
- Oracle Database Performance Tuning Guide
- Oracle Database Sample Schemas

See Also:


Conventions

This document uses these text conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><strong>monospace</strong></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
<tr>
<td>{A</td>
<td>B</td>
</tr>
</tbody>
</table>

Also:

- *_view means all static data dictionary views whose names end with view. For example, *_ERRORS means ALL_ERRORS, DBA_ERRORS, and USER_ERRORS. For more information about any static data dictionary view, or about static dictionary views in general, see Oracle Database Reference.
- Table names not qualified with schema names are in the sample schema HR. For information about the sample schemas, see Oracle Database Sample Schemas.
Syntax Descriptions

Syntax descriptions are provided in this book for various SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF). See Oracle Database SQL Language Reference for information about how to interpret these descriptions.
1

Changes in This Release for Oracle Database PL/SQL Language Reference

1.1 New Features in Release 20c for Oracle Database PL/SQL Language Reference

For Oracle Database 20c, Oracle Database PL/SQL Language Reference documents these new features and enhancements.

See Also:
Oracle Database New Features Guide for the descriptions of all of the features that are new in Oracle Database Release 20c

1.1.1 PL/SQL Extended Iterators

PL/SQL programs use iteration controls to implement business logic over rows of data generated by SQL queries.

Iteration is the basic building block of PL/SQL. PL/SQL is enhanced to help you program iteration controls using new iterators in loops and in qualified expressions. The iterators are clear, simple, understandable, and efficient.

These iteration controls available are:

• Stepped range iteration controls
• Single expression iteration controls
• Collections iteration controls
• Cursor iteration controls

Multiple iteration controls may be chained together.

New stopping and skipping predicate clauses have been added.

The new mutability property of an iterand determines whether or not it can be assigned to in the loop body.

An iterand type can be implicitly or explicitly declared.
1.1.2 PL/SQL Qualified Expressions Enhancements

Aggregates and their necessary adjunct, qualified expressions, improve program clarity and programmer productivity.

Starting with Oracle Database Release 18c, any PL/SQL value can be provided by an expression (for example for a record or for an associative array) like a constructor provides an abstract datatype value. In PL/SQL, we use the terms “qualified expression” and “aggregate” rather than the SQL term “type constructor”, but the functionality is the same.

Starting with Oracle Database Release 20c, three new types of iterator choice association are added for use in qualified expressions. The basic iterator choice association extends the current iterator choice association by allowing a full iterator as the index. The index iterator choice association provides an index expression along with the value expression. The sequence iterator choice association allows a sequence of values to be added to the end of a collection. In each case, the expressions specified may reference the iterands.

1.1.3 SQL Macros

You can create SQL macros (SQM) to factor out common SQL expressions and statements into reusable, parameterized constructs that can be used in other SQL statements.

SQL macros can either be scalar expressions, typically used in SELECT lists, WHERE, GROUP BY, and HAVING clauses, to encapsulate calculations and business logic, or can be table expressions, typically used in a FROM clause, to act as a sort of polymorphic (parameterized) views.

SQL macros increase developer productivity, simplify collaborative development, and improve code quality.
1.1.4 New JSON Data Type

JSON is a new SQL and PL/SQL data type for JSON data. The data is stored in the
database in a binary form for faster access to nested JSON values.

You can use JSON data type and its instances in most places where a SQL data type
is allowed, including:
• As the column type for table or view DDL
• As a parameter type for a PL/SQL subprogram
• In expressions wherever a SQL/JSON function or condition are allowed

Some restrictions apply.

1.1.5 New Pragma SUPPRESSES_WARNING_6009

The SUPPRESSES_WARNING_6009 pragma allows more robust error handling and better
encapsulation and modularization.

The PL/SQL compiler issues warning PLW-06009 if it determines that an OTHERS
exception handler does not, in all cases, end in either an explicit RAISE statement or in
a call to the PL/SQL supplied procedure RAISE_APPLICATION_ERROR. The compiler’s
behavior may be too aggressive for some programming styles when programmers
supply their own reporting subroutines. This new pragma allows to quiet the warning.
1.1.6 PL/SQL Type Attributes in Non Persistable User Defined Types

You can use attributes of PL/SQL scalar data types, such as BOOLEAN and PLS_INTEGER, in non-persistable objects.

Instances of non-persistable types cannot persist on disk.

You can use non-persistable object types in your PL/SQL code if you have no desire to persist instances of these types. This is useful when you are developing programs following Oracle's object oriented programming model.

See Also:

- CREATE TYPE statement [NOT] PERSISTABLE clause for more information about the syntax and semantics

1.2 Deprecated Features

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

The command ALTER TYPE ... INVALIDATE is deprecated. Use the CASCADE clause instead.

The REPLACE clause of ALTER TYPE is deprecated. Use the alter_method_spec clause instead. Alternatively, you can recreate the type using the CREATE OR REPLACE TYPE statement.

For the syntax and semantics, see ALTER TYPE Statement

Starting with Oracle Database 12c release 1 (12.1), the compilation parameter PLSQL_DEBUG is deprecated.

To compile PL/SQL units for debugging, specify PLSQL_OPTIMIZE_LEVEL=1.

For information about compilation parameters, see PL/SQL Units and Compilation Parameters.

1.3 Desupported Features

No features in PL/SQL Language Reference have been desupported for 20c.

See Also:

- Oracle Database Upgrade Guide for more information about desupported features in Oracle Database 20c
Overview of PL/SQL

PL/SQL, the Oracle procedural extension of SQL, is a portable, high-performance transaction-processing language. This overview explains its advantages and briefly describes its main features and its architecture.

Topics

• Advantages of PL/SQL
• Main Features of PL/SQL
• Architecture of PL/SQL

2.1 Advantages of PL/SQL

PL/SQL offers several advantages over other programming languages.

PL/SQL has these advantages:

• Tight Integration with SQL
• High Performance
• High Productivity
• Portability
• Scalability
• Manageability
• Support for Object-Oriented Programming

2.1.1 Tight Integration with SQL

PL/SQL is tightly integrated with SQL, the most widely used database manipulation language.

For example:

• PL/SQL lets you use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, operators, and pseudocolumns.
• PL/SQL fully supports SQL data types.

You need not convert between PL/SQL and SQL data types. For example, if your PL/SQL program retrieves a value from a column of the SQL type VARCHAR2, it can store that value in a PL/SQL variable of the type VARCHAR2.

You can give a PL/SQL data item the data type of a column or row of a database table without explicitly specifying that data type (see "Using the %TYPE Attribute" and "Using the %ROWTYPE Attribute").
• PL/SQL lets you run a SQL query and process the rows of the result set one at a time (see "Processing a Query Result Set One Row at a Time").
• PL/SQL functions can be declared and defined in the WITH clauses of SQL SELECT statements (see Oracle Database SQL Language Reference).

PL/SQL supports both static and dynamic SQL. Static SQL is SQL whose full text is known at compile time. Dynamic SQL is SQL whose full text is not known until run time. Dynamic SQL lets you make your applications more flexible and versatile. For more information, see PL/SQL Static SQL and PL/SQL Dynamic SQL.

2.1.2 High Performance

PL/SQL lets you send a block of statements to the database, significantly reducing traffic between the application and the database.

Bind Variables

When you embed a SQL INSERT, UPDATE, DELETE, MERGE, or SELECT statement directly in your PL/SQL code, the PL/SQL compiler turns the variables in the WHERE and VALUES clauses into bind variables (for details, see "Resolution of Names in Static SQL Statements"). Oracle Database can reuse these SQL statements each time the same code runs, which improves performance.

PL/SQL does not create bind variables automatically when you use dynamic SQL, but you can use them with dynamic SQL by specifying them explicitly (for details, see "EXECUTE IMMEDIATE Statement").

Subprograms

PL/SQL subprograms are stored in executable form, which can be invoked repeatedly. Because stored subprograms run in the database server, a single invocation over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and invocation overhead. For more information about subprograms, see "Subprograms".

Optimizer

The PL/SQL compiler has an optimizer that can rearrange code for better performance. For more information about the optimizer, see "PL/SQL Optimizer".

2.1.3 High Productivity

PL/SQL has many features that save designing and debugging time, and it is the same in all environments.

PL/SQL lets you write compact code for manipulating data. Just as a scripting language like PERL can read, transform, and write data in files, PL/SQL can query, transform, and update data in a database.

If you learn to use PL/SQL with one Oracle tool, you can transfer your knowledge to other Oracle tools. For an overview of PL/SQL features, see "Main Features of PL/SQL".

2.1.4 Portability

PL/SQL is a portable and standard language for Oracle development.
You can run PL/SQL applications on any operating system and platform where Oracle Database runs.

2.1.5 Scalability

PL/SQL stored subprograms increase scalability by centralizing application processing on the database server.

The shared memory facilities of the shared server let Oracle Database support thousands of concurrent users on a single node. For more information about subprograms, see "Subprograms".

For further scalability, you can use Oracle Connection Manager to multiplex network connections. For information about Oracle Connection Manager, see "Oracle Database Net Services Reference".

2.1.6 Manageability

PL/SQL stored subprograms increase manageability because you can maintain only one copy of a subprogram, on the database server, rather than one copy on each client system.

Any number of applications can use the subprograms, and you can change the subprograms without affecting the applications that invoke them. For more information about subprograms, see "Subprograms".

2.1.7 Support for Object-Oriented Programming

PL/SQL allows defining object types that can be used in object-oriented designs.

PL/SQL supports object-oriented programming with "Abstract Data Types".

2.2 Main Features of PL/SQL

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages.

When you can solve a problem with SQL, you can issue SQL statements from your PL/SQL program, without learning new APIs.

Like other procedural programming languages, PL/SQL lets you declare constants and variables, control program flow, define subprograms, and trap runtime errors.

You can break complex problems into easily understandable subprograms, which you can reuse in multiple applications.

Topics

- Error Handling
- Blocks
- Variables and Constants
- Subprograms
- Packages
2.2.1 Error Handling

PL/SQL makes it easy to detect and handle errors.

When an error occurs, PL/SQL raises an exception. Normal execution stops and control transfers to the exception-handling part of the PL/SQL block. You do not have to check every operation to ensure that it succeeded, as in a C program.

For more information, see PL/SQL Error Handling.

2.2.2 Blocks

The basic unit of a PL/SQL source program is the block, which groups related declarations and statements.

A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END. These keywords divide the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required. A block can have a label.

Declarations are local to the block and cease to exist when the block completes execution, helping to avoid cluttered namespaces for variables and subprograms.

Blocks can be nested: Because a block is an executable statement, it can appear in another block wherever an executable statement is allowed.

You can submit a block to an interactive tool (such as SQL*Plus or Enterprise Manager) or embed it in an Oracle Precompiler or OCI program. The interactive tool or program runs the block one time. The block is not stored in the database, and for that reason, it is called an anonymous block (even if it has a label).

An anonymous block is compiled each time it is loaded into memory, and its compilation has three stages:

1. Syntax checking: PL/SQL syntax is checked, and a parse tree is generated.
2. Semantic checking: Type checking and further processing on the parse tree.
3. Code generation

![Note:](image)

An anonymous block is a SQL statement.

For syntax details, see "Block".
Example 2-1   PL/SQL Block Structure

This example shows the basic structure of a PL/SQL block.

```plsql
<< label >> (optional)
DECLARE    -- Declarative part (optional)
    -- Declarations of local types, variables, & subprograms
BEGIN      -- Executable part (required)
    -- Statements (which can use items declared in declarative part)
END;
```

2.2.3 Variables and Constants

PL/SQL lets you declare variables and constants, and then use them wherever you can use an expression.

As the program runs, the values of variables can change, but the values of constants cannot.

For more information, see "Declarations" and "Assigning Values to Variables".

2.2.4 Subprograms

A PL/SQL subprogram is a named PL/SQL block that can be invoked repeatedly.

If the subprogram has parameters, their values can differ for each invocation. PL/SQL has two types of subprograms, procedures and functions. A function returns a result.

For more information about PL/SQL subprograms, see PL/SQL Subprograms.

PL/SQL also lets you invoke external programs written in other languages.

For more information, see "External Subprograms".

2.2.5 Packages

A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions.

A package is compiled and stored in the database, where many applications can share its contents. You can think of a package as an application.

You can write your own packages—for details, see PLSQL Packages. You can also use the many product-specific packages that Oracle Database supplies. For information about these, see Oracle Database PL/SQL Packages and Types Reference.

2.2.6 Triggers

A trigger is a named PL/SQL unit that is stored in the database and run in response to an event that occurs in the database.

You can specify the event, whether the trigger fires before or after the event, and whether the trigger runs for each event or for each row affected by the event. For
example, you can create a trigger that runs every time an `INSERT` statement affects the `EMPLOYEES` table.

For more information about triggers, see PL/SQL Triggers.

### 2.2.7 Input and Output

Most PL/SQL input and output (I/O) is done with SQL statements that store data in database tables or query those tables. All other PL/SQL I/O is done with PL/SQL packages that Oracle Database supplies.

#### Table 2-1  PL/SQL I/O-Processing Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_OUTPUT</td>
<td>Lets PL/SQL blocks, subprograms, packages, and triggers display output.</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
<tr>
<td></td>
<td>Especially useful for displaying PL/SQL debugging information.</td>
<td></td>
</tr>
<tr>
<td>HTF</td>
<td>Has hypertext functions that generate HTML tags (for example, the <code>HTF.ANCHOR</code> function generates the HTML anchor tag <code>&lt;A&gt;</code>).</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
<tr>
<td>HTP</td>
<td>Has hypertext procedures that generate HTML tags.</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
<tr>
<td>DBMS_PIPE</td>
<td>Lets two or more sessions in the same instance communicate.</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
<tr>
<td>UTL_FILE</td>
<td>Lets PL/SQL programs read and write operating system files.</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
<tr>
<td>UTL_HTTP</td>
<td>Lets PL/SQL programs make Hypertext Transfer Protocol (HTTP) callouts, and access data on the Internet over HTTP.</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
<tr>
<td>UTL_SMTP</td>
<td>Sends electronic mails (emails) over Simple Mail Transfer Protocol (SMTP) as specified by RFC821.</td>
<td>Oracle Database PL/SQL Packages and Types Reference</td>
</tr>
</tbody>
</table>

To display output passed to `DBMS_OUTPUT`, you need another program, such as SQL*Plus. To see `DBMS_OUTPUT` output with SQL*Plus, you must first issue the SQL*Plus command `SET SERVEROUTPUT ON`.

Some subprograms in the packages in Table 2-1 can both accept input and display output, but they cannot accept data directly from the keyboard. To accept data directly from the keyboard, use the SQL*Plus commands `PROMPT` and `ACCEPT`. 
2.2.8 Data Abstraction

Data abstraction lets you work with the essential properties of data without being too involved with details.

You can design a data structure first, and then design algorithms that manipulate it.

Topics

- Cursors
- Composite Variables
- Using the %ROWTYPE Attribute
- Using the %TYPE Attribute
- Abstract Data Types

2.2.8.1 Cursors

A cursor is a pointer to a private SQL area that stores information about processing a specific SQL statement or PL/SQL SELECT INTO statement.

You can use the cursor to retrieve the rows of the result set one at a time. You can use cursor attributes to get information about the state of the cursor—for example, how many rows the statement has affected so far.

For more information about cursors, see “Cursors Overview”.

2.2.8.2 Composite Variables

A composite variable has internal components, which you can access individually.

You can pass entire composite variables to subprograms as parameters. PL/SQL has two kinds of composite variables, collections and records.

In a collection, the internal components are always of the same data type, and are called elements. You access each element by its unique index. Lists and arrays are classic examples of collections.
In a record, the internal components can be of different data types, and are called fields. You access each field by its name. A record variable can hold a table row, or some columns from a table row.

For more information about composite variables, see PL/SQL Collections and Records.

2.2.8.3 Using the %ROWTYPE Attribute

The %ROWTYPE attribute lets you declare a record that represents either a full or partial row of a database table or view.

For every column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly.

For more information about %ROWTYPE syntax and semantics, see "%ROWTYPE Attribute". For more details about its usage, see "Declaring Items using the %ROWTYPE Attribute".

2.2.8.4 Using the %TYPE Attribute

The %TYPE attribute lets you declare a data item of the same data type as a previously declared variable or column (without knowing what that type is).

If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly. The %TYPE attribute is particularly useful when declaring variables to hold database values. For more information about %TYPE syntax and semantics, see "%TYPE Attribute". For more details about its usage, see "Declaring Items using the %TYPE Attribute".

2.2.8.5 Abstract Data Types

An Abstract Data Type (ADT) consists of a data structure and subprograms that manipulate the data.

The variables that form the data structure are called attributes. The subprograms that manipulate the attributes are called methods.

ADTs are stored in the database. Instances of ADTs can be stored in tables and used as PL/SQL variables.

ADTs let you reduce complexity by separating a large system into logical components, which you can reuse.

In the static data dictionary view *_OBJECTS, the OBJECT_TYPE of an ADT is TYPE. In the static data dictionary view *_TYPES, the TYPECODE of an ADT is OBJECT.

For more information about ADTs, see "CREATE TYPE Statement".

Note:
ADTs are also called user-defined types and object types.
2.2.9 Control Statements

Control statements are the most important PL/SQL extension to SQL.

PL/SQL has three categories of control statements:

- **Conditional selection statements**, which let you run different statements for different data values.
  
  For more information, see "Conditional Selection Statements".

- **Loop statements**, which let you repeat the same statements with a series of different data values.
  
  For more information, see "LOOP Statements".

- **Sequential control statements**, which allow you to go to a specified, labeled statement, or to do nothing.
  
  For more information, see "Sequential Control Statements".

2.2.10 Conditional Compilation

Conditional compilation lets you customize the functionality in a PL/SQL application without removing source text.

For example, you can:

- Use new features with the latest database release, and disable them when running the application in an older database release.

- Activate debugging or tracing statements in the development environment, and hide them when running the application at a production site.

For more information, see "Conditional Compilation".

2.2.11 Processing a Query Result Set One Row at a Time

PL/SQL lets you issue a SQL query and process the rows of the result set one at a time.

You can use a basic loop, or you can control the process precisely by using individual statements to run the query, retrieve the results, and finish processing.

**Example 2-2  Processing Query Result Rows One at a Time**

This example uses a basic loop.

```sql
BEGIN
  FOR someone IN (  
    SELECT * FROM employees
    WHERE employee_id < 120  
    ORDER BY employee_id
  )
  ...  
END;
```
2.3 Architecture of PL/SQL

Basic understanding of the PL/SQL architecture is beneficial to PL/SQL programmers.

Topics

- PL/SQL Engine
- PL/SQL Units and Compilation Parameters

2.3.1 PL/SQL Engine

The PL/SQL compilation and runtime system is an engine that compiles and runs PL/SQL units.

The engine can be installed in the database or in an application development tool, such as Oracle Forms.

In either environment, the PL/SQL engine accepts as input any valid PL/SQL unit. The engine runs procedural statements, but sends SQL statements to the SQL engine in the database, as shown in Figure 2-1.
Typically, the database processes PL/SQL units.

When an application development tool processes PL/SQL units, it passes them to its local PL/SQL engine. If a PL/SQL unit contains no SQL statements, the local engine processes the entire PL/SQL unit. This is useful if the application development tool can benefit from conditional and iterative control.

For example, Oracle Forms applications frequently use SQL statements to test the values of field entries and do simple computations. By using PL/SQL instead of SQL, these applications can avoid calls to the database.

2.3.2 PL/SQL Units and Compilation Parameters

PL/SQL units are affected by PL/SQL compilation parameters (a category of database initialization parameters). Different PL/SQL units—for example, a package specification and its body—can have different compilation parameter settings.

A PL/SQL unit is one of these:

- PL/SQL anonymous block
- FUNCTION
- LIBRARY
- PACKAGE
- PACKAGE BODY
- PROCEDURE
- TRIGGER
- TYPE
- TYPE BODY

Table 2-2 summarizes the PL/SQL compilation parameters. To display the values of these parameters for specified or all PL/SQL units, query the static data dictionary.
view **ALL_PLSQL_OBJECT_SETTINGS**. For information about this view, see *Oracle Database Reference*.

### Table 2-2 PL/SQL Compilation Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLSCOPE_SETTINGS</td>
<td>Controls the compile-time collection, cross-reference, and storage of PL/SQL source text identifier data. Used by the PL/Scope tool (see <em>Oracle Database Development Guide</em>). For more information about PLSCOPE_SETTINGS, see <em>Oracle Database Reference</em>.</td>
</tr>
<tr>
<td>PLSQL_CCFLAGS</td>
<td>Lets you control conditional compilation of each PL/SQL unit independently. For more information about PLSQL_CCFLAGS, see &quot;How Conditional Compilation Works&quot; and <em>Oracle Database Reference</em>.</td>
</tr>
</tbody>
</table>
| PLSQL_CODE_TYPE          | Specifies the compilation mode for PL/SQL units—**INTERPRETED** (the default) or **NATIVE**. For information about which mode to use, see "Determining Whether to Use PL/SQL Native Compilation". If the optimization level (set by PLSQL_OPTIMIZE_LEVEL) is less than 2:  
  • The compiler generates interpreted code, regardless of PLSQL_CODE_TYPE.  
  • If you specify NATIVE, the compiler warns you that NATIVE was ignored. For more information about PLSQL_CODE_TYPE, see *Oracle Database Reference*. |
| PLSQL_OPTIMIZE_LEVEL     | Specifies the optimization level at which to compile PL/SQL units (the higher the level, the more optimizations the compiler tries to make). PLSQL_OPTIMIZE_LEVEL=1 instructs the PL/SQL compiler to generate and store code for use by the PL/SQL debugger. For more information about PLSQL_OPTIMIZE_LEVEL, see "PL/SQL Optimizer" and *Oracle Database Reference*. |
| PLSQL_WARNINGS           | Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors. For more information about PLSQL_WARNINGS, see "Compile-Time Warnings" and *Oracle Database Reference*. |
| NLS_LENGTH_SEMANTICS     | Lets you create **CHAR** and **VARCHAR2** columns using either byte-length or character-length semantics. For more information about byte and character length semantics, see "CHAR and VARCHAR2 Variables". For more information about NLS_LENGTH_SEMANTICS, see *Oracle Database Reference*. |
Table 2-2  (Cont.) PL/SQLCompilation Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERMIT_92_WRAP_FORMAT</td>
<td>Specifies whether the 12.1 PL/SQL compiler can use wrapped packages that were compiled with the 9.2 PL/SQL compiler. The default value is TRUE. For more information about wrapped packages, see PL/SQL Source Text Wrapping. For more information about PERMIT_92_WRAP_FORMAT, see Oracle Database Reference.</td>
</tr>
</tbody>
</table>

**Note:**

The compilation parameter PLSQL_DEBUG, which specifies whether to compile PL/SQL units for debugging, is deprecated. To compile PL/SQL units for debugging, specify PLSQL_OPTIMIZE_LEVEL=1.

The compile-time values of the parameters in Table 2-2 are stored with the metadata of each stored PL/SQL unit, which means that you can reuse those values when you explicitly recompile the unit. (A stored PL/SQL unit is created with one of the "CREATE [ OR REPLACE ] Statements". An anonymous block is not a stored PL/SQL unit.)

To explicitly recompile a stored PL/SQL unit and reuse its parameter values, you must use an ALTER statement with both the COMPILE clause and the REUSE SETTINGS clause. All ALTER statements have this clause. For a list of ALTER statements, see "ALTER Statements".
3

PL/SQL Language Fundamentals

The PL/SQL language fundamental components are explained.

• Character Sets
• Lexical Units
• Declarations
• References to Identifiers
• Scope and Visibility of Identifiers
• Assigning Values to Variables
• Expressions
• Error-Reporting Functions
• Conditional Compilation

3.1 Character Sets

Any character data to be processed by PL/SQL or stored in a database must be represented as a sequence of bytes. The byte representation of a single character is called a character code. A set of character codes is called a character set.

Every Oracle database supports a database character set and a national character set. PL/SQL also supports these character sets. This document explains how PL/SQL uses the database character set and national character set.

Topics

• Database Character Set
• National Character Set
• About Data-Bound Collation

See Also:

Oracle Database Globalization Support Guide for general information about character sets

3.1.1 Database Character Set

PL/SQL uses the database character set to represent:

• Stored source text of PL/SQL units
For information about PL/SQL units, see “PL/SQL Units and Compilation Parameters”.

- **Character values of data types** `CHAR`, `VARCHAR2`, `CLOB`, and `LONG`
  
  For information about these data types, see “SQL Data Types”.

The database character set can be either single-byte, mapping each supported character to one particular byte, or multibyte-varying-width, mapping each supported character to a sequence of one, two, three, or four bytes. The maximum number of bytes in a character code depends on the particular character set.

Every database character set includes these basic characters:

- **Latin letters**: A through Z and a through z
- **Decimal digits**: 0 through 9
- **Punctuation characters** in Table 3-1
- **Whitespace characters**: space, tab, new line, and carriage return

PL/SQL source text that uses only the basic characters can be stored and compiled in any database. PL/SQL source text that uses nonbasic characters can be stored and compiled only in databases whose database character sets support those nonbasic characters.

### Table 3-1  Punctuation Characters in Every Database Character Set

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left angle bracket</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right angle bracket</td>
</tr>
<tr>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>-</td>
<td>Hyphen or minus sign</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>/</td>
<td>Slash</td>
</tr>
<tr>
<td>=</td>
<td>Equal sign</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>.</td>
<td>Period</td>
</tr>
<tr>
<td>!</td>
<td>Exclamation point</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe or single quotation mark</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation mark or double quotation mark</td>
</tr>
<tr>
<td>@</td>
<td>At sign</td>
</tr>
<tr>
<td>%</td>
<td>Percent sign</td>
</tr>
<tr>
<td>#</td>
<td>Number sign</td>
</tr>
</tbody>
</table>
Table 3-1  (Cont.) Punctuation Characters in Every Database Character Set

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Dollar sign</td>
</tr>
<tr>
<td>_</td>
<td>Underscore</td>
</tr>
<tr>
<td></td>
<td>Vertical bar</td>
</tr>
</tbody>
</table>

See Also:  
*Oracle Database Globalization Support Guide* for more information about the database character set

3.1.2 National Character Set

PL/SQL uses the **national character set** to represent character values of data types NCHAR, NVARCHAR2 and NCLOB.

See Also:
- "SQL Data Types" for information about these data types
- *Oracle Database Globalization Support Guide* for more information about the national character set

3.1.3 About Data-Bound Collation

Collation (also called sort ordering) is a set of rules that determines if a character string equals, precedes, or follows another string when the two strings are compared and sorted.

Different collations correspond to rules of different spoken languages. Collation-sensitive operations are operations that compare text and need a collation to control the comparison rules. The equality operator and the built-in function `INSTR` are examples of collation-sensitive operations.

Starting with Oracle Database 12c release 2 (12.2), a new architecture provides control of the collation to be applied to operations on character data. In the new architecture, collation becomes an attribute of character data, analogous to a data type. You can now declare collation for a column and this collation is automatically applied by all collation-sensitive SQL operations referencing the column. The data-bound collation feature uses syntax and semantics compatible with the ISO/IEC SQL standard.

The PL/SQL language has limited support for the data-bound collation architecture. All data processed in PL/SQL expressions is assumed to have the compatibility collation `USING_NLS_COMP`. This pseudo-collation instructs collation-sensitive operators to behave in the same way as in previous Oracle Database releases. That is, the values
of the session parameters NLS_COMP and NLS_SORT determine the collation to use. However, all SQL statements embedded or constructed dynamically in PL/SQL fully support the new architecture.

A new property called default collation has been added to tables, views, materialized views, packages, stored procedures, stored functions, triggers, and types. The default collation of a unit determines the collation for data containers, such as columns, variables, parameters, literals, and return values, that do not have their own explicit collation declaration in that unit. The default collation for packages, stored procedures, stored functions, triggers, and types must be USING_NLS_COMP.

For syntax and semantics, see the DEFAULT COLLATION Clause.

To facilitate the creation of PL/SQL units in a schema that has a schema default collation other than USING_NLS_COMP, the syntax and semantics for the following statements enable an explicit declaration of the object's default collation to be USING_NLS_COMP:

• CREATE FUNCTION Statement
• CREATE PACKAGE Statement
• CREATE PROCEDURE Statement
• CREATE TRIGGER Statement
• CREATE TYPE Statement

See Also:

• Oracle Database Globalization Support Guide for more information about specifying data-bound collation for PL/SQL units
• Oracle Database Globalization Support Guide for more information about effective schema default collation

3.2 Lexical Units

The lexical units of PL/SQL are its smallest individual components—delimiters, identifiers, literals, pragmas, and comments.

Topics

• Delimiters
• Identifiers
• Literals
• Pragmas
• Comments
• Whitespace Characters Between Lexical Units
3.2.1 Delimiters

A **delimiter** is a character, or character combination, that has a special meaning in PL/SQL.

Do not embed any other characters (including whitespace characters) inside a delimiter.

**Table 3-2** summarizes the PL/SQL delimiters.

<table>
<thead>
<tr>
<th>Delimiter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition operator</td>
</tr>
<tr>
<td>:=</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>=&gt;</td>
<td>Association operator</td>
</tr>
<tr>
<td>%</td>
<td>Attribute indicator</td>
</tr>
<tr>
<td>'</td>
<td>Character string delimiter</td>
</tr>
<tr>
<td>.</td>
<td>Component indicator</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Division operator</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation operator</td>
</tr>
<tr>
<td>(</td>
<td>Expression or list delimiter (begin)</td>
</tr>
<tr>
<td>)</td>
<td>Expression or list delimiter (end)</td>
</tr>
<tr>
<td>:</td>
<td>Host variable indicator</td>
</tr>
<tr>
<td>,</td>
<td>Item separator</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Label delimiter (begin)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Label delimiter (end)</td>
</tr>
<tr>
<td>/*</td>
<td>Multiline comment delimiter (begin)</td>
</tr>
<tr>
<td>*/</td>
<td>Multiline comment delimiter (end)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication operator</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quoted identifier delimiter</td>
</tr>
<tr>
<td>..</td>
<td>Range operator</td>
</tr>
<tr>
<td>=</td>
<td>Relational operator (equal)</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Relational operator (not equal)</td>
</tr>
<tr>
<td>!</td>
<td>Relational operator (not equal)</td>
</tr>
<tr>
<td>^=</td>
<td>Relational operator (not equal)</td>
</tr>
<tr>
<td>^=</td>
<td>Relational operator (not equal)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Relational operator (less than)</td>
</tr>
<tr>
<td>&gt;</td>
<td>Relational operator (greater than)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Relational operator (less than or equal)</td>
</tr>
</tbody>
</table>
### Table 3-2 (Cont.) PL/SQL Delimiters

<table>
<thead>
<tr>
<th>Delimiter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=</td>
<td>Relational operator (greater than or equal)</td>
</tr>
<tr>
<td>@</td>
<td>Remote access indicator</td>
</tr>
<tr>
<td>--</td>
<td>Single-line comment indicator</td>
</tr>
<tr>
<td>;</td>
<td>Statement terminator</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or negation operator</td>
</tr>
</tbody>
</table>

### 3.2.2 Identifiers

Identifiers name PL/SQL elements, which include:

- Constants
- Cursors
- Exceptions
- Keywords
- Labels
- Packages
- Reserved words
- Subprograms
- Types
- Variables

Every character in an identifier, alphabetic or not, is significant. For example, the identifiers `lastname` and `last_name` are different.

You must separate adjacent identifiers by one or more whitespace characters or a punctuation character.

Except as explained in "Quoted User-Defined Identifiers", PL/SQL is case-insensitive for identifiers. For example, the identifiers `lastname`, `LastName`, and `LASTNAME` are the same.

#### Topics

- Reserved Words and Keywords
- Predefined Identifiers
- User-Defined Identifiers

### 3.2.2.1 Reserved Words and Keywords

Reserved words and keywords are identifiers that have special meaning in PL/SQL.

You cannot use reserved words as ordinary user-defined identifiers. You can use them as quoted user-defined identifiers, but it is not recommended. For more information, see "Quoted User-Defined Identifiers".
You can use keywords as ordinary user-defined identifiers, but it is not recommended. For lists of PL/SQL reserved words and keywords, see Table D-1 and Table D-2, respectively.

3.2.2.2 Predefined Identifiers

**Predefined identifiers** are declared in the predefined package `STANDARD`.

An example of a predefined identifier is the exception `INVALID_NUMBER`.

For a list of predefined identifiers, connect to Oracle Database as a user who has the DBA role and use this query:

```
SELECT TYPE_NAME FROM ALL_TYPES WHERE PREDEFINED='YES';
```

You can use predefined identifiers as user-defined identifiers, but it is not recommended. Your local declaration overrides the global declaration (see “Scope and Visibility of Identifiers”).

3.2.2.3 User-Defined Identifiers

A **user-defined identifier** is:

- Composed of characters from the database character set
- Either ordinary or quoted

Tip:

Make user-defined identifiers meaningful. For example, the meaning of `cost_per_thousand` is obvious, but the meaning of `cpt` is not.

3.2.2.3.1 Ordinary User-Defined Identifiers

An ordinary user-defined identifier:

- Begins with a letter
- Can include letters, digits, and these symbols:
  - Dollar sign ($)
  - Number sign (#)
  - Underscore (_)
- Is not a reserved word (listed in Table D-1).

The database character set defines which characters are classified as letters and digits. If COMPATIBLE is set to a value of 12.2 or higher, the representation of the identifier in the database character set cannot exceed 128 bytes. If COMPATIBLE is set to a value of 12.1 or lower, the limit is 30 bytes.

Examples of acceptable ordinary user-defined identifiers:

```
X
t2
phone#
```
Examples of unacceptable ordinary user-defined identifiers:

mine&yours
debit-amount
on/off
user id

3.2.2.3.2 Quoted User-Defined Identifiers

A quoted user-defined identifier is enclosed in double quotation marks.

Between the double quotation marks, any characters from the database character set are allowed except double quotation marks, new line characters, and null characters. For example, these identifiers are acceptable:

"X+Y"
"last name"
"on/off switch"
"employee(s)"
"*** header info ***"

If COMPATIBLE is set to a value of 12.2 or higher, the representation of the quoted identifier in the database character set cannot exceed 128 bytes (excluding the double quotation marks). If COMPATIBLE is set to a value of 12.1 or lower, the limit is 30 bytes.

A quoted user-defined identifier is case-sensitive, with one exception: If a quoted user-defined identifier, without its enclosing double quotation marks, is a valid ordinary user-defined identifier, then the double quotation marks are optional in references to the identifier, and if you omit them, then the identifier is case-insensitive.

It is not recommended, but you can use a reserved word as a quoted user-defined identifier. Because a reserved word is not a valid ordinary user-defined identifier, you must always enclose the identifier in double quotation marks, and it is always case-sensitive.

Example 3-1 Valid Case-Insensitive Reference to Quoted User-Defined Identifier

In this example, the quoted user-defined identifier "HELLO", without its enclosing double quotation marks, is a valid ordinary user-defined identifier. Therefore, the reference Hello is valid.

DECLARE
 "HELLO" varchar2(10) := 'hello';
BEGIN
 DBMS_Output.Put_Line(Hello);
 END;
/

Result:

hello
Example 3-2  Invalid Case-Insensitive Reference to Quoted User-Defined Identifier

In this example, the reference "Hello" is invalid, because the double quotation marks make the identifier case-sensitive.

```
DECLARE
   "HELLO" varchar2(10) := 'hello';
BEGIN
   DBMS_Output.Put_Line(Hello);
END;
/
```

Result:

```
DBMS_Output.Put_Line(Hello);
* ERROR at line 4:
ORA-06550: line 4, column 25:
PLS-00201: identifier 'Hello' must be declared
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```

Example 3-3  Reserved Word as Quoted User-Defined Identifier

This example declares quoted user-defined identifiers "BEGIN", "Begin", and "begin". Although BEGIN, Begin, and begin represent the same reserved word, "BEGIN", "Begin", and "begin" represent different identifiers.

```
DECLARE
   "BEGIN" varchar2(15) := 'UPPERCASE';
   "Begin" varchar2(15) := 'Initial Capital';
   "begin" varchar2(15) := 'lowercase';
BEGIN
   DBMS_Output.Put_Line("BEGIN");
   DBMS_Output.Put_Line("Begin");
   DBMS_Output.Put_Line("begin");
END;
/
```

Result:

```
UPPERCASE
Initial Capital
lowercase
```

PL/SQL procedure successfully completed.

Example 3-4  Neglecting Double Quotation Marks

This example references a quoted user-defined identifier that is a reserved word, neglecting to enclose it in double quotation marks.

```
DECLARE
   "HELLO" varchar2(10) := 'hello';  -- HELLO is not a reserved word
   "BEGIN" varchar2(10) := 'begin';  -- BEGIN is a reserved word
BEGIN
   DBMS_Output.Put_Line(Hello);
   DBMS_Output.Put_Line("BEGIN");
end;
/
```

Chapter 3
Lexical Units

3-9
Example 3-5 Neglecting Case-Sensitivity

This example references a quoted user-defined identifier that is a reserved word, neglecting its case-sensitivity.

DECLARE
   "HELLO" varchar2(10) := 'hello'; -- HELLO is not a reserved word
   "BEGIN" varchar2(10) := 'begin'; -- BEGIN is a reserved word
BEGIN
   DBMS_Output.Put_Line(Hello); -- Identifier is case-insensitive
   DBMS_Output.Put_Line("Begin"); -- Identifier is case-sensitive
END;
/

Result:
   DBMS_Output.Put_Line("Begin"); -- Identifier is case-sensitive
   *
   ERROR at line 6:
   ORA-06550: line 6, column 25:
   PLS-00201: identifier 'Begin' must be declared
   ORA-06550: line 6, column 3:
   PL/SQL: Statement ignored

3.2.3 Literals

A literal is a value that is neither represented by an identifier nor calculated from other values.

For example, 123 is an integer literal and 'abc' is a character literal, but 1+2 is not a literal.

PL/SQL literals include all SQL literals (described in Oracle Database SQL Language Reference) and BOOLEAN literals (which SQL does not have). A BOOLEAN literal is the predefined logical value TRUE, FALSE, or NULL. NULL represents an unknown value.

Note:

Like Oracle Database SQL Language Reference, this document uses the terms character literal and string interchangeably.
When using character literals in PL/SQL, remember:

- **Character literals are case-sensitive.**
  
  For example, 'Z' and 'z' are different.

- **Whitespace characters are significant.**
  
  For example, these literals are different:

```sql
'abc'
' abc'
'abc '  
' abc'
'a b c'
```

- **PL/SQL has no line-continuation character that means “this string continues on the next source line.”** If you continue a string on the next source line, then the string includes a line-break character.

  For example, this PL/SQL code:

  ```sql
  BEGIN
    DBMS_OUTPUT.PUT_LINE('This string breaks here.');
  END;
  /
  
  Prints this:

  This string breaks here.

  If your string does not fit on a source line and you do not want it to include a line-break character, then construct the string with the concatenation operator (||).

  For example, this PL/SQL code:

  ```sql
  BEGIN
    DBMS_OUTPUT.PUT_LINE('This string ' || 'contains no line-break character.');</n
  END;
  /
  
  Prints this:

  This string contains no line-break character.

  For more information about the concatenation operator, see "Concatenation Operator".

- **'0' through '9' are not equivalent to the integer literals 0 through 9.**

  However, because PL/SQL converts them to integers, you can use them in arithmetic expressions.

- **A character literal with zero characters has the value NULL and is called a null string.**

  However, this NULL value is not the BOOLEAN value NULL.

- **An ordinary character literal** is composed of characters in the database character set.
For information about the database character set, see Oracle Database Globalization Support Guide.

- A **national character literal** is composed of characters in the **national character set**.
  
  For information about the national character set, see Oracle Database Globalization Support Guide.

### 3.2.4 Pragmas

A **pragma** is an instruction to the compiler that it processes at compile time.

A pragma begins with the reserved word **PRAGMA** followed by the name of the pragma. Some pragmas have arguments. A pragma may appear before a declaration or a statement. Additional restrictions may apply for specific pragmas. The extent of a pragma’s effect depends on the pragma. A pragma whose name or argument is not recognized by the compiler has no effect.

```
pragma ::= autonomous_trans_pragma
       | coverage_pragma
       | deprecate_pragma
       | exception_init_pragma
       | inline_pragma
       | restrict_references_pragma
       | serially_reusable_pragma
       | suppresses_warning_6009_pragma
       | udf_pragmas
```

For information about pragmas syntax and semantics, see:

- "AUTONOMOUS_TRANSACTION Pragma"
- "COVERAGE Pragma"
- "DEPRECATE Pragma"
- "EXCEPTION_INIT Pragma"
- "INLINE Pragma"
- "RESTRICT_REFERENCES Pragma"
- "SERIALLY_REUSABLE Pragma"
- "SUPPRESSES_WARNING_6009 Pragma"
- "UDF Pragma"
3.2.5 Comments

The PL/SQL compiler ignores comments. Their purpose is to help other application developers understand your source text.

Typically, you use comments to describe the purpose and use of each code segment. You can also disable obsolete or unfinished pieces of code by turning them into comments.

Topics

• Single-Line Comments
• Multiline Comments

See Also:
"Comment"

3.2.5.1 Single-Line Comments

A single-line comment begins with -- and extends to the end of the line.

Caution:

Do not put a single-line comment in a PL/SQL block to be processed dynamically by an Oracle Precompiler program. The Oracle Precompiler program ignores end-of-line characters, which means that a single-line comment ends when the block ends.

While testing or debugging a program, you can disable a line of code by making it a comment. For example:

-- DELETE FROM employees WHERE comm_pct IS NULL

Example 3-6 Single-Line Comments

This example has three single-line comments.

DECLARE
    howmany    NUMBER;
    num_tables NUMBER;
BEGIN
    -- Begin processing
    SELECT COUNT(*) INTO howmany
    FROM USER_OBJECTS
    WHERE OBJECT_TYPE = 'TABLE'; -- Check number of tables
    num_tables := howmany; -- Compute another value
END;
/
3.2.5.2 Multiline Comments

A multiline comment begins with /*, ends with */*, and can span multiple lines.

You can use multiline comment delimiters to "comment out" sections of code. When doing so, be careful not to cause nested multiline comments. One multiline comment cannot contain another multiline comment. However, a multiline comment can contain a single-line comment. For example, this causes a syntax error:

```/*
IF 2 + 2 = 4 THEN
  some_condition := TRUE;
/* We expect this THEN to always be performed */
END IF;
*/
```

This does not cause a syntax error:

```/*
IF 2 + 2 = 4 THEN
  some_condition := TRUE;
-- We expect this THEN to always be performed
END IF;
*/
```

**Example 3-7  Multiline Comments**

This example has two multiline comments. (The SQL function **TO_CHAR** returns the character equivalent of its argument. For more information about **TO_CHAR**, see **Oracle Database SQL Language Reference**.)

```sql
DECLARE
  some_condition  BOOLEAN;
  pi              NUMBER := 3.1415926;
  radius          NUMBER := 15;
  area            NUMBER;
BEGIN
  /* Perform some simple tests and assignments */
  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
    /* We expect this THEN to always be performed */
    END IF;

  /* This line computes the area of a circle using pi,
  which is the ratio between the circumference and diameter.
  After the area is computed, the result is displayed. */
  area := pi * radius**2;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || TO_CHAR(area));
END;
/
```

Result:

The area is: 706.858335
3.2.6 Whitespace Characters Between Lexical Units

You can put whitespace characters between lexical units, which often makes your source text easier to read.

Example 3-8  Whitespace Characters Improving Source Text Readability

```plsql
DECLARE
  x    NUMBER := 10;
  y    NUMBER := 5;
  max  NUMBER;
BEGIN
  IF x>y THEN max:=x;ELSE max:=y;END IF;   -- correct but hard to read
  -- Easier to read:
  IF x > y THEN
    max:=x;
  ELSE
    max:=y;
  END IF;
END;
/```

3.3 Declarations

A declaration allocates storage space for a value of a specified data type, and names the storage location so that you can reference it.

You must declare objects before you can reference them. Declarations can appear in the declarative part of any block, subprogram, or package.

Topics

- Declaring Variables
- Declaring Constants
- Initial Values of Variables and Constants
- NOT NULL Constraint
- Declaring Items using the %TYPE Attribute

For information about declaring objects other than variables and constants, see the syntax of declare_section in "Block".

3.3.1 NOT NULL Constraint

You can impose the NOT NULL constraint on a scalar variable or constant (or scalar component of a composite variable or constant).

The NOT NULL constraint prevents assigning a null value to the item. The item can acquire this constraint either implicitly (from its data type) or explicitly.

A scalar variable declaration that specifies NOT NULL, either implicitly or explicitly, must assign an initial value to the variable (because the default initial value for a scalar variable is NULL).
PL/SQL treats any zero-length string as a **NULL** value. This includes values returned by character functions and **BOOLEAN** expressions.

To test for a **NULL** value, use the "**IS [NOT] NULL Operator**".

**Examples**

**Example 3-9**  **Variable Declaration with NOT NULL Constraint**

In this example, the variable `acct_id` acquires the **NOT NULL** constraint explicitly, and the variables `a`, `b`, and `c` acquire it from their data types.

```
DECLARE
    acct_id      INTEGER(4) NOT NULL := 9999;
    a            NATURALN := 9999;
    b            POSITIVEN := 9999;
    c            SIMPLE_INTEGER := 9999;
BEGIN
    NULL;
END;
/
```

**Example 3-10**  **Variables Initialized to NULL Values**

In this example, all variables are initialized to **NULL**.

```
DECLARE
    null_string  VARCHAR2(80) := TO_CHAR('');
    address      VARCHAR2(80);
    zip_code     VARCHAR2(80) := SUBSTR(address, 25, 0);
    name         VARCHAR2(80);
    valid        BOOLEAN := (name != '');
BEGIN
    NULL;
END;
/
```

**3.3.2 Declaring Variables**

A variable declaration always specifies the name and data type of the variable.

For most data types, a variable declaration can also specify an initial value.

The variable name must be a valid user-defined identifier.

The data type can be any PL/SQL data type. The PL/SQL data types include the SQL data types. A data type is either scalar (without internal components) or composite (with internal components).

**Example**

**Example 3-11**  **Scalar Variable Declarations**

This example declares several variables with scalar data types.

```
DECLARE
    part_number       NUMBER(6);     -- SQL data type
    part_name         VARCHAR2(20);  -- SQL data type
    in_stock          BOOLEAN;       -- PL/SQL-only data type
    part_price        NUMBER(6,2);   -- SQL data type
    part_description  VARCHAR2(50);  -- SQL data type
```
3.3.3 Declaring Constants

A constant holds a value that does not change.

The information in "Declaring Variables" also applies to constant declarations, but a constant declaration has two more requirements: the keyword CONSTANT and the initial value of the constant. (The initial value of a constant is its permanent value.)

Example 3-12 Constant Declarations

This example declares three constants with scalar data types.

```
BEGIN
    DECLARE
        credit_limit CONSTANT REAL := 5000.00;  -- SQL data type
        max_days_in_year CONSTANT INTEGER := 366;      -- SQL data type
        urban_legend CONSTANT BOOLEAN := FALSE;    -- PL/SQL-only data type
    BEGIN
        NULL;
    END;
END;
```

3.3.4 Initial Values of Variables and Constants

In a variable declaration, the initial value is optional unless you specify the NOT NULL constraint. In a constant declaration, the initial value is required.

If the declaration is in a block or subprogram, the initial value is assigned to the variable or constant every time control passes to the block or subprogram. If the declaration is in a package specification, the initial value is assigned to the variable or constant for each session (whether the variable or constant is public or private).

To specify the initial value, use either the assignment operator (:=) or the keyword DEFAULT, followed by an expression. The expression can include previously declared constants and previously initialized variables.

If you do not specify an initial value for a variable, assign a value to it before using it in any other context.
Examples

Example 3-13 Variable and Constant Declarations with Initial Values

This example assigns initial values to the constant and variables that it declares. The initial value of \texttt{area} depends on the previously declared constant \texttt{pi} and the previously initialized variable \texttt{radius}.

\begin{verbatim}
DECLARE
  hours_worked INTEGER := 40;
  employee_count INTEGER := 0;

  pi CONSTANT REAL := 3.14159;
  radius REAL := 1;
  area REAL := (pi * radius**2);
BEGIN
  NULL;
END;
/
\end{verbatim}

Example 3-14 Variable Initialized to NULL by Default

In this example, the variable \texttt{counter} has the initial value \texttt{NULL}, by default. The example uses the "IS [NOT] NULL Operator" to show that \texttt{NULL} is different from zero.

\begin{verbatim}
DECLARE
  counter INTEGER;  -- initial value is NULL by default
BEGIN
  counter := counter + 1;  -- NULL + 1 is still NULL

  IF counter IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('counter is NULL.);
  END IF;
END;
/
\end{verbatim}

Result:

\begin{verbatim}
counter is NULL.
\end{verbatim}

Related Topics

- "Declaring Associative Array Constants" for information about declaring constant associative arrays
- "Declaring Record Constants" for information about declaring constant records
- "NOT NULL Constraint"

3.3.5 Declaring Items using the %TYPE Attribute

The %TYPE attribute lets you declare a data item of the same data type as a previously declared variable or column (without knowing what that type is). If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly.

The syntax of the declaration is:

\begin{verbatim}
referencing_item referenced_item%TYPE;
\end{verbatim}
For the kinds of items that can be referencing and referenced items, see "%TYPE Attribute".

The referencing item inherits the following from the referenced item:

- Data type and size
- Constraints (unless the referenced item is a column)

The referencing item does not inherit the initial value of the referenced item. Therefore, if the referencing item specifies or inherits the NOT NULL constraint, you must specify an initial value for it.

The %TYPE attribute is particularly useful when declaring variables to hold database values. The syntax for declaring a variable of the same type as a column is:

```
variable_name table_name.column_name%TYPE;
```

See Also:

"Declaring Items using the %ROWTYPE Attribute", which lets you declare a record variable that represents either a full or partial row of a database table or view

Examples

**Example 3-15  Declaring Variable of Same Type as Column**

In this example, the variable `surname` inherits the data type and size of the column `employees.last_name`, which has a NOT NULL constraint. Because `surname` does not inherit the NOT NULL constraint, its declaration does not need an initial value.

```sql
DECLARE
    surname  employees.last_name%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

**Example 3-16  Declaring Variable of Same Type as Another Variable**

In this example, the variable `surname` inherits the data type, size, and NOT NULL constraint of the variable `name`. Because `surname` does not inherit the initial value of `name`, its declaration needs an initial value (which cannot exceed 25 characters).

```sql
DECLARE
    name     VARCHAR(25) NOT NULL := 'Smith';
    surname  name%TYPE := 'Jones';
BEGIN
    DBMS_OUTPUT.PUT_LINE('name=' || name);
    DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```
3.4 References to Identifiers

When referencing an identifier, you use a name that is either simple, qualified, remote, or both qualified and remote.

The **simple name** of an identifier is the name in its declaration. For example:

```
DECLARE
    a INTEGER;  -- Declaration
BEGIN
    a := 1;     -- Reference with simple name
END;
/
```

If an identifier is declared in a named PL/SQL unit, you can (and sometimes must) reference it with its **qualified name**. The syntax (called **dot notation**) is:

```
unit_name.simple_identifier_name
```

For example, if package `p` declares identifier `a`, you can reference the identifier with the qualified name `p.a`. The unit name also can (and sometimes must) be qualified. You **must** qualify an identifier when it is not visible (see "Scope and Visibility of Identifiers").

If the identifier names an object on a remote database, you must reference it with its **remote name**. The syntax is:

```
simple_identifier_name@link_to_remote_database
```

If the identifier is declared in a PL/SQL unit on a remote database, you must reference it with its **qualified remote name**. The syntax is:

```
unit_name.simple_identifier_name@link_to_remote_database
```

You can create synonyms for remote schema objects, but you cannot create synonyms for objects declared in PL/SQL subprograms or packages. To create a synonym, use the SQL statement `CREATE SYNONYM`, explained in *Oracle Database SQL Language Reference*.

For information about how PL/SQL resolves ambiguous names, see **PL/SQL Name Resolution**.

**Note:**

You can reference identifiers declared in the packages `STANDARD` and `DBMS_STANDARD` without qualifying them with the package names, unless you have declared a local identifier with the same name (see "Scope and Visibility of Identifiers").
3.5 Scope and Visibility of Identifiers

The **scope** of an identifier is the region of a PL/SQL unit from which you can reference the identifier. The **visibility** of an identifier is the region of a PL/SQL unit from which you can reference the identifier without qualifying it. An identifier is **local** to the PL/SQL unit that declares it. If that unit has subunits, the identifier is **global** to them.

If a subunit redeclares a global identifier, then inside the subunit, both identifiers are in scope, but only the local identifier is visible. To reference the global identifier, the subunit must qualify it with the name of the unit that declared it. If that unit has no name, then the subunit cannot reference the global identifier.

A PL/SQL unit cannot reference identifiers declared in other units at the same level, because those identifiers are neither local nor global to the block.

You cannot declare the same identifier twice in the same PL/SQL unit. If you do, an error occurs when you reference the duplicate identifier.

You can declare the same identifier in two different units. The two objects represented by the identifier are distinct. Changing one does not affect the other.

In the same scope, give labels and subprograms unique names to avoid confusion and unexpected results.

**Examples**

**Example 3-17  Scope and Visibility of Identifiers**

This example shows the scope and visibility of several identifiers. The first sub-block redeclares the global identifier `a`. To reference the global variable `a`, the first sub-block would have to qualify it with the name of the outer block—but the outer block has no name. Therefore, the first sub-block cannot reference the global variable `a`; it can reference only its local variable `a`. Because the sub-blocks are at the same level, the first sub-block cannot reference `d`, and the second sub-block cannot reference `c`.

```plsql
-- Outer block:
DECLARE
  a CHAR;  -- Scope of a (CHAR) begins
  b REAL;  -- Scope of b begins
BEGIN
  -- Visible: a (CHAR), b

  -- First sub-block:
  DECLARE
    a INTEGER; -- Scope of a (INTEGER) begins
    c REAL;    -- Scope of c begins
  BEGIN
    -- Visible: a (INTEGER), b, c
    NULL;
  END;  -- Scopes of a (INTEGER) and c end

  -- Second sub-block:
  DECLARE
    d REAL;  -- Scope of d begins
  BEGIN
    -- Visible: a (CHAR), b, d
    NULL;
  END;  -- Scope of d ends
```


Example 3-18  Qualifying Redeclared Global Identifier with Block Label

This example labels the outer block with the name outer. Therefore, after the sub-block redeclares the global variable birthdate, it can reference that global variable by qualifying its name with the block label. The sub-block can also reference its local variable birthdate, by its simple name.

```
<<outer>>  -- label
DECLARE
  birthdate DATE := TO_DATE('09-AUG-70', 'DD-MON-YY');
BEGIN
  DECLARE
    birthdate DATE := TO_DATE('29-SEP-70', 'DD-MON-YY');
    BEGIN
      IF birthdate = outer.birthdate THEN
        DBMS_OUTPUT.PUT_LINE ('Same Birthday');
      ELSE
        DBMS_OUTPUT.PUT_LINE ('Different Birthday');
      END IF;
    END;
END;
/
```

Result:

Different Birthday

Example 3-19  Qualifying Identifier with Subprogram Name

In this example, the procedure check_credit declares a variable, rating, and a function, check_rating. The function redeclares the variable. Then the function references the global variable by qualifying it with the procedure name.

```
CREATE OR REPLACE PROCEDURE check_credit (credit_limit NUMBER) AS
  rating NUMBER := 3;
FUNCTION check_rating RETURN BOOLEAN IS
  rating NUMBER := 1;
  over_limit BOOLEAN;
BEGIN
  IF check_credit.rating <= credit_limit THEN  -- reference global variable
    over_limit := FALSE;
  ELSE
    over_limit := TRUE;
    rating := credit_limit;                    -- reference local variable
  END IF;
  RETURN over_limit;
END check_rating;
BEGIN
  IF check_rating THEN
    DBMS_OUTPUT.PUT_LINE
    ('Credit rating over limit (' || TO_CHAR(credit_limit) || '). ' ||
      'Rating: ' || TO_CHAR(rating));
  ELSE
    DBMS_OUTPUT.PUT_LINE
```
('Credit rating OK. ' || 'Rating: ' || TO_CHAR(rating));
END IF;
END;
/
BEGIN
  check_credit(1);
END;
/

Result:
Credit rating over limit (1). Rating: 3

Example 3-20  Duplicate Identifiers in Same Scope
You cannot declare the same identifier twice in the same PL/SQL unit. If you do, an error occurs when you reference the duplicate identifier, as this example shows.

DECLARE
  id  BOOLEAN;
  id  VARCHAR2(5);  -- duplicate identifier
BEGIN
  id := FALSE;
END;
/

Result:
  id := FALSE;
  *
ERROR at line 5:
ORA-06550: line 5, column 3:
PLS-00371: at most one declaration for 'ID' is permitted
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored

Example 3-21  Declaring Same Identifier in Different Units
You can declare the same identifier in two different units. The two objects represented by the identifier are distinct. Changing one does not affect the other, as this example shows. In the same scope, give labels and subprograms unique names to avoid confusion and unexpected results.

DECLARE
  PROCEDURE p
  IS
    x  VARCHAR2(1);
  BEGIN
    x := 'a';  -- Assign the value 'a' to x
    DBMS_OUTPUT.PUT_LINE('In procedure p, x = ' || x);
  END;

  PROCEDURE q
  IS
    x  VARCHAR2(1);
  BEGIN
    x := 'b';  -- Assign the value 'b' to x
    DBMS_OUTPUT.PUT_LINE('In procedure q, x = ' || x);
  END;
BEGIN
  p;
  q;
END;
/

Result:
In procedure p, x = a
In procedure q, x = b

Example 3-22 Label and Subprogram with Same Name in Same Scope

In this example, echo is the name of both a block and a subprogram. Both the block and the subprogram declare a variable named x. In the subprogram, echo.x refers to the local variable x, not to the global variable x.

<<echo>>
DECLARE
  x NUMBER := 5;

PROCEDURE echo AS
  x NUMBER := 0;
BEGIN
  DBMS_OUTPUT.PUT_LINE('x = ' || x);
  DBMS_OUTPUT.PUT_LINE('echo.x = ' || echo.x);
END;
BEGIN
  echo;
END;
/

Result:
  x = 0
echo.x = 0

Example 3-23 Block with Multiple and Duplicate Labels

This example has two labels for the outer block, compute_ratio and another_label. The second label appears again in the inner block. In the inner block, another_label.denominator refers to the local variable denominator, not to the global variable denominator, which results in the error ZERO_DIVIDE.

<<compute_ratio>>
<<another_label>>
DECLARE
  numerator NUMBER := 22;
  denominator NUMBER := 7;
BEGIN
  <<another_label>>
  DECLARE
    denominator NUMBER := 0;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Ratio with compute_ratio.denominator = ');
    DBMS_OUTPUT.PUT_LINE(numerator/compute_ratio.denominator);
    DBMS_OUTPUT.PUT_LINE('Ratio with another_label.denominator = ');
  END;
END;
BEGIN
  another_label;
END;
/
3.6 Assigning Values to Variables

After declaring a variable, you can assign a value to it in these ways:

- Use the assignment statement to assign it the value of an expression.
- Use the SELECT INTO or FETCH statement to assign it a value from a table.
- Pass it to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram.

The variable and the value must have compatible data types. One data type is **compatible** with another data type if it can be implicitly converted to that type. For information about implicit data conversion, see *Oracle Database SQL Language Reference*.

**Topics**

- Assigning Values to Variables with the Assignment Statement
- Assigning Values to Variables with the SELECT INTO Statement
- Assigning Values to Variables as Parameters of a Subprogram
- Assigning Values to BOOLEAN Variables

**See Also:**

- "Assigning Values to Collection Variables"
- "Assigning Values to Record Variables"
- "FETCH Statement"

3.6.1 Assigning Values to Variables with the Assignment Statement

To assign the value of an expression to a variable, use this form of the assignment statement:
variable_name := expression;

For the complete syntax of the assignment statement, see "Assignment Statement".

For the syntax of an expression, see "Expression".

Example 3-24 Assigning Values to Variables with Assignment Statement

This example declares several variables (specifying initial values for some) and then uses assignment statements to assign the values of expressions to them.

DECLARE  -- You can assign initial values here
wages          NUMBER;
hours_worked   NUMBER := 40;
hourly_salary  NUMBER := 22.50;
bonus          NUMBER := 150;
country        VARCHAR2(128);
counter        NUMBER := 0;
done           BOOLEAN;
valid_id       BOOLEAN;
emp_rec1       employees%ROWTYPE;
emp_rec2       employees%ROWTYPE;
TYPE commissions IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
comm_tab       commissions;
BEGIN  -- You can assign values here too
wages := (hours_worked * hourly_salary) + bonus;
country := 'France';
country := UPPER('Canada');
done := (counter > 100);
valid_id := TRUE;
emp_rec1.first_name := 'Antonio';
emp_rec1.last_name := 'Ortiz';
emp_rec1 := emp_rec2;
comm_tab(5) := 20000 * 0.15;
END;
/

3.6.2 Assigning Values to Variables with the SELECT INTO Statement

A simple form of the SELECT INTO statement is:

SELECT select_item [, select_item ]...
INTO variable_name [, variable_name ]...
FROM table_name;

For each select_item, there must be a corresponding, type-compatible variable_name. Because SQL does not have a BOOLEAN type, variable_name cannot be a BOOLEAN variable.

For the complete syntax of the SELECT INTO statement, see "SELECT INTO Statement".

Example 3-25 Assigning Value to Variable with SELECT INTO Statement

This example uses a SELECT INTO statement to assign to the variable bonus the value that is 10% of the salary of the employee whose employee_id is 100.

DECLARE
  bonus   NUMBER(8,2);
BEGIN

```
SELECT salary * 0.10 INTO bonus
FROM employees
WHERE employee_id = 100;
END;

DBMS_OUTPUT.PUT_LINE('bonus = ' || TO_CHAR(bonus));
/

Result:
bonus = 2400
```

### 3.6.3 Assigning Values to Variables as Parameters of a Subprogram

If you pass a variable to a subprogram as an OUT or IN OUT parameter, and the subprogram assigns a value to the parameter, the variable retains that value after the subprogram finishes running. For more information, see "Subprogram Parameters".

**Example 3-26 Assigning Value to Variable as IN OUT Subprogram Parameter**

This example passes the variable `new_sal` to the procedure `adjust_salary`. The procedure assigns a value to the corresponding formal parameter, `sal`. Because `sal` is an IN OUT parameter, the variable `new_sal` retains the assigned value after the procedure finishes running.

```sql
DECLARE
  emp_salary  NUMBER(8,2);
PROCEDURE adjust_salary (emp        NUMBER, sal IN OUT NUMBER, adjustment NUMBER) IS
BEGIN
  sal := sal + adjustment;
END;
BEGIN
  SELECT salary INTO emp_salary
  FROM employees
  WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE('Before invoking procedure, emp_salary: ' || emp_salary);
  adjust_salary(100, emp_salary, 1000);
  DBMS_OUTPUT.PUT_LINE('After invoking procedure, emp_salary: ' || emp_salary);
END;
/

Result:
Before invoking procedure, emp_salary: 24000
After invoking procedure, emp_salary: 25000
```
3.6.4 Assigning Values to BOOLEAN Variables

The only values that you can assign to a BOOLEAN variable are TRUE, FALSE, and NULL.

For more information about the BOOLEAN data type, see "BOOLEAN Data Type".

Example 3-27 Assigning Value to BOOLEAN Variable

This example initializes the BOOLEAN variable done to NULL by default, assigns it the literal value FALSE, compares it to the literal value TRUE, and assigns it the value of a BOOLEAN expression.

```sql
DECLARE
    done BOOLEAN; -- Initial value is NULL by default
    counter NUMBER := 0;
BEGIN
    done := FALSE; -- Assign literal value
    WHILE done != TRUE -- Compare to literal value
        LOOP
            counter := counter + 1;
            done := (counter > 500); -- Assign value of BOOLEAN expression
        END LOOP;
END;
/
```

3.7 Expressions

An expression is a combination of one or more values, operators, and SQL functions that evaluates to a value.

An expression always returns a single value. The simplest expressions, in order of increasing complexity, are:

1. A single constant or variable (for example, a)
2. A unary operator and its single operand (for example, −a)
3. A binary operator and its two operands (for example, a+b)

An operand can be a variable, constant, literal, operator, function invocation, or placeholder—or another expression. Therefore, expressions can be arbitrarily complex. For expression syntax, see Expression.

The data types of the operands determine the data type of the expression. Every time the expression is evaluated, a single value of that data type results. The data type of that result is the data type of the expression.

Topics

- Concatenation Operator
- Operator Precedence
- Logical Operators
- Short-Circuit Evaluation
- Comparison Operators
- BOOLEAN Expressions
3.7.1 Concatenation Operator

The concatenation operator (||) appends one string operand to another.

The concatenation operator ignores null operands.

For more information about the syntax of the concatenation operator, see "character_expression ::=".

**Example 3-28  Concatenation Operator**

```sql
DECLARE
  x VARCHAR2(4) := 'suit';
  y VARCHAR2(4) := 'case';
BEGIN
  DBMS_OUTPUT.PUT_LINE (x || y);
END;
/
```

**Result:**
suitcase

**Example 3-29  Concatenation Operator with NULL Operands**

The concatenation operator ignores null operands, as this example shows.

```sql
BEGIN
  DBMS_OUTPUT.PUT_LINE ('apple' || NULL || NULL || 'sauce');
END;
/
```

**Result:**
applesauce

3.7.2 Operator Precedence

An operation is either a unary operator and its single operand or a binary operator and its two operands. The operations in an expression are evaluated in order of operator precedence.

Table 3-3 shows operator precedence from highest to lowest. Operators with equal precedence are evaluated in no particular order.

**Table 3-3  Operator Precedence**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>+, -</td>
<td>identity, negation</td>
</tr>
<tr>
<td>*, /</td>
<td>multiplication, division</td>
</tr>
</tbody>
</table>
Table 3-3  (Cont.) Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -,</td>
<td></td>
</tr>
<tr>
<td>=, &lt;, &gt;, &lt;=, &gt;=, !, ~, ^=, ^=, IS NULL, LIKE, BETWEEN, IN</td>
<td>comparison</td>
</tr>
<tr>
<td>NOT</td>
<td>negation</td>
</tr>
<tr>
<td>AND</td>
<td>conjunction</td>
</tr>
<tr>
<td>OR</td>
<td>inclusion</td>
</tr>
</tbody>
</table>

To control the order of evaluation, enclose operations in parentheses, as in Example 3-30.

When parentheses are nested, the most deeply nested operations are evaluated first.

You can also use parentheses to improve readability where the parentheses do not affect evaluation order.

**Example 3-30  Controlling Evaluation Order with Parentheses**

```sql
DECLARE
  a INTEGER := 1+2**2;
  b INTEGER := (1+2)**2;
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
  DBMS_OUTPUT.PUT_LINE('b = ' || TO_CHAR(b));
END;
/
```

Result:

a = 5
b = 9

**Example 3-31  Expression with Nested Parentheses**

In this example, the operations (1+2) and (3+4) are evaluated first, producing the values 3 and 7, respectively. Next, the operation 3*7 is evaluated, producing the result 21. Finally, the operation 21/7 is evaluated, producing the final value 3.

```sql
DECLARE
  a INTEGER := (1+2)*(3+4))/7;
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
END;
/
```

Result:

a = 3

**Example 3-32  Improving Readability with Parentheses**

In this example, the parentheses do not affect the evaluation order. They only improve readability.
DECLARE
a INTEGER := 2**2*3**2;
b INTEGER := (2**2) * (3**2);
BEGIN
    DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
    DBMS_OUTPUT.PUT_LINE('b = ' || TO_CHAR(b));
END;
/

Result:
a = 36
b = 36

Example 3-33 Operator Precedence

This example shows the effect of operator precedence and parentheses in several more complex expressions.

DECLARE
    salary NUMBER := 60000;
    commission NUMBER := 0.10;
BEGIN
    -- Division has higher precedence than addition:
    DBMS_OUTPUT.PUT_LINE('5 + 12 / 4 = ' || TO_CHAR(5 + 12 / 4));
    DBMS_OUTPUT.PUT_LINE('12 / 4 + 5 = ' || TO_CHAR(12 / 4 + 5));

    -- Parentheses override default operator precedence:
    DBMS_OUTPUT.PUT_LINE('8 + 6 / 2 = ' || TO_CHAR(8 + 6 / 2));
    DBMS_OUTPUT.PUT_LINE('(8 + 6) / 2 = ' || TO_CHAR((8 + 6) / 2));

    -- Most deeply nested operation is evaluated first:
    DBMS_OUTPUT.PUT_LINE('100 + (20 / 5 + (7 - 3)) = ' ||
                        TO_CHAR(100 + (20 / 5 + (7 - 3))));

    -- Parentheses, even when unnecessary, improve readability:
    DBMS_OUTPUT.PUT_LINE('salary * 0.05 + (commission * 0.25) = ' ||
                         TO_CHAR((salary * 0.05) + (commission * 0.25)));
    DBMS_OUTPUT.PUT_LINE('salary * 0.05 + commission * 0.25 = ' ||
                         TO_CHAR(salary * 0.05 + commission * 0.25));
END;
/

Result:

5 + 12 / 4 = 8
12 / 4 + 5 = 8
8 + 6 / 2 = 11
(8 + 6) / 2 = 7
100 + (20 / 5 + (7 - 3)) = 108
(salary * 0.05) + (commission * 0.25) = 3000.025
salary * 0.05 + commission * 0.25 = 3000.025
3.7.3 Logical Operators

The logical operators **AND**, **OR**, and **NOT** follow a tri-state logic.

**AND** and **OR** are binary operators; **NOT** is a unary operator.

**Table 3-4**  Logical Truth Table

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x AND y</th>
<th>x OR y</th>
<th>NOT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
<td>TRUE</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>FALSE</td>
<td>FALSE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**AND** returns **TRUE** if and only if both operands are **TRUE**.

**OR** returns **TRUE** if either operand is **TRUE**.

**NOT** returns the opposite of its operand, unless the operand is **NULL**. **NOT** **NULL** returns **NULL**, because **NULL** is an indeterminate value.

**Example 3-34**  Procedure Prints BOOLEAN Variable

This example creates a procedure, `print_boolean`, that prints the value of a **BOOLEAN** variable. The procedure uses the "**IS [NOT] NULL Operator**". Several examples in this chapter invoke `print_boolean`.

```sql
CREATE OR REPLACE PROCEDURE print_boolean (
  b_name   VARCHAR2,
  b_value  BOOLEAN
) AUTHID DEFINER IS
BEGIN
  IF b_value IS NULL THEN
    DBMS_OUTPUT.PUT_LINE (b_name || ' = NULL');
  ELSIF b_value = TRUE THEN
    DBMS_OUTPUT.PUT_LINE (b_name || ' = TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE (b_name || ' = FALSE');
  END IF;
END;
/
```

**Example 3-35**  AND Operator

As **Table 3-4** and this example show, **AND** returns **TRUE** if and only if both operands are **TRUE**.
DECLARE
  PROCEDURE print_x_and_y (
    x  BOOLEAN,
    y  BOOLEAN
  ) IS
  BEGIN
    print_boolean ('x', x);
    print_boolean ('y', y);
    print_boolean ('x AND y', x AND y);
  END print_x_and_y;
BEGIN
  print_x_and_y (FALSE, FALSE);
  print_x_and_y (TRUE, FALSE);
  print_x_and_y (FALSE, TRUE);
  print_x_and_y (TRUE, TRUE);
  print_x_and_y (TRUE, NULL);
  print_x_and_y (FALSE, NULL);
  print_x_and_y (NULL, TRUE);
  print_x_and_y (NULL, FALSE);
END;
/

Result:

  x = FALSE
  y = FALSE
  x AND y = FALSE
  x = TRUE
  y = FALSE
  x AND y = FALSE
  x = FALSE
  y = TRUE
  x AND y = FALSE
  x = TRUE
  y = TRUE
  x AND y = TRUE
  x = TRUE
  y = NULL
  x AND y = NULL
  x = FALSE
  y = NULL
  x AND y = FALSE
  x = NULL
  y = TRUE
  x AND y = NULL
  x = NULL
  y = FALSE
  x AND y = FALSE

Example 3-36  OR Operator

As Table 3-4 and this example show, OR returns TRUE if either operand is TRUE. (This example invokes the print_boolean procedure from Example 3-34.)

DECLARE
  PROCEDURE print_x_or_y (
    x  BOOLEAN,
    y  BOOLEAN
  ) IS
  BEGIN
    print_boolean ('x', x);
    print_boolean ('y', y);
    print_boolean ('x OR y', x OR y);
  END print_x_or_y;
BEGIN
print_x_or_y (FALSE, FALSE);
print_x_or_y (TRUE, FALSE);
print_x_or_y (FALSE, TRUE);
print_x_or_y (TRUE, TRUE);
print_x_or_y (TRUE, NULL);
print_x_or_y (FALSE, NULL);
print_x_or_y (NULL, TRUE);
print_x_or_y (NULL, FALSE);
END;
/

Result:

x = FALSE
y = FALSE
x OR y = FALSE
x = TRUE
y = FALSE
x OR y = TRUE
x = FALSE
y = TRUE
x OR y = TRUE
x = TRUE
y = TRUE
x OR y = TRUE
x = TRUE
y = NULL
x OR y = TRUE
x = FALSE
y = NULL
x OR y = NULL
x = NULL
y = TRUE
x OR y = TRUE
x = NULL
y = FALSE
x OR y = NULL

Example 3-37  NOT Operator

As Table 3-4 and this example show, NOT returns the opposite of its operand, unless the operand is NULL. NOT NULL returns NULL, because NULL is an indeterminate value.
(This example invokes the print_boolean procedure from Example 3-34.)

DECLARE
PROCEDURE print_not_x (x BOOLEAN)
IS
BEGIN
print_boolean ('x', x);
print_boolean ('NOT x', NOT x);
END print_not_x;
BEGIN
    print_not_x (TRUE);
    print_not_x (FALSE);
    print_not_x (NULL);
END;
/

Result:

x = TRUE
NOT x = FALSE
x = FALSE
NOT x = TRUE
x = NULL
NOT x = NULL

Example 3-38  NULL Value in Unequal Comparison

In this example, you might expect the sequence of statements to run because x and y seem unequal. But, NULL values are indeterminate. Whether x equals y is unknown. Therefore, the IF condition yields NULL and the sequence of statements is bypassed.

DECLARE
    x NUMBER := 5;
    y NUMBER := NULL;
BEGIN
    IF x != y THEN  -- yields NULL, not TRUE
        DBMS_OUTPUT.PUT_LINE('x != y');  -- not run
    ELSIF x = y THEN -- also yields NULL
        DBMS_OUTPUT.PUT_LINE('x = y');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Can''t tell if x and y are equal or not.');
    END IF;
END;
/

Result:

Can't tell if x and y are equal or not.

Example 3-39  NULL Value in Equal Comparison

In this example, you might expect the sequence of statements to run because a and b seem equal. But, again, that is unknown, so the IF condition yields NULL and the sequence of statements is bypassed.

DECLARE
    a NUMBER := NULL;
    b NUMBER := NULL;
BEGIN
    IF a = b THEN  -- yields NULL, not TRUE
        DBMS_OUTPUT.PUT_LINE('a = b');  -- not run
    ELSIF a != b THEN -- yields NULL, not TRUE
        DBMS_OUTPUT.PUT_LINE('a != b');  -- not run
    ELSE
        DBMS_OUTPUT.PUT_LINE('Can''t tell if two NULLs are equal');
    END IF;
END;
Result:

Can't tell if two NULLs are equal

**Example 3-40  NOT NULL Equals NULL**

In this example, the two IF statements appear to be equivalent. However, if either \( x \) or \( y \) is NULL, then the first IF statement assigns the value of \( y \) to high and the second IF statement assigns the value of \( x \) to high.

```sql
DECLARE
    x    INTEGER := 2;
    y    INTEGER := 5;
    high INTEGER;
BEGIN
    IF (x > y)  -- If x or y is NULL, then (x > y) is NULL
        THEN high := x;  -- run if (x > y) is TRUE
        ELSE high := y;  -- run if (x > y) is FALSE or NULL
    END IF;

    IF NOT (x > y)  -- If x or y is NULL, then NOT (x > y) is NULL
        THEN high := y;  -- run if NOT (x > y) is TRUE
        ELSE high := x;  -- run if NOT (x > y) is FALSE or NULL
    END IF;
END;
```

**Example 3-41  Changing Evaluation Order of Logical Operators**

This example invokes the `print_boolean` procedure from Example 3-34 three times. The third and first invocation are logically equivalent—the parentheses in the third invocation only improve readability. The parentheses in the second invocation change the order of operation.

```sql
DECLARE
    x  BOOLEAN := FALSE;
    y  BOOLEAN := FALSE;
BEGIN
    print_boolean ('NOT x AND y', NOT x AND y);
    print_boolean ('NOT (x AND y)', NOT (x AND y));
    print_boolean ('(NOT x) AND y', (NOT x) AND y);
END;
```

Result:

```
NOT x AND y = FALSE
NOT (x AND y) = TRUE
(NOT x) AND y = FALSE
```
3.7.4 Short-Circuit Evaluation

When evaluating a logical expression, PL/SQL uses **short-circuit evaluation**. That is, PL/SQL stops evaluating the expression as soon as it can determine the result.

Therefore, you can write expressions that might otherwise cause errors.

In Example 3-42, short-circuit evaluation prevents the OR expression from causing a divide-by-zero error. When the value of `on_hand` is zero, the value of the left operand is TRUE, so PL/SQL does not evaluate the right operand. If PL/SQL evaluated both operands before applying the OR operator, the right operand would cause a division by zero error.

**Example 3-42  Short-Circuit Evaluation**

```plsql
DECLARE
  on_hand  INTEGER := 0;
  on_order INTEGER := 100;
BEGIN
  -- Does not cause divide-by-zero error;
  -- evaluation stops after first expression
  IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
    DBMS_OUTPUT.PUT_LINE('On hand quantity is zero.');
  END IF;
END;
/
```

Result:

On hand quantity is zero.

3.7.5 Comparison Operators

Comparison operators compare one expression to another. The result is always either TRUE, FALSE, or NULL.

If the value of one expression is NULL, then the result of the comparison is also NULL.

The comparison operators are:

- **IS [NOT] NULL Operator**
- **Relational Operators**
- **LIKE Operator**
- **BETWEEN Operator**
- **IN Operator**
3.7.1 IS [NOT] NULL Operator

The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. The IS NOT NULL operator does the opposite.

Comparisons involving NULL values always yield NULL.

To test whether a value is NULL, use IF value IS NULL, as in these examples:

- Example 3-14, "Variable Initialized to NULL by Default"
- Example 3-34, "Procedure Prints BOOLEAN Variable"
- Example 3-53, "Searched CASE Expression with WHEN ... IS NULL"

3.7.2 Relational Operators

This table summarizes the relational operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>=&gt;</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

Topics

- Arithmetic Comparisons
- BOOLEAN Comparisons
3.7.5.2.1 Arithmetic Comparisons

One number is greater than another if it represents a larger quantity.

Real numbers are stored as approximate values, so Oracle recommends comparing them for equality or inequality.

Example 3-43   Relational Operators in Expressions

This example invokes the print_boolean procedure from Example 3-35 to print the values of expressions that use relational operators to compare arithmetic values.

BEGIN
  print_boolean ('(2 + 2 =  4)', 2 + 2 = 4);
  print_boolean ('(2 + 2 <> 4)', 2 + 2 <> 4);
  print_boolean ('(2 + 2 != 4)', 2 + 2 != 4);
  print_boolean ('(2 + 2 ~= 4)', 2 + 2 ~= 4);
  print_boolean ('(2 + 2 ^= 4)', 2 + 2 ^= 4);
  print_boolean ('(1 < 2)', 1 < 2);
  print_boolean ('(1 > 2)', 1 > 2);
  print_boolean ('(1 <= 2)', 1 <= 2);
  print_boolean ('(1 >= 1)', 1 >= 1);
END;
/

Result:

(2 + 2 =  4) = TRUE
(2 + 2 <> 4) = FALSE
(2 + 2 != 4) = FALSE
(2 + 2 ~= 4) = FALSE
(2 + 2 ^= 4) = FALSE
(1 < 2) = TRUE
(1 > 2) = FALSE
(1 <= 2) = TRUE
(1 >= 1) = TRUE

3.7.5.2.2 BOOLEAN Comparisons

By definition, TRUE is greater than FALSE. Any comparison with NULL returns NULL.

3.7.5.2.3 Character Comparisons

By default, one character is greater than another if its binary value is larger.

For example, this expression is true:

'y' > 'z'

Strings are compared character by character. For example, this expression is true:
'Kathy' > 'Kathryn'

If you set the initialization parameter NLS_COMP=ANSI, string comparisons use the collating sequence identified by the NLS_SORT initialization parameter.

A **collating sequence** is an internal ordering of the character set in which a range of numeric codes represents the individual characters. One character value is greater than another if its internal numeric value is larger. Each language might have different rules about where such characters occur in the collating sequence. For example, an accented letter might be sorted differently depending on the database character set, even though the binary value is the same in each case.

By changing the value of the NLS_SORT parameter, you can perform comparisons that are case-insensitive and accent-insensitive.

A **case-insensitive comparison** treats corresponding uppercase and lowercase letters as the same letter. For example, these expressions are true:

- 'a' = 'A'
- 'Alpha' = 'ALPHA'

To make comparisons case-insensitive, append _CI to the value of the NLS_SORT parameter (for example, BINARY_CI or XGERMAN_CI).

An **accent-insensitive comparison** is case-insensitive, and also treats letters that differ only in accents or punctuation characters as the same letter. For example, these expressions are true:

- 'Cooperate' = 'Co-Operate'
- 'Co-Operate' = 'coöperate'

To make comparisons both case-insensitive and accent-insensitive, append _AI to the value of the NLS_SORT parameter (for example, BINARY_AI or FRENCH_M_AI).

Semantic differences between the CHAR and VARCHAR2 data types affect character comparisons.

For more information, see "Value Comparisons".

### 3.7.5.2.4 Date Comparisons

One date is greater than another if it is more recent.

For example, this expression is true:

- `'01-JAN-91' > '31-DEC-90'`

### 3.7.5.3 LIKE Operator

The **LIKE operator** compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.

Case is significant.

The pattern can include the two **wildcard characters** underscore (_) and percent sign (%).

- Underscore matches exactly one character.
- Percent sign (%) matches zero or more characters.
To search for the percent sign or underscore, define an escape character and put it before the percent sign or underscore.

---

**See Also:**

- *Oracle Database SQL Language Reference* for more information about LIKE
- *Oracle Database SQL Language Reference* for information about REGEXP_LIKE, which is similar to LIKE

---

**Example 3-44  LIKE Operator in Expression**

The string 'Johnson' matches the pattern 'J%s_n' but not 'J%S_N', as this example shows.

```sql
DECLARE
    PROCEDURE compare (value VARCHAR2, pattern VARCHAR2) IS
    BEGIN
        IF value LIKE pattern THEN
            DBMS_OUTPUT.PUT_LINE ('TRUE');
        ELSE
            DBMS_OUTPUT.PUT_LINE ('FALSE');
        END IF;
    END;
BEGIN
    compare('Johnson', 'J%s_n');
    compare('Johnson', 'J%S_N');
END; /
```

Result:

TRUE
FALSE

---

**Example 3-45  Escape Character in Pattern**

This example uses the backslash as the escape character, so that the percent sign in the string does not act as a wildcard.

```sql
DECLARE
    PROCEDURE half_off (sale_sign VARCHAR2) IS
    BEGIN
        IF sale_sign LIKE '50\% off!' ESCAPE '\' THEN
            DBMS_OUTPUT.PUT_LINE ('TRUE');
        ELSE
            DBMS_OUTPUT.PUT_LINE ('FALSE');
        END IF;
    END;
BEGIN
    half_off('Going out of business!');
    half_off('50% off!');
END; /
```
3.7.5.4 BETWEEN Operator

The BETWEEN operator tests whether a value lies in a specified range.

The value of the expression $x$ BETWEEN $a$ AND $b$ is defined to be the same as the value of the expression $(x\geq a)$ AND $(x\leq b)$. The expression $x$ will only be evaluated once.

See Also:

Oracle Database SQL Language Reference for more information about BETWEEN

Example 3-46  BETWEEN Operator in Expressions

This example invokes the print_boolean procedure from Example 3-34 to print the values of expressions that include the BETWEEN operator.

```
BEGIN
    print_boolean ('2 BETWEEN 1 AND 3', 2 BETWEEN 1 AND 3);
    print_boolean ('2 BETWEEN 2 AND 3', 2 BETWEEN 2 AND 3);
    print_boolean ('2 BETWEEN 1 AND 2', 2 BETWEEN 1 AND 2);
    print_boolean ('2 BETWEEN 3 AND 4', 2 BETWEEN 3 AND 4);
END;
/

Result:
2 BETWEEN 1 AND 3 = TRUE
2 BETWEEN 2 AND 3 = TRUE
2 BETWEEN 1 AND 2 = TRUE
2 BETWEEN 3 AND 4 = FALSE
```

3.7.5.5 IN Operator

The IN operator tests set membership.

$x$ IN (set) returns TRUE only if $x$ equals a member of set.

See Also:

Oracle Database SQL Language Reference for more information about IN
Example 3-47  IN Operator in Expressions

This example invokes the print_boolean procedure from Example 3-34 to print the values of expressions that include the IN operator.

```
DECLARE
  letter VARCHAR2(1) := 'm';
BEGIN
  print_boolean ('letter IN (''a'', ''b'', ''c'')', letter IN ('a', 'b', 'c'));
  print_boolean ('letter IN (''z'', ''m'', ''y'', ''p'')', letter IN ('z', 'm', 'y', 'p'));
END;
/
```

Result:

letter IN ('a', 'b', 'c') = FALSE
letter IN ('z', 'm', 'y', 'p') = TRUE

Example 3-48  IN Operator with Sets with NULL Values

This example shows what happens when set includes a NULL value. This invokes the print_boolean procedure from Example 3-34.

```
DECLARE
  a INTEGER; -- Initialized to NULL by default
  b INTEGER := 10;
  c INTEGER := 100;
BEGIN
  print_boolean ('100 IN (a, b, c)', 100 IN (a, b, c));
  print_boolean ('100 NOT IN (a, b, c)', 100 NOT IN (a, b, c));
  print_boolean ('100 IN (a, b)', 100 IN (a, b));
  print_boolean ('100 NOT IN (a, b)', 100 NOT IN (a, b));
  print_boolean ('a IN (a, b)', a IN (a, b));
  print_boolean ('a NOT IN (a, b)', a NOT IN (a, b));
END;
/
```

Result:

100 IN (a, b, c) = TRUE
100 NOT IN (a, b, c) = FALSE
100 IN (a, b) = NULL
100 NOT IN (a, b) = NULL
a IN (a, b) = NULL
a NOT IN (a, b) = NULL

3.7.6 BOOLEAN Expressions

A **BOOLEAN expression** is an expression that returns a **BOOLEAN value**—TRUE, FALSE, or NULL.
The simplest BOOLEAN expression is a BOOLEAN literal, constant, or variable. The following are also BOOLEAN expressions:

- \( \text{NOT boolean_expression} \)
- \( \text{boolean_expression relational_operator boolean_expression} \)
- \( \text{boolean_expression \{ AND | OR \} boolean_expression} \)

For a list of relational operators, see Table 3-5. For the complete syntax of a BOOLEAN expression, see "boolean_expression ::=".

Typically, you use BOOLEAN expressions as conditions in control statements (explained in PL/SQL Control Statements) and in WHERE clauses of DML statements.

You can use a BOOLEAN variable itself as a condition; you need not compare it to the value TRUE or FALSE.

**Example 3-49  Equivalent BOOLEAN Expressions**

In this example, the conditions in the loops are equivalent.

```sql
DECLARE
  done BOOLEAN;
BEGIN
  -- These WHILE loops are equivalent
  done := FALSE;
  WHILE done = FALSE LOOP
    done := TRUE;
    END LOOP;
  done := FALSE;
  WHILE NOT (done = TRUE) LOOP
    done := TRUE;
    END LOOP;
  done := FALSE;
  WHILE NOT done LOOP
    done := TRUE;
    END LOOP;
END;
```

3.7.7 CASE Expressions

**Topics**
- Simple CASE Expression
- Searched CASE Expression

3.7.7.1 Simple CASE Expression

For this explanation, assume that a simple CASE expression has this syntax:

```
CASE selector
  WHEN selector_value_1 THEN result_1
  WHEN selector_value_2 THEN result_2
```

The selector is an expression (typically a single variable). Each selector_value and each result can be either a literal or an expression. At least one result must not be the literal NULL.

The simple CASE expression returns the first result for which selector_value matches selector. Remaining expressions are not evaluated. If no selector_value matches selector, the CASE expression returns else_result if it exists and NULL otherwise.

See Also:
"simple_case_expression ::=" for the complete syntax

Example 3-50  Simple CASE Expression

This example assigns the value of a simple CASE expression to the variable appraisal. The selector is grade.

```sql
DECLARE
    grade CHAR(1) := 'B';
    appraisal VARCHAR2(20);
BEGIN
    appraisal :=
        CASE grade
            WHEN 'A' THEN 'Excellent'
            WHEN 'B' THEN 'Very Good'
            WHEN 'C' THEN 'Good'
            WHEN 'D' THEN 'Fair'
            WHEN 'F' THEN 'Poor'
            ELSE 'No such grade'
        END;
    DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

Result:

Grade B is Very Good

Example 3-51  Simple CASE Expression with WHEN NULL

If selector has the value NULL, it cannot be matched by WHEN NULL, as this example shows.

Instead, use a searched CASE expression with WHEN boolean_expression IS NULL, as in Example 3-53.

```sql
DECLARE
    grade CHAR(1); -- NULL by default
    appraisal VARCHAR2(20);
```
BEGIN
  appraisal :=
  CASE grade
    WHEN NULL THEN 'No grade assigned'
    WHEN 'A' THEN 'Excellent'
    WHEN 'B' THEN 'Very Good'
    WHEN 'C' THEN 'Good'
    WHEN 'D' THEN 'Fair'
    WHEN 'F' THEN 'Poor'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/

Result:
Grade is No such grade

3.7.7.2 Searched CASE Expression

For this explanation, assume that a searched CASE expression has this syntax:

CASE
  WHEN boolean_expression_1 THEN result_1
  WHEN boolean_expression_2 THEN result_2
  ...
  WHEN boolean_expression_n THEN result_n
  [ ELSE
    else_result ]
END

The searched CASE expression returns the first result for which boolean_expression is TRUE. Remaining expressions are not evaluated. If no boolean_expression is TRUE, the CASE expression returns else_result if it exists and NULL otherwise.

See Also:
"searched_case_expression ::=" for the complete syntax

Example 3-52 Searched CASE Expression

This example assigns the value of a searched CASE expression to the variable appraisal.

DECLARE
  grade CHAR(1) := 'B';
  appraisal VARCHAR2(120);
  id NUMBER := 8429862;
  attendance NUMBER := 150;
  min_days CONSTANT NUMBER := 200;
  
  FUNCTION attends_this_school (id NUMBER)
    RETURN BOOLEAN IS
    RETURN TRUE;
END;
BEGIN
  appraisal :=
  CASE
    WHEN attends_this_school(id) = FALSE
      THEN 'Student not enrolled'
    WHEN grade = 'F' OR attendance < min_days
      THEN 'Poor (poor performance or bad attendance)'
    WHEN grade = 'A' THEN 'Excellent'
    WHEN grade = 'B' THEN 'Very Good'
    WHEN grade = 'C' THEN 'Good'
    WHEN grade = 'D' THEN 'Fair'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE ('Result for student ' || id || ' is ' || appraisal);
END;
/

Result:

Result for student 8429862 is Poor (poor performance or bad attendance)

Example 3-53  Searched CASE Expression with WHEN ... IS NULL

This example uses a searched CASE expression to solve the problem in Example 3-51.

DECLARE
  grade CHAR(1); -- NULL by default
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
  CASE
    WHEN grade IS NULL THEN 'No grade assigned'
    WHEN grade = 'A' THEN 'Excellent'
    WHEN grade = 'B' THEN 'Very Good'
    WHEN grade = 'C' THEN 'Good'
    WHEN grade = 'D' THEN 'Fair'
    WHEN grade = 'F' THEN 'Poor'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/

Result:

Grade  is No grade assigned

3.7.8 SQL Functions in PL/SQL Expressions

In PL/SQL expressions, you can use all SQL functions except:

- Aggregate functions (such as AVG and COUNT)
- Aggregate function JSON_ARRAYAGG
- Aggregate function JSON_DATAAGUIDE
- Aggregate function JSON_MERGEPATCH
• **Aggregate function** `JSON_OBJECTAGG`
• `JSON_TABLE`
• `JSON_TRANSFORM`
• **JSON condition** `JSON_TEXTCONTAINS`
• Analytic functions (such as `LAG` and `RATIO_TO_REPORT`)
• Conversion function `BIN_TO_NUM`
• Data mining functions (such as `CLUSTER_ID` and `FEATURE_VALUE`)
• Encoding and decoding functions (such as `DECODE` and `DUMP`)
• Model functions (such as `ITERATION_NUMBER` and `PREVIOUS`)
• Object reference functions (such as `REF` and `VALUE`)
• XML functions
• These collation SQL operators and functions:
  – `COLLATE` operator
  – `COLLATION` function
  – `NLS_COLLATION_ID` function
  – `NLS_COLLATION_NAME` function
• These miscellaneous functions:
  – `CUBE_TABLE`
  – `DATAOBJ_TO_PARTITION`
  – `LNNVL`
  – `NVL2`
  – `SYS_CONNECT_BY_PATH`
  – `SYS_TYPEID`
  – `WIDTH_BUCKET`

PL/SQL supports an overload of `BITAND` for which the arguments and result are `BINARY_INTEGER`.

When used in a PL/SQL expression, the `RAWTOHEX` function accepts an argument of data type `RAW` and returns a `VARCHAR2` value with the hexadecimal representation of bytes that comprise the value of the argument. Arguments of types other than `RAW` can be specified only if they can be implicitly converted to `RAW`. This conversion is possible for `CHAR`, `VARCHAR2`, and `LONG` values that are valid arguments of the `HEXTORAW` function, and for `LONG RAW` and `BLOB` values of up to 16380 bytes.

### 3.7.9 Static Expressions

A **static expression** is an expression whose value can be determined at compile time—that is, it does not include character comparisons, variables, or function invocations. Static expressions are the only expressions that can appear in conditional compilation directives.
Definition of Static Expression

- An expression is static if it is the NULL literal.
- An expression is static if it is a character, numeric, or boolean literal.
- An expression is static if it is a reference to a static constant.
- An expression is static if it is a reference to a conditional compilation variable begun with $$.
- An expression is static if it is an operator is allowed in static expressions, if all of its operands are static, and if the operator does not raise an exception when it is evaluated on those operands.

Table 3-6 Operators Allowed in Static Expressions

<table>
<thead>
<tr>
<th>Operators</th>
<th>Operators Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Expression delimiter</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td>*, /, +, -</td>
<td>Arithmetic operators for multiplication, division, addition or positive, subtraction or negative</td>
</tr>
<tr>
<td>=, !=, &lt;, &lt;=, &gt;, &gt;=</td>
<td>Comparison operators</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical operator</td>
</tr>
<tr>
<td>XOR</td>
<td>Binary operator</td>
</tr>
</tbody>
</table>

This list shows functions allowed in static expressions.

- ABS
- ACOS
- ASCII
- ASCIISTR
- ASIN
- ATAN
- ATAN2
- BITAND
- CEIL
- CHR
- COMPOSE
- CONVERT
- COS
- COSH
- DECOMPOSE
- EXP
- FLOOR
- HEXTORAW
- INSTR
- INSTRB
- INSTRC
- INSTR2
- INSTR4
- IS [NOT] INFINITE
- IS [NOT] NAN
- LENGTH
- LENGTH2
- LENGTH4
- LENGTHB
- LENGTHC
- LN
- LOG
- LOWER
- LPAD
- LTRIM
- MOD
- NVL
- POWER
- RAWTOHEX
- REM
- REMAINDER
- REPLACE
- ROUND
- RPAD
- RTRIM
- SIGN
- SIN
- SINH
- SQRT
- SUBSTR
- SUBSTR2
- SUBSTR4
- SUBSTRB
• SUBSTRC
• TAN
• TANH
• TO_BINARY_DOUBLE
• TO_BINARY_FLOAT
• TO_CHAR
• TO_NUMBER
• TRIM
• TRUNC
• UPPER

Static expressions can be used in the following subtype declarations:

• Length of string types (VARCHAR2, NCHAR, CHAR, NVARCHAR2, RAW, and the ANSI equivalents)
• Scale and precision of NUMBER types and subtypes such as FLOAT
• Interval type precision (year, month, second)
• Time and Timestamp precision
• VARRAY bounds
• Bounds of ranges in type declarations

In each case, the resulting type of the static expression must be the same as the declared item subtype and must be in the correct range for the context.

Topics

• PLS_INTEGER Static Expressions
• BOOLEAN Static Expressions
• VARCHAR2 Static Expressions
• Static Constants

See Also:
"Expressions" for general information about expressions

3.7.9.1 PLS_INTEGER Static Expressions

PLS_INTEGER static expressions are:

• PLS_INTEGER literals
  For information about literals, see "Literals".
• PLS_INTEGER static constants
  For information about static constants, see "Static Constants".
3.7.9.2 BOOLEAN Static Expressions

BOOLEAN static expressions are:

- **BOOLEAN literals** (TRUE, FALSE, or NULL)
- **BOOLEAN static constants**
  
  For information about static constants, see "Static Constants".

- Where \( x \) and \( y \) are PLS_INTEGER static expressions:
  - \( x > y \)
  - \( x < y \)
  - \( x >= y \)
  - \( x <= y \)
  - \( x = y \)
  - \( x <> y \)

  For information about PLS_INTEGER static expressions, see "PLS_INTEGER Static Expressions".

- Where \( x \) and \( y \) are BOOLEAN expressions:
  - \( \text{NOT} \ y \)
  - \( x \text{ AND} y \)
  - \( x \text{ OR} y \)
  - \( x > y \)
  - \( x >= y \)
  - \( x = y \)
  - \( x <= y \)
  - \( x <> y \)

  For information about BOOLEAN expressions, see "BOOLEAN Expressions".

- Where \( x \) is a static expression:
  - \( x \text{ IS} \text{ NULL} \)
  - \( x \text{ IS} \text{ NOT} \text{ NULL} \)

  For information about static expressions, see "Static Expressions".

See Also:

"PLS_INTEGER and BINARY_INTEGER Data Types" for information about the PLS_INTEGER data type
3.7.9.3 VARCHAR2 Static Expressions

VARCHAR2 static expressions are:

- String literal with maximum size of 32,767 bytes
  For information about literals, see "Literals".
- NULL
- \texttt{TO\_CHAR(x)}, where \(x\) is a \texttt{PLS\_INTEGER} static expression
  For information about the \texttt{TO\_CHAR} function, see \textit{Oracle Database SQL Language Reference}.
- \texttt{TO\_CHAR(x, f, n)} where \(x\) is a \texttt{PLS\_INTEGER} static expression and \(f\) and \(n\) are \texttt{VARCHAR2} static expressions
  For information about the \texttt{TO\_CHAR} function, see \textit{Oracle Database SQL Language Reference}.
- \(x \mid\mid y\) where \(x\) and \(y\) are \texttt{VARCHAR2} or \texttt{PLS\_INTEGER} static expressions
  For information about \texttt{PLS\_INTEGER} static expressions, see "\texttt{PLS\_INTEGER} Static Expressions".

3.7.9.4 Static Constants

A \textbf{static constant} is declared in a package specification with this syntax:

\begin{verbatim}
constant_name CONSTANT data_type := static_expression;
\end{verbatim}

The type of \texttt{static_expression} must be the same as \texttt{data_type} (either \texttt{BOOLEAN} or \texttt{PLS\_INTEGER}).

The static constant must always be referenced as \texttt{package\_name.constant\_name}, even in the body of the \texttt{package\_name} package.

If you use \texttt{constant\_name} in the \texttt{BOOLEAN} expression in a conditional compilation directive in a PL/SQL unit, then the PL/SQL unit depends on the package \texttt{package\_name}. If you alter the package specification, the dependent PL/SQL unit might become invalid and need recompilation (for information about the invalidation of dependent objects, see \textit{Oracle Database Development Guide}).

If you use a package with static constants to control conditional compilation in multiple PL/SQL units, Oracle recommends that you create only the package specification, and
dedicate it exclusively to controlling conditional compilation. This practice minimizes invalidations caused by altering the package specification.

To control conditional compilation in a single PL/SQL unit, you can set flags in the `PLSQL_CCFLAGS` compilation parameter. For information about this parameter, see "Assigning Values to Inquiry Directives" and Oracle Database Reference.

**See Also:**
- "Declaring Constants" for general information about declaring constants
- PL/SQL Packages for more information about packages
- Oracle Database Development Guide for more information about schema object dependencies

**Example 3-54  Static Constants**

In this example, the package `my_debug` defines the static constants `debug` and `trace` to control debugging and tracing in multiple PL/SQL units. The procedure `my_proc1` uses only `debug`, and the procedure `my_proc2` uses only `trace`, but both procedures depend on the package. However, the recompiled code might not be different. For example, if you only change the value of `debug` to `FALSE` and then recompile the two procedures, the compiled code for `my_proc1` changes, but the compiled code for `my_proc2` does not.

```sql
CREATE PACKAGE my_debug IS
    debug CONSTANT BOOLEAN := TRUE;
    trace CONSTANT BOOLEAN := TRUE;
END my_debug;
/

CREATE PROCEDURE my_proc1 AUTHID DEFINER IS
BEGIN
    $IF my_debug.debug $THEN
        DBMS_OUTPUT.put_line('Debugging ON');
    $ELSE
        DBMS_OUTPUT.put_line('Debugging OFF');
    $END
END my_proc1;
/

CREATE PROCEDURE my_proc2 AUTHID DEFINER IS
BEGIN
    $IF my_debug.trace $THEN
        DBMS_OUTPUT.put_line('Tracing ON');
    $ELSE
        DBMS_OUTPUT.put_line('Tracing OFF');
    $END
END my_proc2;
/
```

### 3.8 Error-Reporting Functions

PL/SQL has two error-reporting functions, `SQLCODE` and `SQLERRM`, for use in PL/SQL exception-handling code.
3.9 Conditional Compilation

Conditional compilation lets you customize the functionality of a PL/SQL application without removing source text.

For example, you can:

- Use new features with the latest database release and disable them when running the application in an older database release.
- Activate debugging or tracing statements in the development environment and hide them when running the application at a production site.

Topics

- How Conditional Compilation Works
- Conditional Compilation Examples
- Retrieving and Printing Post-Processed Source Text
- Conditional Compilation Directive Restrictions

3.9.1 How Conditional Compilation Works

Conditional compilation uses selection directives, which are similar to `IF` statements, to select source text for compilation.

The condition in a selection directive usually includes an inquiry directive. Error directives raise user-defined errors. All conditional compilation directives are built from preprocessor control tokens and PL/SQL text.

Topics

- Preprocessor Control Tokens
- Selection Directives
- Error Directives
- Inquiry Directives
- DBMS_DB_VERSION Package

See Also:

"Static Expressions"
3.9.1.1 Preprocessor Control Tokens

A preprocessor control token identifies code that is processed before the PL/SQL unit is compiled.

Syntax

$plsql_identifier

There cannot be space between $ and plsql_identifier.

The character $ can also appear inside plsql_identifier, but it has no special meaning there.

These preprocessor control tokens are reserved:

- $IF
- $THEN
- $ELSE
- $ELSIF
- $ERROR

For information about plsql_identifier, see "Identifiers".

3.9.1.2 Selection Directives

A selection directive selects source text to compile.

Syntax

$IF boolean_static_expression $THEN
  text
[ $ELSIF boolean_static_expression $THEN
  text
]...
[ $ELSE
  text
$END
]

For the syntax of boolean_static_expression, see "BOOLEAN Static Expressions". The text can be anything, but typically, it is either a statement (see "statement ::=") or an error directive (explained in "Error Directives").

The selection directive evaluates the BOOLEAN static expressions in the order that they appear until either one expression has the value TRUE or the list of expressions is exhausted. If one expression has the value TRUE, its text is compiled, the remaining expressions are not evaluated, and their text is not analyzed. If no expression has the value TRUE, then if $ELSE is present, its text is compiled; otherwise, no text is compiled.

For examples of selection directives, see "Conditional Compilation Examples".
3.9.3 Error Directives

An **error directive** produces a user-defined error message during compilation.

**Syntax**

```
$ERROR varchar2_static_expression $END
```

It produces this compile-time error message, where `string` is the value of `varchar2_static_expression`:

`PLS-00179: $ERROR: string`

For the syntax of `varchar2_static_expression`, see "VARCHAR2 Static Expressions".

For an example of an error directive, see Example 3-58.

3.9.4 Inquiry Directives

An **inquiry directive** provides information about the compilation environment.

**Syntax**

```
$$name
```

For information about `name`, which is an unquoted PL/SQL identifier, see "Identifiers".

An inquiry directive typically appears in the `boolean_static_expression` of a selection directive, but it can appear anywhere that a variable or literal of its type can appear. Moreover, it can appear where regular PL/SQL allows only a literal (not a variable)—for example, to specify the size of a `VARCHAR2` variable.

**Topics**

- Predefined Inquiry Directives
- Assigning Values to Inquiry Directives
- Unresolvable Inquiry Directives

3.9.1.4.1 Predefined Inquiry Directives

The predefined inquiry directives are:

- `$$PLSQL_LINE`
  
  A `PLS_INTEGER` literal whose value is the number of the source line on which the directive appears in the current PL/SQL unit. An example of `$$PLSQL_LINE` in a selection directive is:
  
  ```
  $IF $$PLSQL_LINE = 32 $THEN ...
  ```
• **$PLSQL_UNIT**  
  A **VARCHAR2** literal that contains the name of the current PL/SQL unit. If the current PL/SQL unit is an anonymous block, then **$PLSQL_UNIT** contains a **NULL** value.

• **$PLSQL_UNIT_OWNER**  
  A **VARCHAR2** literal that contains the name of the owner of the current PL/SQL unit. If the current PL/SQL unit is an anonymous block, then **$PLSQL_UNIT_OWNER** contains a **NULL** value.

• **$PLSQL_UNIT_TYPE**  
  A **VARCHAR2** literal that contains the type of the current PL/SQL unit—**ANONYMOUS BLOCK**, **FUNCTION**, **PACKAGE**, **PACKAGE BODY**, **PROCEDURE**, **TRIGGER**, **TYPE**, or **TYPE BODY**. Inside an anonymous block or non-DML trigger, **$PLSQL_UNIT_TYPE** has the value **ANONYMOUS BLOCK**.

• **$plsql_compilation_parameter**  
  The name **plsql_compilation_parameter** is a PL/SQL compilation parameter (for example, **PLSCOPE_SETTINGS**). For descriptions of these parameters, see Table 2-2.

Because a selection directive needs a **BOOLEAN** static expression, you cannot use **$PLSQL_UNIT**, **$PLSQL_UNIT_OWNER**, or **$PLSQL_UNIT_TYPE** in a **VARCHAR2** comparison such as:

```sql
$IF $PLSQL_UNIT = 'AWARD_BONUS' $THEN ...
$IF $PLSQL_UNIT_OWNER IS HR $THEN ...
$IF $PLSQL_UNIT_TYPE IS FUNCTION $THEN ...
```

However, you can compare the preceding directives to **NULL**. For example:

```sql
$IF $PLSQL_UNIT IS NULL $THEN ...
$IF $PLSQL_UNIT_OWNER IS NOT NULL $THEN ...
$IF $PLSQL_UNIT_TYPE IS NULL $THEN ...
```

**Example 3-55  Predefined Inquiry Directives**

In this example, a SQL*Plus script, uses several predefined inquiry directives as **PLS_INTEGER** and **VARCHAR2** literals to show how their values are assigned.

```sql
SQL> CREATE OR REPLACE PROCEDURE p
2  AUTHID DEFINER IS
3    i PLS_INTEGER;
4 BEGIN
5    DBMS_OUTPUT.PUT_LINE('Inside p');
6    i := $PLSQL_LINE;
7    DBMS_OUTPUT.PUT_LINE('i = ' || i);
8    DBMS_OUTPUT.PUT_LINE('$$PLSQL_LINE = ' || $$PLSQL_LINE);
9    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT = ' || $$PLSQL_UNIT);
10   DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_OWNER = ' || $$PLSQL_UNIT_OWNER);
11   DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_TYPE = ' || $$PLSQL_UNIT_TYPE);
12 END;
13 /
```

Procedure created.

```sql
SQL> BEGIN
2    p;
3    DBMS_OUTPUT.PUT_LINE('Outside p');
4    DBMS_OUTPUT.PUT_LINE('$$PLSQL_LINE = ' || $$PLSQL_LINE);
```
BEGIN
    DBMS_OUTPUT.PUT_LINE('$$PLSCOPE_SETTINGS = ' || $$PLSCOPE_SETTINGS);
    DBMS_OUTPUT.PUT_LINE('$$PLSQL_CCFLAGS = ' || $$PLSQL_CCFLAGS);
    DBMS_OUTPUT.PUT_LINE('$$PLSQL_CODE_TYPE = ' || $$PLSQL_CODE_TYPE);
    DBMS_OUTPUT.PUT_LINE('$$PLSQL_OPTIMIZE_LEVEL = ' || $$PLSQL_OPTIMIZE_LEVEL);
    DBMS_OUTPUT.PUT_LINE('$$PLSQL_WARNINGS = ' || $$PLSQL_WARNINGS);
    DBMS_OUTPUT.PUT_LINE('$$NLS_LENGTH_SEMANTICS = ' || $$NLS_LENGTH_SEMANTICS);
END;
/

Result:

$$PLSCOPE_SETTINGS = IDENTIFIERS:NONE
$$PLSQL_CCFLAGS =
$$PLSQL_CODE_TYPE = INTERPRETED
$$PLSQL_OPTIMIZE_LEVEL = 2
$$PLSQL_WARNINGS = ENABLE:ALL
$$NLS_LENGTH_SEMANTICS = BYTE
3.9.1.4.2 Assigning Values to Inquiry Directives

You can assign values to inquiry directives with the `PLSQL_CCFLAGS` compilation parameter.

For example:

```sql
ALTER SESSION SET PLSQL_CCFLAGS =
  'name1:value1, name2:value2, ... namen:valuen'
```

Each value must be either a `BOOLEAN` literal (`TRUE`, `FALSE`, or `NULL`) or `PLS_INTEGER` literal. The data type of value determines the data type of name.

The same name can appear multiple times, with values of the same or different data types. Later assignments override earlier assignments. For example, this command sets the value of $$flag to 5 and its data type to `PLS_INTEGER`:

```sql
ALTER SESSION SET PLSQL_CCFLAGS = 'flag:TRUE, flag:5'
```

Oracle recommends against using `PLSQL_CCFLAGS` to assign values to predefined inquiry directives, including compilation parameters. To assign values to compilation parameters, Oracle recommends using the `ALTER SESSION` statement.

For more information about the `ALTER SESSION` statement, see *Oracle Database SQL Language Reference*.

---

**Note:**

The compile-time value of `PLSQL_CCFLAGS` is stored with the metadata of stored PL/SQL units, which means that you can reuse the value when you explicitly recompile the units. For more information, see "PL/SQL Units and Compilation Parameters".

For more information about `PLSQL_CCFLAGS`, see *Oracle Database Reference*.

---

**Example 3-57  PLSQL_CCFLAGS Assigns Value to Itself**

This example uses `PLSQL_CCFLAGS` to assign a value to the user-defined inquiry directive $$Some_Flag and (though not recommended) to itself. Because later assignments override earlier assignments, the resulting value of $$Some_Flag is 2 and the resulting value of `PLSQL_CCFLAGS` is the value that it assigns to itself (99), not the value that the `ALTER SESSION` statement assigns to it ('Some_Flag:1, Some_Flag:2, `PLSQL_CCFlags:99`).

```sql
ALTER SESSION SET
PLSQL_CCFlags = 'Some_Flag:1, Some_Flag:2, PLSQL_CCFlags:99'
/
BEGIN
  DBMS_OUTPUT.PUT_LINE($$Some_Flag);
  DBMS_OUTPUT.PUT_LINE($$PLSQL_CCFlags);
END;
/
```

Result:
3.9.1.3 Unresolvable Inquiry Directives

If the source text is not wrapped, PL/SQL issues a warning if the value of an inquiry directive cannot be determined.

If an inquiry directive (\$\$name\$) cannot be resolved, and the source text is not wrapped, then PL/SQL issues the warning PLW-6003 and substitutes NULL for the value of the unresolved inquiry directive. If the source text is wrapped, the warning message is disabled, so that the unresolved inquiry directive is not revealed.

For information about wrapping PL/SQL source text, see PL/SQL Source Text Wrapping.

3.9.1.5 DBMS_DB_VERSION Package

The DBMS_DB_VERSION package specifies the Oracle version numbers and other information useful for simple conditional compilation selections based on Oracle versions.

The DBMS_DB_VERSION package provides these static constants:

- The PLS_INTEGER constant VERSION identifies the current Oracle Database version.
- The PLS_INTEGER constant RELEASE identifies the current Oracle Database release number.
- Each BOOLEAN constant of the form VER_LE_v has the value TRUE if the database version is less than or equal to v; otherwise, it has the value FALSE.
- Each BOOLEAN constant of the form VER_LE_v_r has the value TRUE if the database version is less than or equal to v and release is less than or equal to r; otherwise, it has the value FALSE.

For more information about the DBMS_DB_VERSION package, see Oracle Database PL/SQL Packages and Types Reference.

3.9.2 Conditional Compilation Examples

Examples of conditional compilation using selection and user-defined inquiry directives.

Example 3-58 Code for Checking Database Version

This example generates an error message if the database version and release is less than Oracle Database 10g release 2; otherwise, it displays a message saying that the version and release are supported and uses a COMMIT statement that became available at Oracle Database 10g release 2.

```
BEGIN
  $IF DBMS_DB_VERSION.VER_LE_10_1 $THEN  -- selection directive begins
    $ERROR 'unsupported database release' $END  -- error directive
  $ELSE
    DBMS_OUTPUT.PUT_LINE ('Release ' || DBMS_DB_VERSION.VERSION || '.' ||
      DBMS_DB_VERSION.RELEASE || ' is supported.');
  $ENDIF
```
-- This COMMIT syntax is newly supported in 10.2:
COMMIT WRITE IMMEDIATE NOWAIT;
$END -- selection directive ends
END;
/

Result:
Release 12.1 is supported.

Example 3-59 Compiling Different Code for Different Database Versions

This example sets the values of the user-defined inquiry directives $$my_debug and $$my_tracing and then uses conditional compilation:

• In the specification of package my_pkg, to determine the base type of the subtype my_real (BINARY_DOUBLE is available only for Oracle Database versions 10g and later.)

• In the body of package my_pkg, to compute the values of my_pi and my_e differently for different database versions

• In the procedure circle_area, to compile some code only if the inquiry directive $$my_debug has the value TRUE.

ALTER SESSION SET PLSQL_CCFLAGS = 'my_debug:FALSE, my_tracing:FALSE';

CREATE OR REPLACE PACKAGE my_pkg AUTHID DEFINER AS
    SUBTYPE my_real IS
        $IF DBMS_DB_VERSION.VERSION < 10 $THEN
            NUMBER;
        $ELSE
            BINARY_DOUBLE;
        $END

        my_pi my_real;
        my_e  my_real;
    END my_pkg;
/

CREATE OR REPLACE PACKAGE BODY my_pkg AS
BEGIN
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN
        my_pi := 3.14159265358979323846264338327950288420;
        my_e  := 2.71828182845904523536028747135266249775;
    $ELSE
        my_pi := 3.14159265358979323846264338327950288420d;
        my_e  := 2.71828182845904523536028747135266249775d;
    $END
    END my_pkg;
/

CREATE OR REPLACE PROCEDURE circle_area(radius my_pkg.my_real) AUTHID DEFINER IS
    my_area       my_pkg.my_real;
    my_data_type  VARCHAR2(30);
BEGIN
    my_area := my_pkg.my_pi * (radius**2);

    DBMS_OUTPUT.PUT_LINE
        ("Radius: ' || TO_CHAR(radius) || ' Area: ' || TO_CHAR(my_area));
    

$IF $my_debug $THEN
  SELECT DATA_TYPE INTO my_data_type
  FROM USER_ARGUMENTS
  WHERE OBJECT_NAME = 'CIRCLE_AREA'
  AND ARGUMENT_NAME = 'RADIUS';

  DBMS_OUTPUT.PUT_LINE
  ('Data type of the RADIUS argument is: ' || my_data_type);
$END
END;
/

CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE
('PACKAGE', 'HR', 'MY_PKG');

Result:
PACKAGE my_pkg AUTHID DEFINER AS
  SUBTYPE my_real IS
    BINARY_DOUBLE;
  my_pi my_real;
  my_e my_real;
END my_pkg;

Call completed.

3.9.3 Retrieving and Printing Post-Processed Source Text

The DBMS_PREPROCESSOR package provides subprograms that retrieve and print the source text of a PL/SQL unit in its post-processed form.

For information about the DBMS_PREPROCESSOR package, see Oracle Database PL/SQL Packages and Types Reference.

Example 3-60    Displaying Post-Processed Source Text

This example invokes the procedure
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE to print the post-processed form of my_pkg (from "Example 3-59"). Lines of code in "Example 3-59" that are not included in the post-processed text appear as blank lines.

CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE
  ('PACKAGE', 'HR', 'MY_PKG');

Result:
PACKAGE my_pkg AUTHID DEFINER AS
  SUBTYPE my_real IS
    BINARY_DOUBLE;
  my_pi my_real;
  my_e my_real;
END my_pkg;

3.9.4 Conditional Compilation Directive Restrictions

Conditional compilation directives are subject to these semantic restrictions.

A conditional compilation directive cannot appear in the specification of a schema-level user-defined type (created with the "CREATE TYPE Statement"). This type
specification specifies the attribute structure of the type, which determines the attribute structure of dependent types and the column structure of dependent tables.

⚠️ **Caution:**

Using a conditional compilation directive to change the attribute structure of a type can cause dependent objects to "go out of sync" or dependent tables to become inaccessible. Oracle recommends that you change the attribute structure of a type only with the "ALTER TYPE Statement". The **ALTER TYPE** statement propagates changes to dependent objects.

If a conditional compilation directive is used in a schema-level type specification, the compiler raises the error **PLS-00180: preprocessor directives are not supported in this context.**

As all conditional compiler constructs are processed by the PL/SQL preprocessor, the SQL Parser imposes the following restrictions on the location of the first conditional compilation directive in a stored PL/SQL unit or anonymous block:

- In a package specification, a package body, a type body, a schema-level function and in a schema-level procedure, at least one nonwhitespace PL/SQL token must appear after the identifier of the unit name before a conditional compilation directive is valid.

**Note:**

- The PL/SQL comments, "--" or "/*", are counted as whitespace tokens.
- If the token is invalid in PL/SQL, then a PLS-00103 error is issued. But if a conditional compilation directive is used in violation of this rule, then an ORA error is produced.

**Example 3-61** and **Example 3-62**, show that the first conditional compilation directive appears after the first PL/SQL token that follows the identifier of the unit being defined.

- In a trigger or an anonymous block, the first conditional compilation directive cannot appear before the keyword **DECLARE** or **BEGIN**, whichever comes first.

The SQL parser also imposes this restriction: If an anonymous block uses a placeholder, the placeholder cannot appear in a conditional compilation directive. For example:

```sql
BEGIN
  :n := 1; -- valid use of placeholder
  $IF ... $THEN
    :n := 1; -- invalid use of placeholder
  $END
```
Example 3-61  Using Conditional Compilation Directive in the Definition of a Package Specification

This example shows the placement of the first conditional compilation directive after an AUTHID clause, but before the keyword IS, in the definition of the package specification.

```
CREATE OR REPLACE PACKAGE cc_pkg
  AUTHID DEFINER
  $IF $$XFLAG $THEN ACCESSIBLE BY(pl_pkg) $END
IS
  i NUMBER := 10;
  trace CONSTANT BOOLEAN := TRUE;
END cc_pkg;
```

Result:

Package created.

Example 3-62  Using Conditional Compilation Directive in the Formal Parameter List of a Subprogram

This example shows the placement of the first conditional compilation directive after the left parenthesis, in the formal parameter list of a PL/SQL procedure definition.

```
CREATE OR REPLACE PROCEDURE my_proc (    $IF $$xxx $THEN i IN PLS_INTEGER $ELSE i IN INTEGER $END
  ) IS
BEGIN
  NULL;
END my_proc;
```

Result:

Procedure created.
PL/SQL Data Types

Every PL/SQL constant, variable, parameter, and function return value has a data type that determines its storage format and its valid values and operations.

This chapter explains scalar data types, which store values with no internal components.

A scalar data type can have subtypes. A subtype is a data type that is a subset of another data type, which is its base type. A subtype has the same valid operations as its base type. A data type and its subtypes comprise a data type family.

PL/SQL predefines many types and subtypes in the package STANDARD and lets you define your own subtypes.

The PL/SQL scalar data types are:

• The SQL data types
  • BOOLEAN
  • PLS_INTEGER
  • BINARY_INTEGER
  • REF CURSOR
  • User-defined subtypes

Topics

• SQL Data Types
  • BOOLEAN Data Type
  • PLS_INTEGER and BINARY_INTEGER Data Types
  • SIMPLE_INTEGER Subtype of PLS_INTEGER
  • User-Defined PL/SQL Subtypes

See Also:

• “PL/SQL Collections and Records" for information about composite data types
• "Cursor Variables" for information about REF CURSOR
• "CREATE TYPE Statement" for information about creating schema-level user-defined data types
• “PL/SQL Predefined Data Types” for the predefined PL/SQL data types and subtypes, grouped by data type family
4.1 SQL Data Types

The PL/SQL data types include the SQL data types.

For information about the SQL data types, see *Oracle Database SQL Language Reference*—all information there about data types and subtypes, data type comparison rules, data conversion, literals, and format models applies to both SQL and PL/SQL, except as noted here:

- Different Maximum Sizes
- Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE
- Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

Unlike SQL, PL/SQL lets you declare variables, to which the following topics apply:

- CHAR and VARCHAR2 Variables
- LONG and LONG RAW Variables
- ROWID and UROWID Variables

4.1.1 Different Maximum Sizes

The SQL data types listed in Table 4-1 have different maximum sizes in PL/SQL and SQL.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Maximum Size in PL/SQL</th>
<th>Maximum Size in SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR^1</td>
<td>32,767 bytes</td>
<td>2,000 bytes</td>
</tr>
<tr>
<td>NCHAR^1</td>
<td>32,767 bytes</td>
<td>2,000 bytes</td>
</tr>
<tr>
<td>RAW^1</td>
<td>32,767 bytes</td>
<td>2,000 bytes^2</td>
</tr>
<tr>
<td>VARCHAR2^1</td>
<td>32,767 bytes</td>
<td>4,000 bytes^2</td>
</tr>
<tr>
<td>NVARCHAR2^1</td>
<td>32,767 bytes</td>
<td>4,000 bytes^2</td>
</tr>
<tr>
<td>LONG^3</td>
<td>32,760 bytes</td>
<td>2 gigabytes (GB) - 1</td>
</tr>
<tr>
<td>LONG RAW^3</td>
<td>32,760 bytes</td>
<td>2 GB</td>
</tr>
<tr>
<td>BLOB</td>
<td>128 terabytes (TB)</td>
<td>(4 GB - 1) * \textit{database_block_size}</td>
</tr>
<tr>
<td>CLOB</td>
<td>128 TB</td>
<td>(4 GB - 1) * \textit{database_block_size}</td>
</tr>
<tr>
<td>NCLOB</td>
<td>128 TB</td>
<td>(4 GB - 1) * \textit{database_block_size}</td>
</tr>
</tbody>
</table>

^1 When specifying the maximum size of a value of this data type in PL/SQL, use an integer literal (not a constant or variable) whose value is in the range from 1 through 32,767.

^2 To eliminate this size difference, follow the instructions in *Oracle Database SQL Language Reference*.

^3 Supported only for backward compatibility with existing applications.
4.1.2 Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE

The SQL data types BINARY_FLOAT and BINARY_DOUBLE represent single-precision and double-precision IEEE 754-format floating-point numbers, respectively.

BINARY_FLOAT and BINARY_DOUBLE computations do not raise exceptions, so you must check the values that they produce for conditions such as overflow and underflow by comparing them to predefined constants (for examples, see Oracle Database SQL Language Reference). PL/SQL has more of these constants than SQL does.

Table 4-2 lists and describes the predefined PL/SQL constants for BINARY_FLOAT and BINARY_DOUBLE, and identifies those that SQL also defines.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY_FLOAT_NAN (*)</td>
<td>BINARY_FLOAT value for which the condition IS NAN (not a number) is true</td>
</tr>
<tr>
<td>BINARY_FLOAT_INFINITY</td>
<td>Single-precision positive infinity</td>
</tr>
<tr>
<td>BINARY_FLOAT_MAX_NORMAL</td>
<td>Maximum normal BINARY_FLOAT value</td>
</tr>
<tr>
<td>BINARY_FLOAT_MIN_NORMAL</td>
<td>Minimum normal BINARY_FLOAT value</td>
</tr>
<tr>
<td>BINARY_FLOAT_MAX_SUBNORMAL</td>
<td>Maximum subnormal BINARY_FLOAT value</td>
</tr>
<tr>
<td>BINARY_FLOAT_MIN_SUBNORMAL</td>
<td>Minimum subnormal BINARY_FLOAT value</td>
</tr>
<tr>
<td>BINARY_DOUBLE_NAN (*)</td>
<td>BINARY_DOUBLE value for which the condition IS NAN (not a number) is true</td>
</tr>
<tr>
<td>BINARY_DOUBLE_INFINITY</td>
<td>Double-precision positive infinity</td>
</tr>
<tr>
<td>BINARY_DOUBLE_MAX_NORMAL</td>
<td>Maximum normal BINARY_DOUBLE value</td>
</tr>
<tr>
<td>BINARY_DOUBLE_MIN_NORMAL</td>
<td>Minimum normal BINARY_DOUBLE value</td>
</tr>
<tr>
<td>BINARY_DOUBLE_MAX_SUBNORMAL</td>
<td>Maximum subnormal BINARY_DOUBLE value</td>
</tr>
<tr>
<td>BINARY_DOUBLE_MIN_SUBNORMAL</td>
<td>Minimum subnormal BINARY_DOUBLE value</td>
</tr>
</tbody>
</table>

(*) SQL also predefines this constant.

4.1.3 Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

PL/SQL predefines these subtypes:

- SIMPLE_FLOAT, a subtype of SQL data type BINARY_FLOAT
- SIMPLE_DOUBLE, a subtype of SQL data type BINARY_DOUBLE

Each subtype has the same range as its base type and has a NOT NULL constraint (explained in "NOT NULL Constraint").
If you know that a variable will never have the value NULL, declare it as SIMPLE_FLOAT or SIMPLE_DOUBLE, rather than BINARY_FLOAT or BINARY_DOUBLE. Without the overhead of checking for nullness, the subtypes provide significantly better performance than their base types. The performance improvement is greater with PLSQL_CODE_TYPE='NATIVE' than with PLSQL_CODE_TYPE='INTERPRETED' (for more information, see "Use Data Types that Use Hardware Arithmetic").

4.1.4 CHAR and VARCHAR2 Variables

Topics

• Assigning or Inserting Too-Long Values
• Declaring Variables for Multibyte Characters
• Differences Between CHAR and VARCHAR2 Data Types

4.1.4.1 Assigning or Inserting Too-Long Values

If the value that you assign to a character variable is longer than the maximum size of the variable, an error occurs. For example:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := 'abc  ';
END;
/
```

Result:

```
DECLARE
  *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 4

Similarly, if you insert a character variable into a column, and the value of the variable is longer than the defined width of the column, an error occurs. For example:

```
DROP TABLE t;
CREATE TABLE t (c CHAR(3 CHAR));

DECLARE
  s VARCHAR2(5 CHAR) := 'abc  ';
BEGIN
  INSERT INTO t(c) VALUES(s);
END;
/
```

Result:

```
BEGIN
  *
ERROR at line 1:
ORA-12899: value too large for column "HR"."T"."C" (actual: 5, maximum: 3)
ORA-06512: at line 4
```
To strip trailing blanks from a character value before assigning it to a variable or inserting it into a column, use the `RTRIM` function, explained in *Oracle Database SQL Language Reference*. For example:

```sql
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := RTRIM('abc  ');
  INSERT INTO t(c) VALUES(RTRIM('abc  '));
END;
/
```

Result:
PL/SQL procedure successfully completed.

### 4.1.4.2 Declaring Variables for Multibyte Characters

The maximum size of a `CHAR` or `VARCHAR2` variable is 32,767 bytes, whether you specify the maximum size in characters or bytes. The maximum number of characters in the variable depends on the character set type and sometimes on the characters themselves:

<table>
<thead>
<tr>
<th>Character Set Type</th>
<th>Maximum Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-byte character set</td>
<td>32,767</td>
</tr>
<tr>
<td><code>n</code>-byte fixed-width multibyte character set (for example, AL16UTF16)</td>
<td>FLOOR(32,767/n)</td>
</tr>
<tr>
<td><code>n</code>-byte variable-width multibyte character set with character widths between 1 and <code>n</code> bytes (for example, JA16SJIS or AL32UTF8)</td>
<td>Depends on characters themselves—can be anything from 32,767 (for a string containing only 1-byte characters) through FLOOR(32,767/n) (for a string containing only <code>n</code>-byte characters).</td>
</tr>
</tbody>
</table>

When declaring a `CHAR` or `VARCHAR2` variable, to ensure that it can always hold `n` characters in any multibyte character set, declare its length in characters—that is, `CHAR(n CHAR)` or `VARCHAR2(n CHAR)`, where `n` does not exceed FLOOR(32767/4) = 8191.

#### See Also:

*Oracle Database Globalization Support Guide* for information about Oracle Database character set support

### 4.1.4.3 Differences Between CHAR and VARCHAR2 Data Types

`CHAR` and `VARCHAR2` data types differ in:

- Predefined Subtypes
- How Blank-Padding Works
- Value Comparisons
4.1.4.3.1 Predefined Subtypes

The **CHAR** data type has one predefined subtype in both PL/SQL and SQL—**CHARACTER**.

The **VARCHAR2** data type has one predefined subtype in both PL/SQL and SQL, **VARCHAR**, and an additional predefined subtype in PL/SQL, **STRING**.

Each subtype has the same range of values as its base type.

---

**Note:**

In a future PL/SQL release, to accommodate emerging SQL standards, **VARCHAR** might become a separate data type, no longer synonymous with **VARCHAR2**.

---

4.1.4.3.2 How Blank-Padding Works

This explains the differences and considerations of using blank-padding with **CHAR** and **VARCHAR2**.

Consider these situations:

- The value that you assign to a variable is shorter than the maximum size of the variable.
- The value that you insert into a column is shorter than the defined width of the column.
- The value that you retrieve from a column into a variable is shorter than the maximum size of the variable.

If the data type of the receiver is **CHAR**, PL/SQL blank-pads the value to the maximum size. Information about trailing blanks in the original value is lost.

If the data type of the receiver is **VARCHAR2**, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, and no information is lost.

**Example 4-1  CHAR and VARCHAR2 Blank-Padding Difference**

In this example, both the **CHAR** variable and the **VARCHAR2** variable have the maximum size of 10 characters. Each variable receives a five-character value with one trailing blank. The value assigned to the **CHAR** variable is blank-padded to 10 characters, and you cannot tell that one of the six trailing blanks in the resulting value was in the original value. The value assigned to the **VARCHAR2** variable is not changed, and you can see that it has one trailing blank.

```sql
DECLARE
  first_name  CHAR(10 CHAR);
  last_name   VARCHAR2(10 CHAR);
BEGIN
  first_name := 'John ';
  last_name  := 'Chen ';
  DBMS_OUTPUT.PUT_LINE('**' || first_name || '**');
  DBMS_OUTPUT.PUT_LINE('**' || last_name || '**');
END;
```

---
4.1.4.3.3 Value Comparisons

The SQL rules for comparing character values apply to PL/SQL character variables.

Whenever one or both values in the comparison have the data type VARCHAR2 or NVARCHAR2, nonpadded comparison semantics apply; otherwise, blank-padded semantics apply. For more information, see Oracle Database SQL Language Reference.

4.1.5 LONG and LONG RAW Variables

Note:

Oracle supports the LONG and LONG RAW data types only for backward compatibility with existing applications. For new applications:

- Instead of LONG, use VARCHAR2(32760), BLOB, CLOB or NCLOB.
- Instead of LONG RAW, use BLOB.

You can insert any LONG value into a LONG column. You can insert any LONG RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,760 bytes from a LONG or LONG RAW column into a LONG or LONG RAW variable.

You can insert any CHAR or VARCHAR2 value into a LONG column. You cannot retrieve a value longer than 32,767 bytes from a LONG column into a CHAR or VARCHAR2 variable.

You can insert any RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,767 bytes from a LONG RAW column into a RAW variable.

See Also:

"Trigger LONG and LONG RAW Data Type Restrictions" for restrictions on LONG and LONG RAW data types in triggers

4.1.6 ROWID and UROWID Variables

When you retrieve a rowid into a ROWID variable, use the ROWIDTOCHAR function to convert the binary value to a character value. For information about this function, see Oracle Database SQL Language Reference.
To convert the value of a ROWID variable to a rowid, use the CHARTOROWID function, explained in Oracle Database SQL Language Reference. If the value does not represent a valid rowid, PL/SQL raises the predefined exception SYS_INVALID_ROWID.

To retrieve a rowid into a UROWID variable, or to convert the value of a UROWID variable to a rowid, use an assignment statement; conversion is implicit.

**Note:**

- UROWID is a more versatile data type than ROWID, because it is compatible with both logical and physical rowids.
- When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the ROWID of the row changes. HCC, a feature of certain Oracle storage systems, is described in Oracle Database Concepts.

**See Also:**

Oracle Database PL/SQL Packages and Types Reference for information about the DBMS_ROWID package, whose subprograms let you create and return information about ROWID values (but not UROWID values)

## 4.2 BOOLEAN Data Type

The PL/SQL data type BOOLEAN stores logical values, which are the boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value.

The syntax for declaring an BOOLEAN variable is:

```plsql
variable_name BOOLEAN
```

The only value that you can assign to a BOOLEAN variable is a BOOLEAN expression. For details, see "BOOLEAN Expressions".

Because SQL has no data type equivalent to BOOLEAN, you cannot:

- Assign a BOOLEAN value to a database table column
- Select or fetch the value of a database table column into a BOOLEAN variable
- Use a BOOLEAN value in a SQL function
  (However, a SQL query can invoke a PL/SQL function that has a BOOLEAN parameter, as in "Example 4-3".)
- Use a BOOLEAN expression in a SQL statement, except as an argument to a PL/SQL function invoked in a SQL query, or in a PL/SQL anonymous block.
An argument to a PL/SQL function invoked in a static SQL query cannot be a BOOLEAN literal. The workaround is to assign the literal to a variable and then pass the variable to the function, as in "Example 4-3".

You cannot pass a BOOLEAN value to the DBMS_OUTPUT.PUT or DBMS_OUTPUT.PUTLINE subprogram. To print a BOOLEAN value, use an IF or CASE statement to translate it to a character value (for information about these statements, see "Conditional Selection Statements").

**Example 4-2  Printing BOOLEAN Values**

In this example, the procedure accepts a BOOLEAN parameter and uses a CASE statement to print Unknown if the value of the parameter is NULL, Yes if it is TRUE, and No if it is FALSE.

```plsql
PROCEDURE print_boolean (b BOOLEAN) AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (
      CASE
        WHEN b IS NULL THEN 'Unknown'
        WHEN b THEN 'Yes'
        WHEN NOT b THEN 'No'
      END
    );
  END;
END;
BEGIN
  print_boolean(TRUE);
  print_boolean(FALSE);
  print_boolean(NULL);
END;
```

**Result:**

Yes
No
Unknown

**Example 4-3  SQL Statement Invokes PL/SQL Function with BOOLEAN Parameter**

In this example, a SQL statement invokes a PL/SQL function that has a BOOLEAN parameter.

```plsql
FUNCTION f (x BOOLEAN, y PLS_INTEGER) RETURN employees.employee_id%TYPE AUTHID CURRENT_USER AS
```

---

**See Also:**

Example 3-34, which creates a print_boolean procedure that uses an IF statement.
BEGIN
  IF x THEN
    RETURN y;
  ELSE
    RETURN 2*y;
  END IF;
END;

DECLARE
  name employees.last_name%TYPE;
  b BOOLEAN := TRUE;
BEGIN
  SELECT last_name INTO name
  FROM employees
  WHERE employee_id = f(b, 100);

  DBMS_OUTPUT.PUT_LINE(name);

  b := FALSE;

  SELECT last_name INTO name
  FROM employees
  WHERE employee_id = f(b, 100);

  DBMS_OUTPUT.PUT_LINE(name);
END;
/

Result:

King
Whalen

4.3 PLS_INTEGER and BINARY_INTEGER Data Types

The PL/SQL data types PLS_INTEGER and BINARY_INTEGER are identical.

For simplicity, this document uses PLS_INTEGER to mean both PLS_INTEGER and BINARY_INTEGER.

The PLS_INTEGER data type stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.

The PLS_INTEGER data type has these advantages over the NUMBER data type and NUMBER subtypes:

- PLS_INTEGER values require less storage.
- PLS_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic.

For efficiency, use PLS_INTEGER values for all calculations in its range.

Topics

- Preventing PLS_INTEGER Overflow
- Predefined PLS_INTEGER Subtypes
- SIMPLE_INTEGER Subtype of PLS_INTEGER
4.3.1 Preventing PLS_INTEGER Overflow

A calculation with two PLS_INTEGER values that overflows the PLS_INTEGER range raises an overflow exception.

For calculations outside the PLS_INTEGER range, use INTEGER, a predefined subtype of the NUMBER data type.

Example 4-4 PLS_INTEGER Calculation Raises Overflow Exception

This example shows that a calculation with two PLS_INTEGER values that overflows the PLS_INTEGER range raises an overflow exception, even if you assign the result to a NUMBER data type.

```
DECLARE
    p1 PLS_INTEGER := 2147483647;
    p2 PLS_INTEGER := 1;
    n NUMBER;
BEGIN
    n := p1 + p2;
END;
/
```

Result:

```
DECLARE
    *
ERROR at line 1:
ORA-01426: numeric overflow
ORA-06512: at line 6
```

Example 4-5 Preventing Example 4-4 Overflow

This example shows the correct use of the INTEGER predefined subtype for calculations outside the PLS_INTEGER range.

```
DECLARE
    p1 PLS_INTEGER := 2147483647;
    p2 INTEGER := 1;
    n NUMBER;
BEGIN
    n := p1 + p2;
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

4.3.2 Predefined PLS_INTEGER Subtypes

This summary lists the predefined subtypes of the PLS_INTEGER data type and describes the data they store.
Table 4-3 Predefined Subtypes of PLS_INTEGER Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NATURAL</td>
<td>Nonnegative PLS_INTEGER value</td>
</tr>
<tr>
<td>NATURALN</td>
<td>Nonnegative PLS_INTEGER value with NOT NULL constraint</td>
</tr>
<tr>
<td>POSITIVE</td>
<td>Positive PLS_INTEGER value</td>
</tr>
<tr>
<td>POSITIVEN</td>
<td>Positive PLS_INTEGER value with NOT NULL constraint</td>
</tr>
<tr>
<td>SIGNTYPE</td>
<td>PLS_INTEGER value -1, 0, or 1 (useful for programming tri-state logic)</td>
</tr>
<tr>
<td>SIMPLE_INTEGER</td>
<td>PLS_INTEGER value with NOT NULL constraint.</td>
</tr>
</tbody>
</table>

PLS_INTEGER and its subtypes can be implicitly converted to these data types:

- CHAR
- VARCHAR2
- NUMBER
- LONG

All of the preceding data types except LONG, and all PLS_INTEGER subtypes, can be implicitly converted to PLS_INTEGER.

A PLS_INTEGER value can be implicitly converted to a PLS_INTEGER subtype only if the value does not violate a constraint of the subtype.

See Also:

- "NOT NULL Constraint" for information about the NOT NULL constraint
- "SIMPLE_INTEGER Subtype of PLS_INTEGER" for more information about SIMPLE_INTEGER

Example 4-6 Violating Constraint of SIMPLE_INTEGER Subtype

This example shows that casting the PLS_INTEGER value NULL to the SIMPLE_INTEGER subtype raises an exception.

```
DECLARE
    a SIMPLE_INTEGER := 1;
    b PLS_INTEGER := NULL;
BEGIN
    a := b;
END;
/
```

Result:

```
DECLARE
    *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 5
```
4.3.3 SIMPLE_INTEGER Subtype of PLS_INTEGER

SIMPLE_INTEGER is a predefined subtype of the PLS_INTEGER data type.

SIMPLE_INTEGER has the same range as PLS_INTEGER and has a NOT NULL constraint. It differs significantly from PLS_INTEGER in its overflow semantics.

If you know that a variable will never have the value NULL or need overflow checking, declare it as SIMPLE_INTEGER rather than PLS_INTEGER. Without the overhead of checking for nullness and overflow, SIMPLE_INTEGER performs significantly better than PLS_INTEGER.

Topics

- SIMPLE_INTEGER Overflow Semantics
- Expressions with Both SIMPLE_INTEGER and Other Operands
- Integer Literals in SIMPLE_INTEGER Range

See Also:

"NOT NULL Constraint"

4.3.3.1 SIMPLE_INTEGER Overflow Semantics

If and only if all operands in an expression have the data type SIMPLE_INTEGER, PL/SQL uses two's complement arithmetic and ignores overflows.

Because overflows are ignored, values can wrap from positive to negative or from negative to positive; for example:

\[2^{30} + 2^{30} = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}\]

\[-2^{31} + -2^{31} = 0x80000000 + 0x80000000 = 0x00000000 = 0\]

For example, this block runs without errors:

```plsql
DECLARE
    n SIMPLE_INTEGER := 2147483645;
BEGIN
    FOR j IN 1..4 LOOP
        n := n + 1;
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
    END LOOP;
    FOR j IN 1..4 LOOP
        n := n - 1;
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
    END LOOP;
END;
/
```

Result:

```
+2147483646
+2147483647
+2147483648
```

---

ORACLE®
4.3.3.2 Expressions with Both SIMPLE_INTEGER and Other Operands

If an expression has both SIMPLE_INTEGER and other operands, PL/SQL implicitly converts the SIMPLE_INTEGER values to NOT NULL.

The PL/SQL compiler issues a warning when SIMPLE_INTEGER and other values are mixed in a way that might negatively impact performance by inhibiting some optimizations.

4.3.3.3 Integer Literals in SIMPLE_INTEGER Range

Integer literals in the SIMPLE_INTEGER range have the data type SIMPLE_INTEGER.

However, to ensure backward compatibility, when all operands in an arithmetic expression are integer literals, PL/SQL treats the integer literals as if they were cast to PLS_INTEGER.

4.4 User-Defined PL/SQL Subtypes

PL/SQL lets you define your own subtypes.

The base type can be any scalar or user-defined PL/SQL data type specifier such as CHAR, DATE, or RECORD (including a previously defined user-defined subtype).

Note:

The information in this topic applies to both user-defined subtypes and the predefined subtypes listed in PL/SQL Predefined Data Types.

Subtypes can:

- Provide compatibility with ANSI/ISO data types
- Show the intended use of data items of that type
- Detect out-of-range values

Topics

- Unconstrained Subtypes
- Constrained Subtypes
- Subtypes with Base Types in Same Data Type Family
4.4.1 Unconstrained Subtypes

An **unconstrained subtype** has the same set of values as its base type, so it is only another name for the base type.

Therefore, unconstrained subtypes of the same base type are interchangeable with each other and with the base type. No data type conversion occurs.

To define an unconstrained subtype, use this syntax:

```
SUBTYPE subtype_name IS base_type
```

For information about `subtype_name` and `base_type`, see **subtype**.

An example of an unconstrained subtype, which PL/SQL predefines for compatibility with ANSI, is:

```
SUBTYPE "DOUBLE PRECISION" IS FLOAT
```

**Example 4-7  User-Defined Unconstrained Subtypes Show Intended Use**

In this example, the unconstrained subtypes `Balance` and `Counter` show the intended uses of data items of their types.

```
DECLARE
    SUBTYPE Balance IS NUMBER;
    checking_account        Balance(6,2);
    savings_account         Balance(8,2);
    certificate_of_deposit  Balance(8,2);
    max_insured  CONSTANT   Balance(8,2) := 250000.00;

    SUBTYPE Counter IS NATURAL;
    accounts     Counter := 1;
    deposits     Counter := 0;
    withdrawals  Counter := 0;
    overdrafts   Counter := 0;

    PROCEDURE deposit (account  IN OUT Balance, amount   IN     Balance)
    IS
        BEGIN
            account  := account + amount;
            deposits := deposits + 1;
        END;

    BEGIN
        NULL;
    END;
/
```

4.4.2 Constrained Subtypes

A **constrained subtype** has only a subset of the values of its base type.

If the base type lets you specify size, precision and scale, or a range of values, then you can specify them for its subtypes. The subtype definition syntax is:
SUBTYPE subtype_name IS base_type
   { precision [ , scale ] | RANGE low_value . . high_value } [ NOT NULL ]

Otherwise, the only constraint that you can put on its subtypes is NOT NULL:

SUBTYPE subtype_name IS base_type [ NOT NULL ]

---

**Note:**

The only base types for which you can specify a range of values are
**PLS_INTEGER** and its subtypes (both predefined and user-defined).

---

A constrained subtype can be implicitly converted to its base type, but the base type
can be implicitly converted to the constrained subtype only if the value does not violate
a constraint of the subtype.

A constrained subtype can be implicitly converted to another constrained subtype with
the same base type only if the source value does not violate a constraint of the target
subtype.

---

**See Also:**

- "subtype_definition ::=" syntax diagram
- "subtype" semantic description
- "Example 4-6", "Violating Constraint of SIMPLE_INTEGER Subtype"
- "Formal Parameters of Constrained Subtypes"
- "NOT NULL Constraint"

---

**Example 4-8  User-Defined Constrained Subtype Detects Out-of-Range Values**

In this example, the constrained subtype **Balance** detects out-of-range values.

```plsql
DECLARE
    SUBTYPE Balance IS NUMBER(8,2);

    checking_account   Balance;
    savings_account    Balance;

BEGIN
    checking_account := 2000.00;
    savings_account  := 1000000.00;
END;
/
```

**Result:**

```plsql
DECLARE
    *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 9
```

---
Example 4-9  Implicit Conversion Between Constrained Subtypes with Same Base Type

In this example, the three constrained subtypes have the same base type. The first two subtypes can be implicitly converted to the third subtype, but not to each other.

DECLARE
  SUBTYPE Digit        IS PLS_INTEGER RANGE 0..9;
  SUBTYPE Double_digit IS PLS_INTEGER RANGE 10..99;
  SUBTYPE Under_100    IS PLS_INTEGER RANGE 0..99;

  d   Digit        :=  4;
  dd  Double_digit := 35;
  u   Under_100;
BEGIN
  u := d;   -- Succeeds; Under_100 range includes Digit range
  u := dd;  -- Succeeds; Under_100 range includes Double_digit range
  dd := d;  -- Raises error; Double_digit range does not include Digit range
END;
/

Result:

DECLARE
  *
  ERROR at line 1:
  ORA-06502: PL/SQL: numeric or value error
  ORA-06512: at line 12

4.4.3 Subtypes with Base Types in Same Data Type Family

If two subtypes have different base types in the same data type family, then one subtype can be implicitly converted to the other only if the source value does not violate a constraint of the target subtype.

For the predefined PL/SQL data types and subtypes, grouped by data type family, see PL/SQL Predefined Data Types.

Example 4-10  Implicit Conversion Between Subtypes with Base Types in Same Family

In this example, the subtypes Word and Text have different base types in the same data type family. The first assignment statement implicitly converts a Word value to Text. The second assignment statement implicitly converts a Text value to Word. The third assignment statement cannot implicitly convert the Text value to Word, because the value is too long.

DECLARE
  SUBTYPE Word IS CHAR(6);
  SUBTYPE Text IS VARCHAR2(15);

  verb       Word := 'run';
  sentence1  Text;
  sentence2  Text := 'Hurry!';
  sentence3  Text := 'See Tom run.';
BEGIN
  sentence1 := verb;  -- 3-character value, 15-character limit
  verb := sentence2;  -- 6-character value, 6-character limit
  verb := sentence3;  -- 12-character value, 6-character limit
END;
/

Result:

DECLARE
  *
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 13
PL/SQL Control Statements

PL/SQL has three categories of control statements: conditional selection statements, loop statements and sequential control statements.

PL/SQL categories of control statements are:

- **Conditional selection statements**, which run different statements for different data values.
  The conditional selection statements are `IF` and `CASE`.

- **Loop statements**, which run the same statements with a series of different data values.
  The loop statements are the basic `LOOP`, `FOR LOOP`, and `WHILE LOOP`.
  The `EXIT` statement transfers control to the end of a loop. The `CONTINUE` statement exits the current iteration of a loop and transfers control to the next iteration. Both `EXIT` and `CONTINUE` have an optional `WHEN` clause, where you can specify a condition.

- **Sequential control statements**, which are not crucial to PL/SQL programming.
  The sequential control statements are `GOTO`, which goes to a specified statement, and `NULL`, which does nothing.

### 5.1 Conditional Selection Statements

The **conditional selection statements**, `IF` and `CASE`, run different statements for different data values.

The `IF` statement either runs or skips a sequence of one or more statements, depending on a condition. The `IF` statement has these forms:

- `IF THEN`
- `IF THEN ELSE`
- `IF THEN ELSIF`

The `CASE` statement chooses from a sequence of conditions, and runs the corresponding statement. The `CASE` statement has these forms:

- Simple `CASE` statement, which evaluates a single expression and compares it to several potential values.
- Searched `CASE` statement, which evaluates multiple conditions and chooses the first one that is true.

The `CASE` statement is appropriate when a different action is to be taken for each alternative.
5.1.1 IF THEN Statement

The IF THEN statement either runs or skips a sequence of one or more statements, depending on a condition.

The IF THEN statement has this structure:

```
IF condition THEN
    statements
END IF;
```

If the condition is true, the statements run; otherwise, the IF statement does nothing.

For complete syntax, see "IF Statement".

**Tip:**

Avoid clumsy IF statements such as:

```
IF new_balance < minimum_balance THEN
    overdrawn := TRUE;
ELSE
    overdrawn := FALSE;
END IF;
```

Instead, assign the value of the BOOLEAN expression directly to a BOOLEAN variable:

```
overdrawn := new_balance < minimum_balance;
```

A BOOLEAN variable is either TRUE, FALSE, or NULL. Do not write:

```
IF overdrawn = TRUE THEN
    RAISE insufficient_funds;
END IF;
```

Instead, write:

```
IF overdrawn THEN
    RAISE insufficient_funds;
END IF;
```

**Example 5-1   IF THEN Statement**

In this example, the statements between THEN and END IF run if and only if the value of sales is greater than quota+200.

```sql
DECLARE
    PROCEDURE p (
        sales  NUMBER,
        quota  NUMBER,
        emp_id NUMBER
    ) IS
    begin
        bonus  NUMBER := 0;
        updated VARCHAR2(3) := "No";
        BEGIN
            IF sales > (quota + 200) THEN
                ...  
            ELSE
                ...  
            END IF;
        END;
```
IF sales > (quota + 200) THEN
    bonus := (sales - quota)/4;

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;

    updated := 'Yes';
END IF;
DBMS_OUTPUT.PUT_LINE (
    'Table updated?  ' || updated || ', ' ||
    'bonus = ' || bonus || '.');
END p;
BEGIN
    p(10100, 10000, 120);
    p(10500, 10000, 121);
END;
/

Result:
Table updated?  No, bonus = 0.
Table updated?  Yes, bonus = 125.

5.1.2 IF THEN ELSE Statement

The IF THEN ELSE statement has this structure:

IF condition THEN
    statements
ELSE
    else_statements
END IF;

If the value of condition is true, the statements run; otherwise, the else_statements run.

IF statements can be nested, as in Example 5-3.

For complete syntax, see “IF Statement”.

Example 5-2  IF THEN ELSE Statement

In this example, the statement between THEN and ELSE runs if and only if the value of sales is greater than quota+200; otherwise, the statement between ELSE and END IF runs.

DECLARE
    PROCEDURE p (
        sales  NUMBER,
        quota  NUMBER,
        emp_id  NUMBER
    ) IS
        bonus  NUMBER := 0;
BEGIN
    IF sales > (quota + 200) THEN
        bonus := (sales - quota)/4;
    ELSE
        bonus := (sales - quota)/4;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('Table updated?  ' || updated || ', ' ||
        'bonus = ' || bonus || '.');
END p;
ELSE
    bonus := 50;
END IF;

DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

UPDATE employees
SET salary = salary + bonus
WHERE employee_id = emp_id;
END p;
BEGIN
    p(10100, 10000, 120);
    p(10500, 10000, 121);
END;
/

Result:
bonus = 50
bonus = 125

Example 5-3  Nested IF THEN ELSE Statements

DECLARE
    PROCEDURE p (sales NUMBER,
            quota NUMBER,
            emp_id NUMBER)
    IS
        bonus NUMBER := 0;
    BEGIN
        IF sales > (quota + 200) THEN
            bonus := (sales - quota)/4;
        ELSE
            IF sales > quota THEN
                bonus := 50;
            ELSE
                bonus := 0;
            END IF;
        END IF;
        DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);
        UPDATE employees
        SET salary = salary + bonus
        WHERE employee_id = emp_id;
    END p;
BEGIN
    p(10100, 10000, 120);
    p(10500, 10000, 121);
    p(9500, 10000, 122);
END;
/

Result:
bonus = 50
bonus = 125
bonus = 0
5.1.3 IF THEN ELSIF Statement

The IF THEN ELSIF statement has this structure:

```sql
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
  [ ELSIF condition_3 THEN
    statements_3
  ]...
[ ELSE
  else_statements
]
END IF;
```

The IF THEN ELSIF statement runs the first statements for which condition is true. Remaining conditions are not evaluated. If no condition is true, the else_statements run, if they exist; otherwise, the IF THEN ELSIF statement does nothing.

A single IF THEN ELSIF statement is easier to understand than a logically equivalent nested IF THEN ELSE statement:

```sql
-- IF THEN ELSIF statement
IF condition_1 THEN statements_1;
  ELSIF condition_2 THEN statements_2;
  ELSIF condition_3 THEN statement_3;
END IF;

-- Logically equivalent nested IF THEN ELSE statements
IF condition_1 THEN
  statements_1;
ELSE
  IF condition_2 THEN
    statements_2;
  ELSE
    IF condition_3 THEN
      statements_3;
    END IF;
  END IF;
END IF;
```

For complete syntax, see "IF Statement".

Example 5-4  IF THEN ELSIF Statement

In this example, when the value of sales is larger than 50000, both the first and second conditions are true. However, because the first condition is true, bonus is assigned the value 1500, and the second condition is never tested. After bonus is assigned the value 1500, control passes to the DBMS_OUTPUT.PUT_LINE invocation.

```sql
DECLARE
  PROCEDURE p (sales NUMBER)
  IS
    bonus NUMBER := 0;
    BEGIN
      IF sales > 50000 THEN
```

bonus := 1500;
ELSIF sales > 35000 THEN
    bonus := 500;
ELSE
    bonus := 100;
END IF;

DBMS_OUTPUT.PUT_LINE (  
    'Sales = ' || sales || ', bonus = ' || bonus || ' .'  
);
END p;
BEGIN
    p(55000);
p(40000);
p(30000);
END;
/

Result:
Sales = 55000, bonus = 1500.
Sales = 40000, bonus = 500.
Sales = 30000, bonus = 100.

Example 5-5  IF THEN ELSIF Statement Simulates Simple CASE Statement

This example uses an IF THEN ELSIF statement with many ELSIF clauses to compare a single value to many possible values. For this purpose, a simple CASE statement is clearer—see Example 5-6.

DECLARE
    grade CHAR(1);
BEGIN
    grade := 'B';

    IF grade = 'A' THEN
        DBMS_OUTPUT.PUT_LINE('Excellent');
    ELSIF grade = 'B' THEN
        DBMS_OUTPUT.PUT_LINE('Very Good');
    ELSIF grade = 'C' THEN
        DBMS_OUTPUT.PUT_LINE('Good');
    ELSIF grade = 'D' THEN
        DBMS_OUTPUT.PUT_LINE('Fair');
    ELSIF grade = 'F' THEN
        DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE
        DBMS_OUTPUT.PUT_LINE('No such grade');
    END IF;
END;
/

Result:
Very Good

5.1.4 Simple CASE Statement

The simple CASE statement has this structure:
CASE selector
  WHEN selector_value_1 THEN statements_1
  WHEN selector_value_2 THEN statements_2
  ...
  WHEN selector_value_n THEN statements_n
  [ ELSE
    else_statements ]
END CASE;

The selector is an expression (typically a single variable). Each selector_value can be either a literal or an expression. (For complete syntax, see "CASE Statement").

The simple CASE statement runs the first statements for which selector_value equals selector. Remaining conditions are not evaluated. If no selector_value equals selector, the CASE statement runs else_statements if they exist and raises the predefined exception CASE_NOT_FOUND otherwise.

Example 5-6 uses a simple CASE statement to compare a single value to many possible values. The CASE statement in Example 5-6 is logically equivalent to the IF THEN ELSIF statement in Example 5-5.

Note:

As in a simple CASE expression, if the selector in a simple CASE statement has the value NULL, it cannot be matched by WHEN NULL (see Example 3-51). Instead, use a searched CASE statement with WHEN condition IS NULL (see Example 3-53).

Example 5-6  Simple CASE Statement

DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/

Result:
Very Good

5.1.5 Searched CASE Statement

The searched CASE statement has this structure:
The searched `CASE` statement runs the first `statements` for which `condition` is true. Remaining conditions are not evaluated. If no `condition` is true, the `CASE` statement runs `else_statements` if they exist and raises the predefined exception `CASE_NOT_FOUND` otherwise. (For complete syntax, see "CASE Statement".)

The searched `CASE` statement in Example 5-7 is logically equivalent to the simple `CASE` statement in Example 5-6.

In both Example 5-7 and Example 5-6, the `ELSE` clause can be replaced by an `EXCEPTION` part. Example 5-8 is logically equivalent to Example 5-7.

### Example 5-7  Searched CASE Statement

```sql
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/`
5.2 LOOP Statements

Loop statements run the same statements iteratively with a series of different values.

A LOOP statement has three parts:

1. An iterand, also known as a loop variable, to pass values from the loop header to the loop body
2. Iteration controls to generate values for the loop
3. A loop body executed once for each value

```
loop_statement ::= [ iteration_scheme ] LOOP
    loop_body
END LOOP [ label ];
```

The loop statements are:

- Basic LOOP
- FOR LOOP
- Cursor FOR LOOP
- WHILE LOOP

The statements that exit a loop are:

- EXIT
- EXIT WHEN

The statements that exit the current iteration of a loop are:

- CONTINUE
- CONTINUE WHEN

EXIT, EXIT WHEN, CONTINUE, and CONTINUE WHEN can appear anywhere inside a loop, but not outside a loop. Oracle recommends using these statements instead of the GOTO statement, which can exit a loop or the current iteration of a loop by transferring control to a statement outside the loop.

A raised exception also exits a loop.

LOOP statements can be labeled, and LOOP statements can be nested. Labels are recommended for nested loops to improve readability. You must ensure that the label in the END LOOP statement matches the label at the beginning of the same loop statement (the compiler does not check).
5.2.1 Basic LOOP Statement

The basic LOOP statement has this structure:

```
[ label ] LOOP
    statements
END LOOP [ label ];
```

With each iteration of the loop, the `statements` run and control returns to the top of the loop. To prevent an infinite loop, a statement or raised exception must exit the loop.

5.2.2 FOR LOOP Statement Overview

The FOR LOOP statement runs one or more statements for each value of the loop index.

A FOR LOOP header specifies the iterator. The iterator specifies an iterand and the iteration controls. The iteration control provides a sequence of values to the iterand for access in the loop body. The loop body has the statements that are executed once for each value of the iterand.

The iteration controls available are:

- **Stepped Range** An iteration control that generates a sequence of stepped numeric values. When step is not specified, the counting control is a stepped range of type pls integer with a step of one.

- **Single Expression** An iteration control that evaluates a single expression.

- **Repeated Expression** An iteration control that repeatedly evaluates a single expression.

- **Values Of** An iteration control that generates all the values from a collection in sequence. The collection can be a vector valued expression, cursor, cursor variable, or dynamic SQL.
Indices Of An iteration control that generates all the indices from a collection in sequence. While all the collection types listed for values of are allowed, indices of is most useful when the collection is a vector variable.

Pairs Of An iteration control that generates all the index and value pairs from a collection. All of the collection types allowed for values of are allowed for pairs of. Pairs of iteration controls require two iterands.

Cursor An iteration control that generates all the records from a cursor, cursor variable, or dynamic SQL.

The FOR LOOP statement has this structure:

```plaintext
[ label ] for_loop_header
  statements
END LOOP [ label ];
```

for_loop_header ::= FOR iterator LOOP

iterator ::= iterand_decl [, iterand_decl] IN iteration_ctl_seq

iterand_decl ::= pls_identifier [ MUTABLE | IMMUTABLE ]
  [ constrained_type ]

iteration_ctl_seq ::= qual_iteration_ctl [,]...

qual_iteration_ctl ::= [ REVERSE ] iteration_control pred_clause_seq

iteration_control ::= stepped_control
  | single_expression_control
  | values_of_control
  | indices_of_control
  | pairs_of_control
  | cursor_control

pred_clause_seq ::= [ stopping_pred ] [ skipping_pred ]

stopping_pred ::= WHILE boolean_expression

skipping_pred ::= WHEN boolean_expression

stepped_control ::= lower_bound .. upper_bound [ BY step ]

single_expression_control ::= [ REPEAT ] expr

See Also:
"FOR LOOP Statement" for more information about syntax and semantics

5.2.2.1 FOR LOOP Iterand

The index or iterand of a FOR LOOP statement is implicitly or explicitly declared as a variable that is local to the loop.
The statements in the loop can read the value of the iterand, but cannot change it. Statements outside the loop cannot reference the iterand. After the `FOR` loop statement runs, the iterand is undefined. A loop iterand is sometimes called a loop counter.

**Example 5-9 FOR LOOP Statement Tries to Change Index Value**

In this example, the `FOR` loop statement tries to change the value of its index, causing an error.

```plsql
BEGIN
  FOR i IN 1..3 LOOP
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2;
    END IF;
  END LOOP;
END;
/
```

Result:

```
i := 2;
* PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored
```

**Example 5-10 Outside Statement References FOR LOOP Statement Index**

In this example, a statement outside the `FOR` loop statement references the loop index, causing an error.

```plsql
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

Result:

```
DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
* PLS-00201: identifier 'I' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

**Example 5-11 FOR LOOP Statement Index with Same Name as Variable**

If the index of a `FOR` loop statement has the same name as a variable declared in an enclosing block, the local implicit declaration hides the other declaration, as this example shows.

```plsql
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;
```

---

The statements in the loop can read the value of the iterand, but cannot change it. Statements outside the loop cannot reference the iterand. After the `FOR` loop statement runs, the iterand is undefined. A loop iterand is sometimes called a loop counter.

**Example 5-9 FOR LOOP Statement Tries to Change Index Value**

In this example, the `FOR` loop statement tries to change the value of its index, causing an error.

```plsql
BEGIN
  FOR i IN 1..3 LOOP
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2;
    END IF;
  END LOOP;
END;
/
```

Result:

```
i := 2;
* PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored
```

**Example 5-10 Outside Statement References FOR LOOP Statement Index**

In this example, a statement outside the `FOR` loop statement references the loop index, causing an error.

```plsql
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

Result:

```
DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
* PLS-00201: identifier 'I' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

**Example 5-11 FOR LOOP Statement Index with Same Name as Variable**

If the index of a `FOR` loop statement has the same name as a variable declared in an enclosing block, the local implicit declaration hides the other declaration, as this example shows.

```plsql
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;
```
Example 5-12    FOR LOOP Statement References Variable with Same Name as Index

This example shows how to change Example 5-11 to allow the statement inside the loop to reference the variable declared in the enclosing block.

```
<<main>>  -- Label block.
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('local: ' || TO_CHAR(i) || ', global: ' || TO_CHAR(main.i) -- Qualify reference with block label.);
  END LOOP;
END main;
```

Result:
local: 1, global: 5
local: 2, global: 5
local: 3, global: 5

Example 5-13    Nested FOR LOOP Statements with Same Index Name

In this example, the indexes of the nested FOR LOOP statements have the same name. The inner loop references the index of the outer loop by qualifying the reference with the label of the outer loop. For clarity only, the inner loop also qualifies the reference to its own index with its own label.

```
BEGIN
  <<outer_loop>>
  FOR i IN 1..3 LOOP
    <<inner_loop>>
    FOR i IN 1..3 LOOP
      IF outer_loop.i = 2 THEN
        DBMS_OUTPUT.PUT_LINE ('outer: ' || TO_CHAR(outer_loop.i) || ' inner: ' || TO_CHAR(inner_loop.i));
      END IF;
    END LOOP inner_loop;
  END LOOP outer_loop;
END;
```

Result:
5.2.2.2 Iterand Mutability

The mutability property of an iterand determines whether or not it can be assigned in the loop body.

If all iteration controls specified in an iterator are cursor controls, the iterand is mutable by default. Otherwise, the iterand is immutable. The default mutability property of an iterand can be changed in the iterand declaration by specifying the `MUTABLE` or `IMMUTABLE` keyword after the iterand variable.

Considerations when declaring an iterand mutable:

- Any modification to the iterand for values of iteration control or the values iterand for a pairs of iteration control will not affect the sequence of values produced by that iteration control.
- Any modification to the iterand for stepped range iteration control or repeated single expression iteration control will likely affect the behaviour of that control and the sequence of values it produces.
- When the PL/SQL compiler can determine that making an iterand mutable may adversely affect runtime performance, it may report a warning.

5.2.2.3 Multiple Iteration Controls

Multiple iteration controls may be chained together by separating them with commas.

Each iteration control has a set of controlling expressions (some controls have none) that are evaluated once when the control starts. Evaluation of these expressions or conversion of the evaluated values to the iterand type may raise exceptions. In such cases, the loop is abandoned and normal exception handling occurs. The iterand is accessible in the list of iteration controls. It is initially set to the default value for its type. If that type has a not null constraint, any reference to the iterand in the controlling expressions for the first iteration control will produce a semantic error because the iterand cannot be implicitly initialized. When an iteration control is exhausted, the iterand contains the final value assigned to it while processing that iteration control and execution advances to the next iteration control. If no values are assigned to the iterand by an iteration control, it retains the value it had prior to the start of that iteration control. If the final value of a mutable iterand is modified in the loop body, that modified value will be visible when evaluating the control expressions from the following iteration control.

Expanding Multiple Iteration Controls Into PL/SQL

The first iteration control is initialized. The loop for the first iteration control is evaluated. The controlling expressions from the next iteration control is evaluated. The loop for the second iteration control is evaluated. Each iteration control and loop is evaluated in turn until there are no more iteration controls.

Example 5-14 Using Multiple Iteration Controls

This example shows the loop variable / taking the value three iteration controls in succession. The value of the iterator is printed for demonstration purpose. It shows
that when a loop control is exhausted, the next iteration control begins. When the last iteration control is exhausted, the loop is complete.

```plsql
DECLARE
  i PLS_INTEGER;
BEGIN
  FOR i IN 1..3, REVERSE i+1..i+10, 51..55 LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
/
```

5.2.2.4 Stepped Range Iteration Controls

Stepped range iteration controls generate a sequence of numeric values. Controlling expressions are the lower bound, upper bound, and step.

```
stepped_control ::= [ REVERSE ] lower_bound..upper_bound [ BY step ]
lower_bound ::= numeric_expression
upper_bound ::= numeric_expression
step ::= numeric_expression
```

Expanding Stepped Range Iteration Controls Into PL/SQL

When the iteration control is initialized, each controlling expression is evaluated and converted to the type of the iterand. Step must have a strictly positive numeric value. If any exception occurs while evaluating the controlling expressions, the loop is abandoned and normal exception handling occurs. When no step is specified, its value is one. The values generated by a stepped range iteration control go from lower bound to upper bound by step. When REVERSE is specified the values are decremented from the upper bound to lower bound by step. If the iterand has a floating point type, some combinations of loop control values may create an infinite loop because of rounding errors. No semantic or dynamic analysis will report this. When the iterand is mutable and is modified in the loop body, the modified value is used for the increment and loop exhaustion test in the next iterand update. This may change the sequence of values processed by the loop.
Example 5-15  FOR LOOP Statements Range Iteration Control

In this example, *iterand* is *i*, *lower_bound* is 1, and *upper_bound* is 3. The loop prints the numbers from 1 to 3.

BEGIN
   FOR i IN 1..3 LOOP
      DBMS_OUTPUT.PUT_LINE (i);
   END LOOP;
END;
/

Result:

1
2
3

Example 5-16  Reverse FOR LOOP Statements Range Iteration Control

The FOR LOOP statement in this example prints the numbers from 3 to 1. The loop variable *i* is implicitly declared as a PLS_INTEGER (the default for counting and indexing loops).

BEGIN
   FOR i IN REVERSE 1..3 LOOP
      DBMS_OUTPUT.PUT_LINE (i);
   END LOOP;
END;
/

Result:

3
2
1

Example 5-17  Stepped Range Iteration Controls

This example shows a loop variable *n* declared explicitly as a NUMBER(5,1). The increment for the counter is 0.5.

BEGIN
   FOR n NUMBER(5,1) IN 1.0 .. 3.0 BY 0.5 LOOP
      DBMS_OUTPUT.PUT_LINE(n);
   END LOOP;
END;
/

Result:

1
1.5
2
2.5
3
Example 5-18  STEP Clause in FOR LOOP Statement

In this example, the FOR LOOP effectively increments the index by five.

BEGIN
    FOR i IN 5..15 BY 5 LOOP
        DBMS_OUTPUT.PUT_LINE (i);
    END LOOP;
END;

Result:
5
10
15

Example 5-19  Simple Step Filter Using FOR LOOP Stepped Range Iterator

This example illustrates a simple step filter. This filter is used in signal processing and other reduction applications. The predicate specifies that every Kth element of the original collection is passed to the collection being created.

FOR i IN start..finish LOOP
    IF (i - start) MOD k = 0 THEN
        newcol(i) := col(i)
    END IF;
END LOOP;

You can implement the step filter using a stepped range iterator.

FOR i IN start..finish BY k LOOP
    newcol(i) := col(i)
END LOOP;

You can implement the same filter by creating a new collection using a stepped iteration control embedded in a qualified expression.

newcol := col_t(FOR I IN start..finish BY k => col(i));

5.2.2.5 Single Expression Iteration Controls

A single expression iteration control generates a single value.

single_expression_control ::= [ REPEAT ] expr

A single expression iteration control has no controlling expressions.

When the iterand is mutable, changes made to it in the loop body will be seen when reevaluating the expression in the repeat form.

Expanding Single Expression Iteration Controls Into PL/SQL

The expression is evaluated, converted to the iterand type to create the next value. Any stopping predicate is evaluated. If it fails to evaluate to TRUE, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to TRUE, skip the next step. Evaluate the loop body. If REPEAT is specified, evaluate the expression again. Otherwise, the iteration control is exhausted.
Example 5-20  Single Expression Iteration Control

This example shows the loop body being executed once.

```
BEGIN
  FOR i IN 1 LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
/
```

Result:
1

This example shows the iterand starting with 1, then \( i^2 \) is evaluated repeatedly until the stopping predicate evaluates to true.

```
BEGIN
  FOR i IN 1, \( \text{REPEAT} \) \( i^2 \) \( \text{WHILE} \) i < 100 LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
/
```

Result:
1
2
4
8
16
32
64

5.2.2.6 Collection Iteration Controls

VALUES OF, INDICES OF, and PAIRS OF iteration controls generate sequences of values for an iterand derived from a collection.

```
collection_iteration_control ::= values_of_control
| indices_of_control
| pairs_of_control

values_of_control ::= VALUES OF expr
| VALUES OF (cursor_object)
| VALUES OF (sql_statement)
| VALUES OF cursor_variable
| VALUES OF (dynamic_sql)

indices_of_control ::= INDICES OF expr
| INDICES OF (cursor_object)
| INDICES OF (sql_statement)
| INDICES OF cursor_variable
| INDICES OF (dynamic_sql)

pairs_of_control ::= PAIRS OF expr
| PAIRS OF (cursor_object)
```
The collection itself is the controlling expression. The collection can be a vector value expression, a cursor object, cursor variable, or dynamic SQL. If a collection is null, it is treated as if it were defined and empty.

A cursor_object is an explicit PL/SQL cursor object. A sql_statement is an implicit PL/SQL cursor object created for a SQL statement specified directly in the iteration control. A cursor_variable is a PL/SQL REF CURSOR object.

When the iterand for a values of iteration control or the value iterand for a VALUES OF iteration control is modified in the loop body, those changes have no effect on the next value generated by the iteration control.

If the collection is modified in the loop body, behavior is unspecified. If a cursor variable is accessed other than through the iterand during execution of the loop body, the behavior is unspecified. Most INDICES OF iteration controls produce a numeric sequence unless the collection is a vector variable.

Expanding VALUES OF Iteration Controls into PL/SQL

The collection is evaluated and assigned to a vector. If the collection is empty, the iteration control is exhausted. A temporary hidden index is initialized with the index of the first element (or last element if REVERSE is specified). A value is fetched from the collection based on the temporary index to create the next value for the iterand. Any stopping predicate is evaluated. If it fails to evaluate to TRUE, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to TRUE, skip the next step. Evaluate the loop body. Advance the index temporary to the index of the next element in the vector (previous element for REVERSE). Determine the next value and reiterate with each iterand value until the iteration control is exhausted.

Example 5-21 VALUES OF Iteration Control

This example prints the values from the collection vec: [11, 10, 34]. The iterand values of the iteration control variable i is the value of the first element in the vector, then the next element, and the last one.

```sql
DECLARE
    TYPE intvec_t IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
    vec intvec_t := intvec_t(3 => 10, 1 => 11, 100 => 34);
BEGIN
    FOR i IN VALUES OF vec LOOP
        DBMS_OUTPUT.PUT_LINE(i);
    END LOOP;
END;
/ 11 10 34
```
Expanding INDICES OF Iteration Controls into PL/SQL

The collection is evaluated and assigned to a vector. If the collection is empty, the iteration control is exhausted. The next value for the iterand is determined (index of the first element or last element if `REVERSE` is specified). The next value is assigned to the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. The loop body is evaluated. Advance the iterand to the next value which is the index of the next element in the vector (previous element for `REVERSE`). Reiterate with each iterand value (assigned the index of the next or previous element) until the iteration control is exhausted.

**Example 5-22  INDICES OF Iteration Control**

This example prints the indices of the collection `vec : [1, 3, 100]`. The iterand values of the iteration control variable `i` is the index of the first element in the vector, then the next element, and the last one.

```plsql
DECLARE
    TYPE intvec_t IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
    vec  intvec_t := intvec_t(3 => 10, 1 => 11, 100 => 34);
    FOR i IN INDICES OF vec LOOP
        DBMS_OUTPUT.PUT_LINE(i);
    END LOOP;
END;
/
```

Result:

```
1
3
100
```

Expanding PAIRS OF Iteration Controls into PL/SQL

The collection is evaluated and assigned to a vector. If the collection is empty, the iteration control is exhausted. The next index value for the iterand is determined (index of the first element or last element if `REVERSE` is specified). The next value of the element indexed by the next value is assigned to the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. The loop body is evaluated. Advance the iterand to the next index value which is the index of the next element in the vector (previous element for `REVERSE`). Reiterate with each iterand value until the iteration control is exhausted.

**Example 5-23  PAIRS OF Iteration Control**

This example inverts a collection `vec` into a collection `result` and prints the resulting index value pairs (10 => 3, 11 => 1, 34 => 100).

```plsql
DECLARE
    TYPE intvec_t IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
    vec  intvec_t := intvec_t(3 => 10, 1 => 11, 100 => 34);
    result intvec_t;
BEGIN
    result := intvec_t(FOR i,j IN PAIRS OF vec INDEX j => i);
    FOR i,j IN PAIRS OF result LOOP
        DBMS_OUTPUT.PUT_LINE(i || '=>'|| j);
    END LOOP;
END;
```
5.2.2.7 Cursor Iteration Controls

Cursor iteration controls generate the sequence of records returned by an explicit or implicit cursor.

The cursor definition is the controlling expression. You cannot use `REVERSE` with a cursor iteration control.

```
cursor_iteration__control ::=  { cursor_object
                                 | sql_statement
                                 | cursor_variable
                                 | dynamic_sql }
```

A `cursor_object` is an explicit PL/SQL cursor object. A `sql_statement` is an implicit PL/SQL cursor object created for a SQL statement specified directly in the iteration control. A `cursor_variable` is a PL/SQL `REF CURSOR` object. A cursor iteration control is equivalent to a `VALUES OF` iteration control whose collection is a cursor. When the iterand is modified in the loop body, it has no effect on the next value generated by the iteration control. When the collection is a cursor variable, it must be open when the iteration control is encountered or an exception will be raised. It remains open when the iteration control is exhausted. If the cursor variable is accessed other than through the iterand during execution of the loop body, the behavior is unspecified.

Expanding Cursor Iteration Controls Into PL/SQL

The cursor is evaluated to create a vector of iterands. If the vector is empty, the iteration control is exhausted. A value is fetched in the vector to create the next value for the iterand. Any stopping predicate is evaluated. If it fails to evaluate to `TRUE`, the iteration control is exhausted. Any skipping predicate is evaluated. If it fails to evaluate to `TRUE`, skip the next step. Evaluate the loop body. Reiterate the same with each iterand value fetched until the iteration control is exhausted.

Example 5-24  Cursor Iteration Controls

This example creates an associative array mapping of id to data from table t.

```
OPEN c FOR SELECT id, data FROM T;
FOR r rec_t IN c LOOP
    result(r.id) := r.data;
END LOOP;
CLOSE c;
```

5.2.2.8 Using Dynamic SQL in Iteration Controls

```
dynamic_sql ::= EXECUTE IMMEDIATE dynamic_sql_stmt [ using_clause ]
```
using_clause ::= USING [ [ IN ] (bind_argument [,])+ ]

Dynamic SQL may be used in a cursor or collection iteration control. Such a construct cannot provide a default type; if it is used as the first iteration control, an explicit type must be specified for the iterand (or for the value iterand for a pairs of control). The using_clause is the only clause allowed. No INTO or dynamic returning clauses may be used. If the specified SQL statement is a kind that cannot return any rows, a runtime error will be reported similar to that reported if a bulk collect into or into clause were specified on an ordinary execute immediate statement.

Example 5-25 Using Dynamic SQL As An Iteration Control

This example shows the iteration control generates all the records from a dynamic SQL. It prints the last_name and employee_id of all employees having an employee_id less than 103. It executes the loop body when the stopping predicate is TRUE.

DECLARE
  cursor_str VARCHAR2(500) := 'SELECT last_name, employee_id FROM hr.employees ORDER BY last_name';
  TYPE rec_t IS RECORD (last_name VARCHAR2(25),
                       employee_id NUMBER);
BEGIN
  FOR r rec_t IN VALUES OF (EXECUTE IMMEDIATE cursor_str) WHEN r.employee_id < 103
  LOOP
    DBMS_OUTPUT.PUT_LINE(r.last_name || ', ' || r.employee_id);
  END LOOP;
END;
/

Result:

De Haan, 102
King, 100
Kochhar, 101

Example 5-26 Using Dynamic SQL As An Iteration Control In a Qualified Expression

v := vec_rec_t( FOR r rec_t IN (EXECUTE IMMEDIATE query_var) SEQUENCE => r);

5.2.2.9 Stopping and Skipping Predicate Clauses

A stopping predicate clause can cause the iteration control to be exhausted while a skipping predicate clause can cause the loop body to be skipped for some values.

The expressions in these predicate clauses are not controlling expressions.

A stopping predicate clause can cause the iteration control to be exhausted. The boolean_expression is evaluated at the beginning of each iteration of the loop. If it fails to evaluate to TRUE, the iteration control is exhausted.

A skipping predicate clause can cause the loop body to be skipped for some values. The boolean_expression is evaluated. If it fails to evaluate to TRUE, the iteration control skips to the next value.

pred_clause_seq ::= [stopping_pred] [skipping_pred]
stopping_pred ::= WHILE boolean_expression

skipping_pred ::= WHEN boolean_expression

Example 5-27 Using FOR LOOP Stopping Predicate Clause

This example shows an iteration control with a WHILE stopping predicate clause. The iteration control is exhausted if the stopping predicate does not evaluate to TRUE.

BEGIN
  FOR power IN 1, REPEAT power*2 WHILE power <= 64 LOOP
    DBMS_OUTPUT.PUT_LINE(power);
    END LOOP;
END;
/

Result:
1
2
4
8
16
32
64

Example 5-28 Using FOR LOOP Skipping Predicate Clause

This example shows an iteration control with a WHEN skipping predicate clause. If the skipping predicate does not evaluate to TRUE, the iteration control skips to the next value.

BEGIN
  FOR power IN 2, REPEAT power*2 WHILE power <= 64 WHEN MOD(power, 32)= 0 LOOP
    DBMS_OUTPUT.PUT_LINE(power);
    END LOOP;
END;
/

Result:
2
32
64

5.2.3 WHILE LOOP Statement

The WHILE LOOP statement runs one or more statements while a condition is true.

It has this structure:

[ label ] WHILE condition LOOP
  statements
END LOOP [ label ];

If the condition is true, the statements run and control returns to the top of the loop, where condition is evaluated again. If the condition is not true, control transfers to the statement after the WHILE LOOP statement. To prevent an infinite loop, a statement inside the loop must make the condition false or null. For complete syntax, see "WHILE LOOP Statement".
An EXIT, EXIT WHEN, CONTINUE, or CONTINUE WHEN in the statements can cause the loop or the current iteration of the loop to end early.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests a condition at the bottom of the loop instead of at the top, so that the statements run at least once. To simulate this structure in PL/SQL, use a basic LOOP statement with an EXIT WHEN statement:

```
LOOP
  statements
  EXIT WHEN condition;
END LOOP;
```

5.3 Sequential Control Statements

Unlike the IF and LOOP statements, the sequential control statements GOTO and NULL are not crucial to PL/SQL programming.

The GOTO statement, which goes to a specified statement, is seldom needed. Occasionally, it simplifies logic enough to warrant its use.

The NULL statement, which does nothing, can improve readability by making the meaning and action of conditional statements clear.

Topics

- GOTO Statement
- NULL Statement

5.3.1 GOTO Statement

The GOTO statement transfers control to a label unconditionally. The label must be unique in its scope and must precede an executable statement or a PL/SQL block. When run, the GOTO statement transfers control to the labeled statement or block.

For GOTO statement restrictions, see "GOTO Statement".

Use GOTO statements sparingly—overusing them results in code that is hard to understand and maintain. Do not use a GOTO statement to transfer control from a deeply nested structure to an exception handler. Instead, raise an exception. For information about the PL/SQL exception-handling mechanism, see PL/SQL Error Handling.

The GOTO statement transfers control to the first enclosing block in which the referenced label appears.

5.3.2 NULL Statement

The NULL statement only passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).

Some uses for the NULL statement are:

- To provide a target for a GOTO statement
- To improve readability by making the meaning and action of conditional statements clear
• To create placeholders and stub subprograms
• To show that you are aware of a possibility, but that no action is necessary

Note:
Using the NULL statement might raise an unreachable code warning if warnings are enabled. For information about warnings, see "Compile-Time Warnings".

Example 5-29  NULL Statement Showing No Action
The NULL statement emphasizes that only salespersons receive commissions.

DECLARE
    v_job_id  VARCHAR2(10);
    v_emp_id  NUMBER(6) := 110;
BEGIN
    SELECT job_id INTO v_job_id
    FROM employees
    WHERE employee_id = v_emp_id;

    IF v_job_id = 'SA_REP' THEN
        UPDATE employees
        SET commission_pct = commission_pct * 1.2;
    ELSE
        NULL;  -- Employee is not a sales rep
    END IF;
END;
/

Example 5-30  NULL Statement as Placeholder During Subprogram Creation
The NULL statement lets you compile this subprogram and fill in the real body later.

CREATE OR REPLACE PROCEDURE award_bonus (
    emp_id NUMBER,
    bonus NUMBER
) AUTHID DEFINER AS
BEGIN    -- Executable part starts here
    NULL;  -- Placeholder
    -- (raises "unreachable code" if warnings enabled)
END award_bonus;
/

Example 5-31  NULL Statement in ELSE Clause of Simple CASE Statement
The NULL statement shows that you have chosen to take no action for grades other than A, B, C, D, and F.

CREATE OR REPLACE PROCEDURE print_grade (
    grade CHAR
) AUTHID DEFINER AS
BEGIN
    CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    ELSE
        NULL;  -- Employee is not a sales rep
    END CASE;
END print_grade;
/
WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
ELSE NULL;
END CASE;
END;
/
BEGIN
  print_grade('A');
  print_grade('S');
END;
/

Result:

Excellent
PL/SQL Collections and Records

PL/SQL lets you define two kinds of composite data types: collection and record.

A **composite data type** stores values that have internal components. You can pass entire composite variables to subprograms as parameters, and you can access internal components of composite variables individually. Internal components can be either scalar or composite. You can use scalar components wherever you can use scalar variables. You can use composite components wherever you can use composite variables of the same type.

**Note:**

If you pass a composite variable as a parameter to a remote subprogram, then you must create a redundant loop-back **DATABASE LINK**, so that when the remote subprogram compiles, the type checker that verifies the source uses the same definition of the user-defined composite variable type as the invoker uses.

In a **collection**, the internal components always have the same data type, and are called **elements**. You can access each element of a collection variable by its unique index, with this syntax: `variable_name(index)`. To create a collection variable, you either define a collection type and then create a variable of that type or use `%TYPE`.

In a **record**, the internal components can have different data types, and are called **fields**. You can access each field of a record variable by its name, with this syntax: `variable_name.field_name`. To create a record variable, you either define a **RECORD** type and then create a variable of that type or use `%ROWTYPE` or `%TYPE`.

You can create a collection of records, and a record that contains collections.

**Collection Topics**

- Collection Types
- Associative Arrays
- Varrays (Variable-Size Arrays)
- Nested Tables
- Collection Constructors
- Qualified Expressions Overview
- Assigning Values to Collection Variables
- Multidimensional Collections
- Collection Comparisons
- Collection Methods
6.1 Collection Types

PL/SQL has three collection types—associative array, VARRAY (variable-size array), and nested table.

Table 6-1 summarizes their similarities and differences.

<table>
<thead>
<tr>
<th>Collection Type</th>
<th>Number of Elements</th>
<th>Index Type</th>
<th>Dense or Sparse</th>
<th>Uninitialized Status</th>
<th>Where Defined</th>
<th>Can Be ADT Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associative array (or index-by table)</td>
<td>Unspecified</td>
<td>String or PLS_INTEGER</td>
<td>Either</td>
<td>Empty</td>
<td>In PL/SQL block or package</td>
<td>No</td>
</tr>
</tbody>
</table>

Note: The components of an explicitly listed composite data structure (such as a collection constructor or record initializer) can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.
Table 6-1  (Cont.) PL/SQL Collection Types

<table>
<thead>
<tr>
<th>Collection Type</th>
<th>Number of Elements</th>
<th>Index Type</th>
<th>Dense or Sparse</th>
<th>Uninitialized Status</th>
<th>Where Defined</th>
<th>Can Be ADT Attribute Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARRAY (variable-size array)</td>
<td>Specified</td>
<td>Integer</td>
<td>Always dense</td>
<td>Null</td>
<td>In PL/SQL block or package or at schema level</td>
<td>Only if defined at schema level</td>
</tr>
<tr>
<td>Nested table</td>
<td>Unspecified</td>
<td>Integer</td>
<td>Starts dense, can become sparse</td>
<td>Null</td>
<td>In PL/SQL block or package or at schema level</td>
<td>Only if defined at schema level</td>
</tr>
</tbody>
</table>

**Number of Elements**

If the number of elements is specified, it is the maximum number of elements in the collection. If the number of elements is unspecified, the maximum number of elements in the collection is the upper limit of the index type.

**Dense or Sparse**

A **dense collection** has no gaps between elements—every element between the first and last element is defined and has a value (the value can be NULL unless the element has a NOT NULL constraint). A **sparse collection** has gaps between elements.

**Uninitialized Status**

An **empty collection** exists but has no elements. To add elements to an empty collection, invoke the EXTEND method (described in "EXTEND Collection Method").

A **null collection** (also called an **atomically null collection**) does not exist. To change a null collection to an existing collection, you must initialize it, either by making it empty or by assigning a non-NULL value to it (for details, see "Collection Constructors" and "Assigning Values to Collection Variables"). You cannot use the EXTEND method to initialize a null collection.

**Where Defined**

A collection type defined in a PL/SQL block is a **local type**. It is available only in the block, and is stored in the database only if the block is in a standalone or package subprogram. (Standalone and package subprograms are explained in "Nested, Package, and Standalone Subprograms").

A collection type defined in a package specification is a **public item**. You can reference it from outside the package by qualifying it with the package name (package_name.type_name). It is stored in the database until you drop the package. (Packages are explained in PL/SQL Packages.)

A collection type defined at schema level is a **standalone type**. You create it with the "CREATE TYPE Statement". It is stored in the database until you drop it with the "DROP TYPE Statement".
A collection type defined in a package specification is incompatible with an identically defined local or standalone collection type (see Example 6-36 and Example 6-37).

Can Be ADT Attribute Data Type

To be an ADT attribute data type, a collection type must be a standalone collection type. For other restrictions, see Restrictions on datatype.

Translating Non-PL/SQL Composite Types to PL/SQL Composite Types

If you have code or business logic that uses another language, you can usually translate the array and set types of that language directly to PL/SQL collection types. For example:

<table>
<thead>
<tr>
<th>Non-PL/SQL Composite Type</th>
<th>Equivalent PL/SQL Composite Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash table</td>
<td>Associative array</td>
</tr>
<tr>
<td>Unordered table</td>
<td>Associative array</td>
</tr>
<tr>
<td>Set</td>
<td>Nested table</td>
</tr>
<tr>
<td>Bag</td>
<td>Nested table</td>
</tr>
<tr>
<td>Array</td>
<td>VARRAY</td>
</tr>
</tbody>
</table>

See Also:

Oracle Database SQL Language Reference for information about the CAST function, which converts one SQL data type or collection-typed value into another SQL data type or collection-typed value.

6.2 Associative Arrays

An associative array (formerly called PL/SQL table or index-by table) is a set of key-value pairs. Each key is a unique index, used to locate the associated value with the syntax `variable_name(index)`.

The data type of index can be either a string type (`VARCHAR2, VARCHAR, STRING, or LONG`) or PLS_INTEGER. Indexes are stored in sort order, not creation order. For string types, sort order is determined by the initialization parameters `NLS_SORT` and `NLS_COMP`.

Like a database table, an associative array:

- Is empty (but not null) until you populate it
- Can hold an unspecified number of elements, which you can access without knowing their positions

Unlike a database table, an associative array:
• Does not need disk space or network operations
• Cannot be manipulated with DML statements

Topics
• Declaring Associative Array Constants
• NLS Parameter Values Affect Associative Arrays Indexed by String
• Appropriate Uses for Associative Arrays

See Also:
• Table 6-1 for a summary of associative array characteristics
• "assoc_array_type_def ::=" for the syntax of an associative array type definition

Example 6-1  Associative Array Indexed by String
This example defines a type of associative array indexed by string, declares a variable of that type, populates the variable with three elements, changes the value of one element, and prints the values (in sort order, not creation order). (FIRST and NEXT are collection methods, described in "Collection Methods".)

Live SQL:
You can view and run this example on Oracle Live SQL at Associative Array Indexed by String

DECLARE
  -- Associative array indexed by string:
  TYPE population IS TABLE OF NUMBER  -- Associative array type
    INDEX BY VARCHAR2(64);            -- indexed by string
  city_population  population;        -- Associative array variable
  i  VARCHAR2(64);                    -- Scalar variable
BEGIN
  -- Add elements (key-value pairs) to associative array:
  city_population('Smallville')  := 2000;
  city_population('Midland')     := 750000;
  city_population('Megalopolis') := 1000000;
  -- Change value associated with key 'Smallville':
  city_population('Smallville') := 2001;
  -- Print associative array:
  i := city_population.FIRST;  -- Get first element of array
WHILE i IS NOT NULL LOOP
   DBMS_Output.PUT_LINE
      ('Population of ' || i || ' is ' || city_population(i));
   i := city_population.NEXT(i);  -- Get next element of array
END LOOP;
/

Result:
Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001

Example 6-2  Function Returns Associative Array Indexed by PLS_INTEGER
This example defines a type of associative array indexed by PLS_INTEGER and a
function that returns an associative array of that type.

DECLARE
   TYPE sum_multiples IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
   n  PLS_INTEGER := 5;   -- number of multiples to sum for display
   sn PLS_INTEGER := 10;  -- number of multiples to sum
   m  PLS_INTEGER := 3;   -- multiple
   FUNCTION get_sum_multiples (multiple IN PLS_INTEGER,
                                 num      IN PLS_INTEGER
                           ) RETURN sum_multiples
   IS
      s  sum_multiples;
   BEGIN
      FOR i IN 1..num LOOP
         s(i) := multiple * ((i * (i + 1)) / 2);  -- sum of multiples
      END LOOP;
      RETURN s;
   END get_sum_multiples;
BEGIN
   DBMS_OUTPUT.PUT_LINE (
      'Sum of the first ' || TO_CHAR(n) || ' multiples of ' ||
      TO_CHAR(m) || ' is ' || TO_CHAR(get_sum_multiples (m, sn)(n))
   );
END;
/

Result:
Sum of the first 5 multiples of 3 is 45
6.2.1 Declaring Associative Array Constants

When declaring an associative array constant, you must create a function that populates the associative array with its initial value and then invoke the function in the constant declaration.

For information about constructors, see "Collection Constructors".

Example 6-3  Declaring Associative Array Constant

In this example, the function does for the associative array what a constructor does for a varray or nested table.

```
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
  TYPE My_AA IS TABLE OF VARCHAR2(20) INDEX BY PLS_INTEGER;
  FUNCTION Init_My_AA RETURN My_AA;
END My_Types;
/
CREATE OR REPLACE PACKAGE BODY My_Types IS
  FUNCTION Init_My_AA RETURN My_AA IS
    Ret My_AA;
    BEGIN
      Ret(-10) := '-ten';
      Ret(0) := 'zero';
      Ret(1) := 'one';
      Ret(2) := 'two';
      Ret(3) := 'three';
      Ret(4) := 'four';
      Ret(9) := 'nine';
      RETURN Ret;
    END Init_My_AA;
END My_Types;
/
DECLARE
  v CONSTANT My_Types.My_AA := My_Types.Init_My_AA();
BEGIN
  DECLARE
    Idx PLS_INTEGER := v.FIRST();
    BEGIN
      WHILE Idx IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(Idx, '999')||LPAD(v(Idx), 7));
        Idx := v.NEXT(Idx);
      END LOOP;
    END;
  END;
END;
/
Result:
  -10   -ten
  0     zero
```
6.2.2 NLS Parameter Values Affect Associative Arrays Indexed by String

National Language Support (NLS) parameters such as NLS_SORT, NLS_COMP, and NLS_DATE_FORMAT affect associative arrays indexed by string.

Topics

• Changing NLS Parameter Values After Populating Associative Arrays
• Indexes of Data Types Other Than VARCHAR2
• Passing Associative Arrays to Remote Databases

See Also:

Oracle Database Globalization Support Guide for information about linguistic sort parameters

6.2.2.1 Changing NLS Parameter Values After Populating Associative Arrays

The initialization parameters NLS_SORT and NLS_COMP determine the storage order of string indexes of an associative array.

If you change the value of either parameter after populating an associative array indexed by string, then the collection methods FIRST, LAST, NEXT, and PRIOR might return unexpected values or raise exceptions. If you must change these parameter values during your session, restore their original values before operating on associative arrays indexed by string.

See Also:

Collection Methods for more information about FIRST, LAST, NEXT, and PRIOR

6.2.2.2 Indexes of Data Types Other Than VARCHAR2

In the declaration of an associative array indexed by string, the string type must be VARCHAR2 or one of its subtypes.

However, you can populate the associative array with indexes of any data type that the TO_CHAR function can convert to VARCHAR2.
If your indexes have data types other than VARCHAR2 and its subtypes, ensure that these indexes remain consistent and unique if the values of initialization parameters change. For example:

- Do not use `TO_CHAR(SYSDATE)` as an index.
  If the value of `NLS_DATE_FORMAT` changes, then the value of `(TO_CHAR(SYSDATE))` might also change.
- Do not use different `NVARCHAR2` indexes that might be converted to the same `VARCHAR2` value.
- Do not use `CHAR` or `VARCHAR2` indexes that differ only in case, accented characters, or punctuation characters.
  If the value of `NLS_SORT` ends in `_CI` (case-insensitive comparisons) or `_AI` (accent- and case-insensitive comparisons), then indexes that differ only in case, accented characters, or punctuation characters might be converted to the same value.

See Also:

*Oracle Database SQL Language Reference* for more information about `TO_CHAR`

6.2.2.3 Passing Associative Arrays to Remote Databases

If you pass an associative array as a parameter to a remote database, and the local and the remote databases have different `NLS_SORT` or `NLS_COMP` values, then:

- The collection method `FIRST`, `LAST`, `NEXT` or `PRIOR` (described in "Collection Methods") might return unexpected values or raise exceptions.
- Indexes that are unique on the local database might not be unique on the remote database, raising the predefined exception `VALUE_ERROR`.

6.2.3 Appropriate Uses for Associative Arrays

An associative array is appropriate for:

- A relatively small lookup table, which can be constructed in memory each time you invoke the subprogram or initialize the package that declares it
- Passing collections to and from the database server

Declare formal subprogram parameters of associative array types. With Oracle Call Interface (OCI) or an Oracle precompiler, bind the host arrays to the corresponding actual parameters. PL/SQL automatically converts between host arrays and associative arrays indexed by `PLS_INTEGER`.

Note:

You cannot bind an associative array indexed by `VARCHAR`. 
Note:

You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone subprogram, you must declare the type of that variable in a package specification. Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and the invoking subprogram or anonymous block (which declares and passes the variable of that type). See Example 11-2.

Tip:

The most efficient way to pass collections to and from the database server is to use associative arrays with the **FORALL** statement or **BULK COLLECT** clause. For details, see "FORALL Statement" and "BULK COLLECT Clause".

An associative array is intended for temporary data storage. To make an associative array persistent for the life of a database session, declare it in a package specification and populate it in the package body.

### 6.3 Varrays (Variable-Size Arrays)

A **varray (variable-size array)** is an array whose number of elements can vary from zero (empty) to the declared maximum size.

To access an element of a varray variable, use the syntax `variable_name(index)`. The lower bound of `index` is 1; the upper bound is the current number of elements. The upper bound changes as you add or delete elements, but it cannot exceed the maximum size. When you store and retrieve a varray from the database, its indexes and element order remain stable.

**Figure 6-1** shows a varray variable named Grades, which has maximum size 10 and contains seven elements. Grades(`n`) references the `n`th element of Grades. The upper bound of Grades is 7, and it cannot exceed 10.

**Figure 6-1  Varray of Maximum Size 10 with 7 Elements**

<table>
<thead>
<tr>
<th>Grades</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
<td>(6)</td>
<td>(7)</td>
</tr>
</tbody>
</table>

The database stores a varray variable as a single object. If a varray variable is less than 4 KB, it resides inside the table of which it is a column; otherwise, it resides outside the table but in the same tablespace.

An uninitialized varray variable is a null collection. You must initialize it, either by making it empty or by assigning a non-**NULL** value to it. For details, see "Collection Constructors" and "Assigning Values to Collection Variables".
Topics

• Appropriate Uses for Varrays

See Also:

• Table 6-1 for a summary of varray characteristics
• "varray_type_def ::=" for the syntax of a VARRAY type definition
• "CREATE TYPE Statement" for information about creating standalone VARRAY types
• Oracle Database SQL Language Reference for more information about varrays

Example 6-4 Varray (Variable-Size Array)

This example defines a local VARRAY type, declares a variable of that type (initializing it with a constructor), and defines a procedure that prints the varray. The example invokes the procedure three times: After initializing the variable, after changing the values of two elements individually, and after using a constructor to change the values of all elements. (For an example of a procedure that prints a varray that might be null or empty, see Example 6-29.)

Live SQL:

You can view and run this example on Oracle Live SQL at Varray (Variable-Size Array)

DECLARE
    TYPE Foursome IS VARRAY(4) OF VARCHAR2(15);  -- VARRAY type
    team Foursome := Foursome('John', 'Mary', 'Alberto', 'Juanita');

PROCEDURE print_team (heading VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE(heading);
        FOR i IN 1..4 LOOP
            DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('---');
    END;

BEGIN
    print_team('2001 Team:');

    team(3) := 'Pierre';  -- Change values of two elements
    team(4) := 'Yvonne';

    print_team('2003 Team:');

    team := Foursome('John', 'Mary', 'Pierre', 'Yvonne');
    print_team('2004 Team:');

END;
print_team('2005 Team:');

-- Invoke constructor to assign new values to varray variable:

team := Foursome('Arun', 'Amitha', 'Allan', 'Mae');
print_team('2009 Team:');
END;
/

Result:

2001 Team:
1. John
2. Mary
3. Alberto
4. Juanita
---

2005 Team:
1. John
2. Mary
3. Pierre
4. Yvonne
---

2009 Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---

6.3.1 Appropriate Uses for Varrays

A varray is appropriate when:

- You know the maximum number of elements.
- You usually access the elements sequentially.

Because you must store or retrieve all elements at the same time, a varray might be impractical for large numbers of elements.

6.4 Nested Tables

In the database, a nested table is a column type that stores an unspecified number of rows in no particular order.

When you retrieve a nested table value from the database into a PL/SQL nested table variable, PL/SQL gives the rows consecutive indexes, starting at 1. Using these indexes, you can access the individual rows of the nested table variable. The syntax is `variable_name(index)`. The indexes and row order of a nested table might not remain stable as you store and retrieve the nested table from the database.

The amount of memory that a nested table variable occupies can increase or decrease dynamically, as you add or delete elements.

An uninitialized nested table variable is a null collection. You must initialize it, either by making it empty or by assigning a non-NULL value to it. For details, see "Collection Constructors" and "Assigning Values to Collection Variables".
Note:

Example 6-22, Example 6-24, and Example 6-25 reuse nt_type and print_nt.

Topics

- Important Differences Between Nested Tables and Arrays
- Appropriate Uses for Nested Tables

See Also:

- Table 6-1 for a summary of nested table characteristics
- "nested_table_type_def ::=" for the syntax of a nested table type definition
- "CREATE TYPE Statement" for information about creating standalone nested table types
- "INSTEAD OF DML Triggers" for information about triggers that update nested table columns of views
- Oracle Database SQL Language Reference for more information about nested tables

Example 6-5  Nested Table of Local Type

This example defines a local nested table type, declares a variable of that type (initializing it with a constructor), and defines a procedure that prints the nested table. (The procedure uses the collection methods FIRST and LAST, described in "Collection Methods"). The example invokes the procedure three times: After initializing the variable, after changing the value of one element, and after using a constructor to the change the values of all elements. After the second constructor invocation, the nested table has only two elements. Referencing element 3 would raise error ORA-06533.

Live SQL:

You can view and run this example on Oracle Live SQL at Nested Table of Local Type

DECLARE
  TYPE Roster IS TABLE OF VARCHAR2(15);  -- nested table type

  -- nested table variable initialized with constructor:
  names Roster := Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');

PROCEDURE print_names (heading VARCHAR2) IS
  BEGIN
DBMS_OUTPUT.PUT_LINE(heading);

FOR i IN names.FIRST .. names.LAST LOOP  -- For first to last element
   DBMS_OUTPUT.PUT_LINE(names(i));
END LOOP;

DBMS_OUTPUT.PUT_LINE('---');
END;

BEGIN
   print_names('Initial Values:');

   names(3) := 'P Perez';  -- Change value of one element
   print_names('Current Values:');

   names := Roster('A Jansen', 'B Gupta');  -- Change entire table
   print_names('Current Values:');
END;
/

Result:

Initial Values:
D Caruso
J Hamil
D Piro
R Singh
---
Current Values:
D Caruso
J Hamil
P Perez
R Singh
---
Current Values:
A Jansen
B Gupta

Example 6-6  Nested Table of Standalone Type

This example defines a standalone nested table type, nt_type, and a standalone procedure to print a variable of that type, print_nt. An anonymous block declares a variable of type nt_type, initializing it to empty with a constructor, and invokes print_nt twice: After initializing the variable and after using a constructor to the change the values of all elements.

Live SQL:
You can view and run this example on Oracle Live SQL at Nested Table of Standalone Type

CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) AUTHID DEFINER IS
   i  NUMBER;
BEGIN
   i := nt.FIRST;
IF i IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('nt is empty');
ELSE
    WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE('nt.(' || i || ') = ');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt(i)), 'NULL'));
        i := nt.NEXT(i);
    END LOOP;
END IF;
DBMS_OUTPUT.PUT_LINE('---');
END print_nt;
/

DECLARE
    nt nt_type := nt_type();  -- nested table variable initialized to empty
BEGIN
    print_nt(nt);
    nt := nt_type(90, 9, 29, 58);
    print_nt(nt);
END;
/

Result:
nt is empty
---
nt.(1) = 90
nt.(2) = 9
nt.(3) = 29
nt.(4) = 58
---

6.4.1 Important Differences Between Nested Tables and Arrays

Conceptually, a nested table is like a one-dimensional array with an arbitrary number
of elements. However, a nested table differs from an array in these important ways:

• An array has a declared number of elements, but a nested table does not. The
  size of a nested table can increase dynamically.
• An array is always dense. A nested array is dense initially, but it can become
  sparse, because you can delete elements from it.

Figure 6-2 shows the important differences between a nested table and an array.

Figure 6-2  Array and Nested Table
6.4.2 Appropriate Uses for Nested Tables

A nested table is appropriate when:

- The number of elements is not set.
- Index values are not consecutive.
- You must delete or update some elements, but not all elements simultaneously.

Nested table data is stored in a separate store table, a system-generated database table. When you access a nested table, the database joins the nested table with its store table. This makes nested tables suitable for queries and updates that affect only some elements of the collection.

- You would create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

6.5 Collection Constructors

A **collection constructor** (constructor) is a system-defined function with the same name as a collection type, which returns a collection of that type.

---

**Note:**

This topic applies only to varrays and nested tables. In this topic, **collection** means varray or nested table. Associative arrays use qualified expressions and aggregates (see Qualified Expressions Overview).

---

The syntax of a constructor invocation is:

```
collection_type ( [ value [, value ]... ] )
```

If the parameter list is empty, the constructor returns an empty collection. Otherwise, the constructor returns a collection that contains the specified values. For semantic details, see "collection_constructor".

You can assign the returned collection to a collection variable (of the same type) in the variable declaration and in the executable part of a block.

**Example 6-7  Initializing Collection (Varray) Variable to Empty**

This example invokes a constructor twice: to initialize the varray variable `team` to empty in its declaration, and to give it new values in the executable part of the block. The procedure `print_team` shows the initial and final values of `team`. To determine when `team` is empty, `print_team` uses the collection method `COUNT`, described in "Collection Methods". (For an example of a procedure that prints a varray that might be null, see Example 6-29.)
DECLARE
  TYPE Foursome IS VARRAY(4) OF VARCHAR2(15);
  team Foursome := Foursome();  -- initialize to empty
PROCEDURE print_team (heading VARCHAR2)
  IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE(heading);
      IF team.COUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Empty');
      ELSE
        FOR i IN 1..4 LOOP
          DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
        END LOOP;
      END IF;
      DBMS_OUTPUT.PUT_LINE('---');
    END;
BEGIN
  print_team('Team:');
  team := Foursome('John', 'Mary', 'Alberto', 'Juanita');
  print_team('Team:');
END;
/

Result:

Team:
Empty
---
Team:
1.John
2.Mary
3.Alberto
4.Juanita
---

6.6 Qualified Expressions Overview

Qualified expressions improve program clarity and developer productivity by providing the ability to declare and define a complex value in a compact form where the value is needed.

A qualified expression combines expression elements to create values of a RECORD type or associative array type.

Qualified expressions use an explicit type indication to provide the type of the qualified item. This explicit indication is known as a typemark.

Qualified expressions have this structure:
qualified_expression ::= typemark ( aggregate )

aggregate ::= [ positional_choice_list ] [ explicit_choice_list ]

positional_choice_list ::= ( expr )+ | sequence_iterator_choice

explicit_choice_list ::= named_choice_list
                     | indexed_choice_list
                     | basic_iterator_choice
                     | index_iterator_choice

named_choice_list ::= identifier => expr [,]+

indexed_choice_list ::= expr => expr [,] +

sequence_iterator_choice ::= FOR iterator SEQUENCE => expr

basic_iterator_choice ::= FOR iterator => expr

index_iterator_choice ::= FOR iterator INDEX expr => expr

See "qualified_expression ::=" for more information about the syntax and semantics.

Expanding Basic Iterator Choice Association Into PL/SQL

The basic iterator choice association uses the iterand as an index.

For each iterand value, the expression is evaluated and added to the collection using the iterand value as the index.

```
typemark (FOR iterand IN iteration_controls => expr)
  - Create an empty collection of type typemark
FOR iterand IN iteration_controls LOOP
  DECLARE
    expr_temp typemark%valuetype := expr;
  BEGIN
    - Extend collection_temp to iterand if appropriate for typemark
    collection_temp(iterand) := expr_temp;
  END;
END LOOP;
```

Expansion of basic iterator choice association can be described informally as follows.

For each value of iterand generated by the iteration controls:

1. Evaluate the expression producing an expression value.
2. If appropriate for the collection type, extend the collection to the index specified by the iterand.
3. Add the expression value to the collection at the index specified by the iterand value.

Example 6-8 Basic Iterator Choice Association in Qualified Expressions

This example creates a vector of the first N fibonacci numbers.

```
result := vec_t (FOR i IN 1..n => fib(i));
```
This example creates a vector of the first N even numbers.

```plsql
result := vec_t (FOR i IN 1..N => 2*i);
```

**Expanding Index Iterator Choice Association Into PL/SQL**

The index iterator choice association provides an index expression along with the value expression.

For each iterand value, the index expression and value expression are evaluated. Then the expanded value is added to the collection using the expanded index.

```plsql
typemark (FOR iterand IN iteration_controls
    INDEX expr1 => expr2)
- Create an empty collection of type typemark
FOR iterand IN iteration_controls LOOP
    DECLARE
        index_temp typemark%indextype := expr1;
        expr_temp typemark%valuetype := expr2;
    BEGIN
        - Extend collection_temp to index_temp if appropriate for typemark
        collection_temp(index_temp) := expr_temp;
    END;
END LOOP;
```

Expansion of index iterator choice association can be described informally as follows.

For each value of iterand generated by the iteration controls:

1. Evaluate the expression producing an expression value.
2. Evaluate the index expression producing an index value.
3. If appropriate for the collection type, extend the collection to the index specified by the index value.
4. Add the expression value to the collection at the index specified by the index value.

**Example 6-9  Index Iterator Choice Association in Qualified Expressions**

This example creates a copy of vec with values incremented by N.

```plsql
result := vec_t (FOR I,j IN PAIRS OF vec INDEX I => j+n);
```

This example creates a vector of the first N even numbers.

```plsql
result := vec_t (FOR i IN 2..N BY 2 INDEX i/2 => i);
```

**Expanding Sequence Iterator Choice Association Into PL/SQL**

The sequence iterator choice association allows a sequence of values to be added to the end of a collection. In each case, the expressions specified may reference the iterands.

For each iterand value, the value expression is evaluated and added to the end of the collection.

```plsql
typemark (FOR iterand IN iteration_controls
    SEQUENCE => expr)
```
- Create an empty collection of type typemark
  DECLARE
    col_size PLS_INTEGER := current_end_of_collection;
  FOR iterand IN iteration_controls LOOP
    col_size := col_size + 1;
    DECLARE
      expr_temp typemark%valuetype := expr;
    BEGIN
      -- Extend collection_temp by one if appropriate for typemark
      collection_temp(col_size) := expr_temp;
    END;
  END LOOP;

Expansion of sequence iterator choice association can be described informally as follows.

For each value of iterand generated by the iteration controls:

1. Evaluate the expression producing an expression value.
2. If appropriate for the collection type, extend the collection by one.
3. Add the expression value to the collection at its end.

Example 6-10  Sequence Iterator Choice Association in Qualified Expressions

This example concatenates vectors v1 and reversed v2 together.

result := vec_t (FOR v IN VALUES OF v1,
                   REVERSE VALUES OF v2
                   SEQUENCE => v);

This example creates a vector of the prime numbers less than or equal to N.

result := vec_t (FOR i IN 1..n WHEN is_prime(i)
                   SEQUENCE => i);

Example 6-11  Assigning Values to RECORD Type Variables Using Qualified Expressions

This example shows the declaration, initialization, and definition of RECORD type variables. Type rec_t is defined and partially initialized in package pkg. Variable v_rec1 is declared with that type and assigned initial values using the positional notation. Variable v_rec2 is declared with that type as well and assigned initial values using the named association syntax. Variable v_rec3 is assigned the NULL values. The procedure print_rec displays the values of the local variable v_rec1, followed by the procedure parameter pi_rec variable values. If no parameter is passed to the procedure, it displays the initial values set in the procedure definition.

Live SQL:

You can view and run this example on Oracle Live SQL at "18c Assigning Values to RECORD Type Variables Using Qualified Expressions"

CREATE PACKAGE pkg IS
  TYPE rec_t IS RECORD
(year PLS_INTEGER := 2,
  name VARCHAR2 (100) );
END;

DECLARE
  v_rec1 pkg.rec_t := pkg.rec_t(1847,'ONE EIGHT FOUR SEVEN');
  v_rec2 pkg.rec_t := pkg.rec_t(year => 1, name => 'ONE');
  v_rec3 pkg.rec_t := pkg.rec_t(NULL,NULL);

PROCEDURE print_rec ( pi_rec pkg.rec_t := pkg.rec_t(1847+1, 'a'||'b')) IS
  v_rec1 pkg.rec_t := pkg.rec_t(2847,'TWO EIGHT FOUR SEVEN');
BEGIN
  DBMS_OUTPUT.PUT_LINE(NVL(v_rec1.year,0) ||' ' ||NVL(v_rec1.name,'N/A'));
  DBMS_OUTPUT.PUT_LINE(NVL(pi_rec.year,0) ||' ' ||NVL(pi_rec.name,'N/A'));
END;
BEGIN
  print_rec(v_rec1);
  print_rec(v_rec2);
  print_rec(v_rec3);
  print_rec();
END;

2847 TWO EIGHT FOUR SEVEN
1847 ONE EIGHT FOUR SEVEN
2847 TWO EIGHT FOUR SEVEN
1 ONE
2847 TWO EIGHT FOUR SEVEN
0 N/A
2847 TWO EIGHT FOUR SEVEN
1848 ab

Example 6-12 Assigning Values to Associative Array Type Variables Using Qualified Expressions

This example uses a function to display the values of a table of BOOLEAN.

CREATE FUNCTION print_bool (v IN BOOLEAN)
  RETURN VARCHAR2
IS
  v_rtn VARCHAR2(10);
BEGIN
  CASE v
  WHEN TRUE THEN
    v_rtn := 'TRUE';
  WHEN FALSE THEN
    v_rtn := 'FALSE';
  END CASE;
  RETURN v_rtn;
END;

BEGIN
  print_bool(TRUE);
  print_bool(FALSE);
END;
ELSE
  v_rtn := 'NULL';
END CASE;
RETURN v_rtn;
END print_bool;

The variable v_aa1 is initialized using index key-value pairs.

DECLARE
  TYPE t_aa IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
  v_aa1 t_aa := t_aa(1=>FALSE,
                      2=>TRUE,
                      3=>NULL);
BEGIN
  DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(1)));
  DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(2)));
  DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(3)));
END;

FALSE
TRUE
NULL

### 6.7 Assigning Values to Collection Variables

You can assign a value to a collection variable in these ways:

- Invoke a constructor to create a collection and assign it to the collection variable.
- Use the assignment statement to assign it the value of another existing collection variable.
- Pass it to a subprogram as an **OUT** or **IN OUT** parameter, and then assign the value inside the subprogram.
- Use a qualified expression to assign values to an associative array (see Example 6-12).

To assign a value to a scalar element of a collection variable, reference the element as `collection_variable_name(index)` and assign it a value.

**Topics**

- Data Type Compatibility
- Assigning Null Values to Varray or Nested Table Variables
- Assigning Set Operation Results to Nested Table Variables
### 6.7.1 Data Type Compatibility

You can assign a collection to a collection variable only if they have the same data type. Having the same element type is not enough.

**Example 6-13  Data Type Compatibility for Collection Assignment**

In this example, VARCHAR types *triplet* and *trio* have the same element type, VARCHAR(15). Collection variables *group1* and *group2* have the same data type, triplet, but collection variable *group3* has the data type trio. The assignment of *group1* to *group2* succeeds, but the assignment of *group1* to *group3* fails.

#### Live SQL:

You can view and run this example on Oracle Live SQL at Data Type Compatibility for Collection Assignment

```sql
DECLARE
    TYPE triplet IS VARRAY(3) OF VARCHAR2(15);
    TYPE trio    IS VARRAY(3) OF VARCHAR2(15);

    group1 triplet := triplet('Jones', 'Wong', 'Marceau');
    group2 triplet;
    group3 trio;
BEGIN
    group2 := group1;  -- succeeds
    group3 := group1;  -- fails
END;
/
```

**Result:**

ORA-06550: line 10, column 13:
PLS-00382: expression is of wrong type
6.7.2 Assigning Null Values to Varray or Nested Table Variables

To a varray or nested table variable, you can assign the value NULL or a null collection of the same data type. Either assignment makes the variable null.

Example 6-14 initializes the nested table variable `dept_names` to a non-null value; assigns a null collection to it, making it null; and re-initializes it to a different non-null value.

Example 6-14 Assigning Null Value to Nested Table Variable

```sql
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30);
  dept_names dnames_tab := dnames_tab('Shipping','Sales','Finance','Payroll');  -- Initialized to non-null value
  empty_set dnames_tab;  -- Not initialized, therefore null

PROCEDURE print_dept_names_status IS
  BEGIN
    IF dept_names IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('dept_names is null.');
    ELSE
      DBMS_OUTPUT.PUT_LINE('dept_names is not null.');</
    END IF;
  END print_dept_names_status;
BEGIN
  print_dept_names_status;
  dept_names := empty_set;  -- Assign null collection to dept_names.
  print_dept_names_status;
  dept_names := dnames_tab ('Shipping','Sales','Finance','Payroll');  -- Re-initialize dept_names
  print_dept_names_status;
END; /

Result:

department_names is not null.
department_names is null.
department_names is not null.
```
6.7.3 Assigning Set Operation Results to Nested Table Variables

To a nested table variable, you can assign the result of a SQL `MULTISET` operation or SQL `SET` function invocation.

The SQL `MULTISET` operators combine two nested tables into a single nested table. The elements of the two nested tables must have comparable data types. For information about the `MULTISET` operators, see *Oracle Database SQL Language Reference*.

The SQL `SET` function takes a nested table argument and returns a nested table of the same data type whose elements are distinct (the function eliminates duplicate elements). For information about the `SET` function, see *Oracle Database SQL Language Reference*.

**Example 6-15  Assigning Set Operation Results to Nested Table Variable**

This example assigns the results of several `MULTISET` operations and one `SET` function invocation of the nested table variable `answer`, using the procedure `print_nested_table` to print `answer` after each assignment. The procedure uses the collection methods `FIRST` and `LAST`, described in "Collection Methods".

```
DECLARE
    TYPE nested_typ IS TABLE OF NUMBER;

    nt1    nested_typ := nested_typ(1,2,3);
    nt2    nested_typ := nested_typ(3,2,1);
    nt3    nested_typ := nested_typ(2,3,1,3);
    nt4    nested_typ := nested_typ(1,2,4);
    answer nested_typ;

    PROCEDURE print_nested_table (nt nested_typ) IS
        output VARCHAR2(128);
        BEGIN
            IF nt IS NULL THEN
                DBMS_OUTPUT.PUT_LINE('Result: null set');
            ELSIF nt.COUNT = 0 THEN
                DBMS_OUTPUT.PUT_LINE('Result: empty set');
            ELSE
                FOR i IN nt.FIRST .. nt.LAST LOOP  -- For first to last element
                    output := output || nt(i) || ' ';
                END LOOP;
                DBMS_OUTPUT.PUT_LINE('Result: ' || output);
            END IF;
        END print_nested_table;

    BEGIN
        answer := nt1 MULTISET UNION nt4;
        print_nested_table(answer);
        answer := nt1 MULTISET UNION nt3;
    END;
```

---

**Live SQL:**

You can view and run this example on Oracle Live SQL at [Assigning Set Operation Results to Nested Table Variable](http://example.com).
print_nested_table(answer);
answer := nt1 MULTISET UNION DISTINCT nt3;
print_nested_table(answer);
answer := nt2 MULTISET INTERSECT nt3;
print_nested_table(answer);
answer := nt2 MULTISET INTERSECT DISTINCT nt3;
print_nested_table(answer);
another := SET(nt3);
print_nested_table(answer);
another := nt3 MULTISET EXCEPT nt2;
print_nested_table(answer);
another := nt3 MULTISET EXCEPT DISTINCT nt2;
print_nested_table(answer);
END;
/

Result:
Result: 1 2 3 1 2 4
Result: 1 2 3 2 3 1 3
Result: 1 2 3
Result: 3 2 1
Result: 3 2 1
Result: 2 3 1
Result: 3
Result: empty set

6.8 Multidimensional Collections

Although a collection has only one dimension, you can model a multidimensional collection with a collection whose elements are collections.

Example 6-16 Two-Dimensional Varray (Varray of Varrays)

In this example, nva is a two-dimensional varray—a varray of varrays of integers.

```
DECLARE
    TYPE t1 IS VARRAY(10) OF INTEGER;  -- varray of integer
    va t1 := t1(2,3,5);
    TYPE nt1 IS VARRAY(10) OF t1;      -- varray of varray of integer
    nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
    i INTEGER;
    va1 t1;
BEGIN
    i := nva(2)(3);
    DBMS_OUTPUT.PUT_LINE('i = ' || i);
    nva.EXTEND;
    nva(5) := t1(56, 32);            -- replace inner varray elements
    nva(4) := t1(45, 43, 67, 43345);  -- replace an inner integer element
```

Live SQL:
You can view and run this example on Oracle Live SQL at Two-Dimensional Varray (Varray of Varrays)
nva(4)(4) := 1;                -- replace 43345 with 1
nva(4).EXTEND;    -- add element to 4th varray element
nva(4)(5) := 89;  -- store integer 89 there
END;
/

Result:
i = 73

Example 6-17   Nested Tables of Nested Tables and Varrays of Integers

In this example, ntb1 is a nested table of nested tables of strings, and ntb2 is a nested table of varrays of integers.

Live SQL:
You can view and run this example on Oracle Live SQL at Nested Tables of Nested Tables and Varrays of Integers

DECLARE
  TYPE tb1 IS TABLE OF VARCHAR2(20);  -- nested table of strings
  vtb1 tb1 := tb1('one', 'three');

  TYPE nt1b IS TABLE OF tb1; -- nested table of nested tables of strings
  vntb1 nt1b := nt1b(vtb1);

  TYPE tv1 IS VARRAY(10) OF INTEGER;  -- varray of integers
  TYPE nt1b2 IS TABLE OF tv1;          -- nested table of varrays of integers
  vntb2 nt1b2 := nt1b2(tv1(3,5), tv1(5,7,3));
BEGIN
  vntb1.EXTEND;
  vntb1(2) := vntb1(1);
  vntb1.DELETE(1);     -- delete first element of vntb1
  vntb1(2).DELETE(1);  -- delete first string from second table in nested table
END;
/

Example 6-18   Nested Tables of Associative Arrays and Varrays of Strings

In this example, aa1 is an associative array of associative arrays, and ntb2 is a nested table of varrays of strings.

Live SQL:
You can view and run this example on Oracle Live SQL at Nested Tables of Associative Arrays and Varrays of Strings

DECLARE
  TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS_INTEGER;  -- associative arrays
  v4 tb1;
  v5 tb1;
TYPE aa1 IS TABLE OF tb1 INDEX BY PLS_INTEGER;  -- associative array of
v2 aa1;                                         -- associative arrays

TYPE val IS VARRAY(10) OF VARCHAR2(20);  -- varray of strings
v1 val := val('hello', 'world');

TYPE ntb2 IS TABLE OF val INDEX BY PLS_INTEGER;  -- associative array of varrays
v3 ntb2;

BEGIN
v4(1)   := 34;     -- populate associative array
v4(2)   := 46456;
v4(456) := 343;

v2(23) := v4;  -- populate associative array of associative arrays

v3(34) := val(33, 456, 656, 343);  -- populate associative array varrays

v2(35) := v5;      -- assign empty associative array to v2(35)
v2(35)(2) := 78;
END;
/

6.9 Collection Comparisons

To determine if one collection variable is less than another (for example), you must
define what less than means in that context and write a function that returns TRUE or FALSE.

You cannot compare associative array variables to the value NULL or to each other.

Except for Comparing Nested Tables for Equality and Inequality, you cannot natively compare two collection variables with relational operators. This restriction also applies to implicit comparisons. For example, a collection variable cannot appear in a DISTINCT, GROUP BY, or ORDER BY clause.

Topics

• Comparing Varray and Nested Table Variables to NULL
• Comparing Nested Tables for Equality and Inequality
• Comparing Nested Tables with SQL Multiset Conditions

See Also:

• Table 3-5
• PL/SQL Subprograms for information about writing functions

ORACLE
6.9.1 Comparing Varray and Nested Table Variables to NULL

Use the IS[NOT] NULL operator when comparing to the NULL value.

You can compare varray and nested table variables to the value NULL with the "IS [NOT] NULL Operator", but not with the relational operators equal (=) and not equal (<>), !=, -=, or ^=).

**Example 6-19  Comparing Varray and Nested Table Variables to NULL**

This example compares a varray variable and a nested table variable to NULL correctly.

```sql
DECLARE
    TYPE Foursome IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type
    team Foursome; -- varray variable

    TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type
    names Roster := Roster('Adams', 'Patel'); -- nested table variable
BEGIN
    IF team IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('team IS NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE('team IS NOT NULL');
    END IF;

    IF names IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('names IS NOT NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE('names IS NULL');
    END IF;
END;
/

Result:
team IS NULL
names IS NOT NULL
```

6.9.2 Comparing Nested Tables for Equality and Inequality

Two nested table variables are equal if and only if they have the same set of elements (in any order).

If two nested table variables have the same nested table type, and that nested table type does not have elements of a record type, then you can compare the two variables for equality or inequality with the relational operators equal (=) and not equal (<>), !=, -=, ^=).
Example 6-20  Comparing Nested Tables for Equality and Inequality

This example compares nested table variables for equality and inequality with relational operators.

```
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30); -- element type is not record type

  dept_names1 dnames_tab :=
  dnames_tab('Shipping','Sales','Finance','Payroll');

  dept_names2 dnames_tab :=
  dnames_tab('Sales','Finance','Shipping','Payroll');

  dept_names3 dnames_tab :=
  dnames_tab('Sales','Finance','Payroll');

BEGIN
  IF dept_names1 = dept_names2 THEN
    DBMS_OUTPUT.PUT_LINE('dept_names1 = dept_names2');
  END IF;

  IF dept_names2 != dept_names3 THEN
    DBMS_OUTPUT.PUT_LINE('dept_names2 != dept_names3');
  END IF;
END;
/
```

Result:

```
depth_names1 = dept_names2
dept_names2 != dept_names3
```
6.9.3 Comparing Nested Tables with SQL Multiset Conditions

You can compare nested table variables, and test some of their properties, with SQL multiset conditions.

See Also:

- Oracle Database SQL Language Reference for more information about multiset conditions
- Oracle Database SQL Language Reference for details about CARDINALITY syntax
- Oracle Database SQL Language Reference for details about SET syntax

Example 6-21 Comparing Nested Tables with SQL Multiset Conditions

This example uses the SQL multiset conditions and two SQL functions that take nested table variable arguments, CARDINALITY and SET.

Live SQL:

You can view and run this example on Oracle Live SQL at Comparing Nested Tables with SQL Multiset Conditions

DECLARE
    TYPE nested_typ IS TABLE OF NUMBER;
    nt1 nested_typ := nested_typ(1,2,3);
    nt2 nested_typ := nested_typ(3,2,1);
    nt3 nested_typ := nested_typ(2,3,1,3);
    nt4 nested_typ := nested_typ(1,2,4);
PROCEDURE testify (        
    truth BOOLEAN := NULL,  
    quantity NUMBER := NULL   
) IS
    BEGIN
        IF truth IS NOT NULL THEN
            DBMS_OUTPUT.PUT_LINE (  
                CASE truth  
                    WHEN TRUE THEN 'True'  
                    WHEN FALSE THEN 'False'  
                END  
            );
        END IF;
        IF quantity IS NOT NULL THEN
            DBMS_OUTPUT.PUT_LINE(quantity);
        END IF;
    END;
BEGIN
    testify(truth => (nt1 IN (nt2,nt3,nt4))); -- condition
    testify(truth => (nt1 SUBMULTISET OF nt3)); -- condition
END;
testify(truth => (nt1 NOT SUBMULTISET OF nt4));  -- condition
testify(truth => (4 MEMBER OF nt1));  -- condition
testify(truth => (nt3 IS A SET));  -- condition
testify(truth => (nt3 IS NOT A SET));  -- condition
testify(truth => (nt1 IS EMPTY));  -- condition
testify(quantity => (CARDINALITY(nt3))));  -- function
testify(quantity => (CARDINALITY(SET(nt3)))));  -- 2 functions
END;
/

Result:
True
True
True
False
False
True
False
4
3

6.10 Collection Methods

A collection method is a PL/SQL subprogram—either a function that returns information about a collection or a procedure that operates on a collection. Collection methods make collections easier to use and your applications easier to maintain.

Table 6-2 summarizes the collection methods.

Note:
With a null collection, EXISTS is the only collection method that does not raise the predefined exception COLLECTION_IS_NULL.

Table 6-2  Collection Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>Procedure</td>
<td>Deletes elements from collection.</td>
</tr>
<tr>
<td>TRIM</td>
<td>Procedure</td>
<td>Deletes elements from end of varray or nested table.</td>
</tr>
<tr>
<td>EXTEND</td>
<td>Procedure</td>
<td>Adds elements to end of varray or nested table.</td>
</tr>
<tr>
<td>EXISTS</td>
<td>Function</td>
<td>Returns TRUE if and only if specified element of varray or nested table exists.</td>
</tr>
<tr>
<td>FIRST</td>
<td>Function</td>
<td>Returns first index in collection.</td>
</tr>
<tr>
<td>LAST</td>
<td>Function</td>
<td>Returns last index in collection.</td>
</tr>
<tr>
<td>COUNT</td>
<td>Function</td>
<td>Returns number of elements in collection.</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Function</td>
<td>Returns maximum number of elements that collection can have.</td>
</tr>
<tr>
<td>PRIOR</td>
<td>Function</td>
<td>Returns index that precedes specified index.</td>
</tr>
<tr>
<td>NEXT</td>
<td>Function</td>
<td>Returns index that succeeds specified index.</td>
</tr>
</tbody>
</table>
The basic syntax of a collection method invocation is:

```
collection_name.method
```

For detailed syntax, see "Collection Method Invocation".

A collection method invocation can appear anywhere that an invocation of a PL/SQL subprogram of its type (function or procedure) can appear, except in a SQL statement. (For general information about PL/SQL subprograms, see PL/SQL Subprograms.)

In a subprogram, a collection parameter assumes the properties of the argument bound to it. You can apply collection methods to such parameters. For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

**Topics**

- DELETE Collection Method
- TRIM Collection Method
- EXTEND Collection Method
- EXISTS Collection Method
- FIRST and LAST Collection Methods
- COUNT Collection Method
- LIMIT Collection Method
- PRIOR and NEXT Collection Methods

### 6.10.1 DELETE Collection Method

**DELETE** is a procedure that deletes elements from a collection.

This method has these forms:

- **DELETE** deletes all elements from a collection of any type.
  
  This operation immediately frees the memory allocated to the deleted elements.

- From an associative array or nested table (but not a varray):
  
  - **DELETE(n)** deletes the element whose index is \( n \), if that element exists; otherwise, it does nothing.
  
  - **DELETE(m, n)** deletes all elements whose indexes are in the range \( m .. n \), if both \( m \) and \( n \) exist and \( m \leq n \); otherwise, it does nothing.

For these two forms of **DELETE**, PL/SQL keeps placeholders for the deleted elements. Therefore, the deleted elements are included in the internal size of the collection, and you can restore a deleted element by assigning a valid value to it.

**Example 6-22  DELETE Method with Nested Table**

This example declares a nested table variable, initializing it with six elements; deletes and then restores the second element; deletes a range of elements and then restores one of them; and then deletes all elements. The restored elements occupy the same memory as the corresponding deleted elements. The procedure `print_nt` prints the nested table variable after initialization and after each **DELETE** operation. The type `nt_type` and procedure `print_nt` are defined in Example 6-6.
DECLARE
  nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
  print_nt(nt);
  nt.DELETE(2);     -- Delete second element
  print_nt(nt);
  nt(2) := 2222;    -- Restore second element
  print_nt(nt);
  nt.DELETE(2, 4);  -- Delete range of elements
  print_nt(nt);
  nt(3) := 3333;    -- Restore third element
  print_nt(nt);
  nt.DELETE;        -- Delete all elements
  print_nt(nt);
END;
/

Result:
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(2) = 2222
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 3333
nt.(5) = 55
nt.(6) = 66
---
nt is empty
---

Example 6-23  DELETE Method with Associative Array Indexed by String

This example populates an associative array indexed by string and deletes all elements, which frees the memory allocated to them. Next, the example replaces the deleted elements—that is, adds new elements that have the same indexes as the
deleted elements. The new replacement elements do not occupy the same memory as
the corresponding deleted elements. Finally, the example deletes one element and
then a range of elements. The procedure print_aa_str shows the effects of the
operations.

DECLARE
    TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
    aa_str  aa_type_str;

PROCEDURE print_aa_str IS
    i  VARCHAR2(10);
BEGIN
    i := aa_str.FIRST;
    IF i IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('aa_str is empty');
    ELSE
        WHILE i IS NOT NULL LOOP
            DBMS_OUTPUT.PUT('aa_str.(' || i || ') = ');
            DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa_str(i)), 'NULL'));
            i := aa_str.NEXT(i);
        END LOOP;
    END IF;
    DBMS_OUTPUT.PUT_LINE('---');
END print_aa_str;
BEGIN
    aa_str('M') := 13;
    aa_str('Z') := 26;
    aa_str('C') := 3;
    print_aa_str;

    aa_str.DELETE;  -- Delete all elements
    print_aa_str;

    aa_str('M') := 13;  -- Replace deleted element with same value
    aa_str('Z') := 260; -- Replace deleted element with new value
    aa_str('C') := 30;  -- Replace deleted element with new value
    aa_str('W') := 23;  -- Add new element
    aa_str('J') := 10;  -- Add new element
    aa_str('N') := 14;  -- Add new element
    aa_str('P') := 16;  -- Add new element
    aa_str('W') := 23;  -- Add new element
    aa_str('J') := 10;  -- Add new element
    print_aa_str;

    aa_str.DELETE('C');  -- Delete one element
    print_aa_str;

    aa_str.DELETE('N','W'); -- Delete range of elements
    print_aa_str;

    aa_str.DELETE('Z','M'); -- Does nothing
    print_aa_str;
END;
6.10.2 TRIM Collection Method

TRIM is a procedure that deletes elements from the end of a varray or nested table.

This method has these forms:

- **TRIM** removes one element from the end of the collection, if the collection has at least one element; otherwise, it raises the predefined exception SUBSCRIPT_BEYOND_COUNT.

- **TRIM(n)** removes $n$ elements from the end of the collection, if there are at least $n$ elements at the end; otherwise, it raises the predefined exception SUBSCRIPT_BEYOND_COUNT.

TRIM operates on the internal size of a collection. That is, if DELETE deletes an element but keeps a placeholder for it, then TRIM considers the element to exist. Therefore, TRIM can delete a deleted element.

PL/SQL does not keep placeholders for trimmed elements. Therefore, trimmed elements are not included in the internal size of the collection, and you cannot restore a trimmed element by assigning a valid value to it.

⚠️ Caution:

Do not depend on interaction between TRIM and DELETE. Treat nested tables like either fixed-size arrays (and use only DELETE) or stacks (and use only TRIM and EXTEND).
Example 6-24  TRIM Method with Nested Table

This example declares a nested table variable, initializing it with six elements; trims the last element; deletes the fourth element; and then trims the last two elements—one of which is the deleted fourth element. The procedure `print_nt` prints the nested table variable after initialization and after the TRIM and DELETE operations. The type `nt_type` and procedure `print_nt` are defined in Example 6-6.

```sql
DECLARE
    nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
    print_nt(nt);
    nt.TRIM;  -- Trim last element
    print_nt(nt);
    nt.DELETE(4);  -- Delete fourth element
    print_nt(nt);
    nt.TRIM(2);  -- Trim last two elements
    print_nt(nt);
END;
/
```

Result:

<table>
<thead>
<tr>
<th>i</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>66</td>
</tr>
<tr>
<td>---</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
</tr>
<tr>
<td>---</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
</tr>
<tr>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>

6.10.3 EXTEND Collection Method

`EXTEND` is a procedure that adds elements to the end of a varray or nested table.

The collection can be empty, but not null. (To make a collection empty or add elements to a null collection, use a constructor. For more information, see "Collection Constructors").

The `EXTEND` method has these forms:

- `EXTEND` appends one null element to the collection.
• **EXTEND(n)** appends \( n \) null elements to the collection.

• **EXTEND(n,i)** appends \( n \) copies of the \( i \)th element to the collection.

---

**Note:**

**EXTEND(n,i)** is the only form that you can use for a collection whose elements have the **NOT NULL** constraint.

**EXTEND** operates on the internal size of a collection. That is, if **DELETE** deletes an element but keeps a placeholder for it, then **EXTEND** considers the element to exist.

**Example 6-25  EXTEND Method with Nested Table**

This example declares a nested table variable, initializing it with three elements; appends two copies of the first element; deletes the fifth (last) element; and then appends one null element. Because **EXTEND** considers the deleted fifth element to exist, the appended null element is the sixth element. The procedure **print_nt** prints the nested table variable after initialization and after the **EXTEND** and **DELETE** operations. The type **nt_type** and procedure **print_nt** are defined in Example 6-6.

```sql
DECLARE
    nt nt_type := nt_type(11, 22, 33);
BEGIN
    print_nt(nt);
    nt.EXTEND(2,1);  -- Append two copies of first element
    print_nt(nt);
    nt.DELETE(5);    -- Delete fifth element
    print_nt(nt);
    nt.EXTEND;       -- Append one null element
    print_nt(nt);
END;
/
```

**Result:**

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
nt.(5) = 11
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
```
### 6.10.4 EXISTS Collection Method

**EXISTS** is a function that tells you whether the specified element of a varray or nested table exists.

**EXISTS**(\(n\)) returns **TRUE** if the \(n\)th element of the collection exists and **FALSE** otherwise. If \(n\) is out of range, **EXISTS** returns **FALSE** instead of raising the predefined exception **SUBSCRIPT_OUTSIDE_LIMIT**.

For a deleted element, **EXISTS**\((n)\) returns **FALSE**, even if **DELETE** kept a placeholder for it.

**Example 6-26**  EXISTS Method with Nested Table

This example initializes a nested table with four elements, deletes the second element, and prints either the value or status of elements 1 through 6.

```sql
DECLARE
    TYPE NumList IS TABLE OF INTEGER;
    n NumList := NumList(1,3,5,7);
BEGIN
    n.DELETE(2); -- Delete second element
    FOR i IN 1..6 LOOP
        IF n.EXISTS(i) THEN
            DBMS_OUTPUT.PUT_LINE('n(' || i || ') = ' || n(i));
        ELSE
            DBMS_OUTPUT.PUT_LINE('n(' || i || ') does not exist');
        END IF;
    END LOOP;
END;
/
```

**Result:**

- \(n(1) = 1\)
- \(n(2)\) does not exist
- \(n(3) = 5\)
- \(n(4) = 7\)
- \(n(5)\) does not exist
- \(n(6)\) does not exist

### 6.10.5 FIRST and LAST Collection Methods

**FIRST** and **LAST** are functions.

If the collection has at least one element, **FIRST** and **LAST** return the indexes of the first and last elements, respectively (ignoring deleted elements, even if **DELETE** kept placeholders for them). If the collection has only one element, **FIRST** and **LAST** return the same index. If the collection is empty, **FIRST** and **LAST** return **NULL**.

**Topics**
- **FIRST and LAST Methods for Associative Array**
- **FIRST and LAST Methods for Varray**
6.10.5.1 FIRST and LAST Methods for Associative Array

For an associative array indexed by PLS_INTEGER, the first and last elements are those with the smallest and largest indexes, respectively. For an associative array indexed by string, the first and last elements are those with the lowest and highest key values, respectively.

Key values are in sorted order (for more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String").

Example 6-27  FIRST and LAST Values for Associative Array Indexed by PLS_INTEGER

This example shows the values of FIRST and LAST for an associative array indexed by PLS_INTEGER, deletes the first and last elements, and shows the values of FIRST and LAST again.

```plsql
DECLARE
    TYPE aa_type_int IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
    aa_int  aa_type_int;
PROCEDURE print_first_and_last IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_int.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_int.LAST);
END print_first_and_last;
BEGIN
    aa_int(1) := 3;
    aa_int(2) := 6;
    aa_int(3) := 9;
    aa_int(4) := 12;
    DBMS_OUTPUT.PUT_LINE('Before deletions:');
    print_first_and_last;
    aa_int.DELETE(1);
    aa_int.DELETE(4);
    DBMS_OUTPUT.PUT_LINE('After deletions:');
    print_first_and_last;
END;
/
```

Result:

Before deletions:
FIRST = 1
LAST = 4
After deletions:
FIRST = 2
LAST = 3

Example 6-28  FIRST and LAST Values for Associative Array Indexed by String

This example shows the values of FIRST and LAST for an associative array indexed by string, deletes the first and last elements, and shows the values of FIRST and LAST again.
DECLARE
    TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
    aa_str  aa_type_str;

PROCEDURE print_first_and_last IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_str.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_str.LAST);
END print_first_and_last;

BEGIN
    aa_str('Z') := 26;
    aa_str('A') := 1;
    aa_str('K') := 11;
    aa_str('R') := 18;

    DBMS_OUTPUT.PUT_LINE('Before deletions:');
    print_first_and_last;

    aa_str.DELETE('A');
    aa_str.DELETE('Z');

    DBMS_OUTPUT.PUT_LINE('After deletions:');
    print_first_and_last;
END;
/

Result:
Before deletions:
FIRST = A
LAST = Z
After deletions:
FIRST = K
LAST = R

6.10.5.2 FIRST and LAST Methods for Varray

For a varray that is not empty, FIRST always returns 1. For every varray, LAST always equals COUNT.

Example 6-29 Printing Varray with FIRST and LAST in FOR LOOP

This example prints the varray team using a FOR LOOP statement with the bounds team.FIRST and team.LAST. Because a varray is always dense, team(i) inside the loop always exists.

DECLARE
    TYPE team_type IS VARRAY(4) OF VARCHAR2(15);
    team team_type;

PROCEDURE print_team (heading VARCHAR2)
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);
    IF team IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('Does not exist');
    ELSIF team.FIRST IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('Has no members');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Has members');
    END IF;
    FOR i IN 1..team.FIRST LOOP
        DBMS_OUTPUT.PUT_LINE(TEAM(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Has ' || team.LAST || ' members');
END print_team;

BEGIN
    team(1) := 'Jon';
    team(2) := 'Sue';
    team(3) := 'Fred';
    team(4) := 'Mary';

    DBMS_OUTPUT.PUT_LINE('Before deletions:');
    print_team('Before deletions:');

    team.DELETE(1);
    team.DELETE(4);

    DBMS_OUTPUT.PUT_LINE('After deletions:');
    print_team('After deletions:');
END;
FOR i IN team.FIRST..team.LAST LOOP
   DBMS_OUTPUT.PUT_LINE(i || ' ' || team(i));
END LOOP;
END IF;

DBMS_OUTPUT.PUT_LINE('---');
END;

BEGIN
print_team('Team Status:');

   team := team_type();  -- Team is funded, but nobody is on it.
   print_team('Team Status:');

   team := team_type('John', 'Mary');  -- Put 2 members on team.
   print_team('Initial Team:');

   team := team_type('Arun', 'Amitha', 'Allan', 'Mae');  -- Change team.
   print_team('New Team:');
END;
/

Result:

Team Status:
Does not exist
---
Team Status:
Has no members
---
Initial Team:
1. John
2. Mary
---
New Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---

Related Topic

• Example 6-31

6.10.5.3 FIRST and LAST Methods for Nested Table

For a nested table, LAST equals COUNT unless you delete elements from its middle, in which case LAST is larger than COUNT.

Example 6-30  Printing Nested Table with FIRST and LAST in FOR LOOP

This example prints the nested table team using a FOR LOOP statement with the bounds team.FIRST and team.LAST. Because a nested table can be sparse, the FOR LOOP statement prints team(i) only if team.EXISTS(i) is TRUE.

DECLARE
   TYPE team_type IS TABLE OF VARCHAR2(15);
   team team_type;
   PROCEDURE print_team (heading VARCHAR2) IS
BEGIN
DBMS_OUTPUT.PUT_LINE(heading);

IF team IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Does not exist');
ELSIF team.FIRST IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Has no members');
ELSE
    FOR i IN team.FIRST..team.LAST LOOP
        DBMS_OUTPUT.PUT(i || '. ');
        IF team.EXISTS(i) THEN
            DBMS_OUTPUT.PUT_LINE(team(i));
        ELSE
            DBMS_OUTPUT.PUT_LINE('(to be hired)');
        END IF;
    END LOOP;
END IF;
DBMS_OUTPUT.PUT_LINE('---');
END;

BEGIN
print_team('Team Status:');

    team := team_type();  -- Team is funded, but nobody is on it.
print_team('Team Status:');

    team := team_type('Arun', 'Amitha', 'Allan', 'Mae');  -- Add members.
print_team('Initial Team:');

    team.DELETE(2,3);  -- Remove 2nd and 3rd members.
print_team('Current Team:');
END;
/

Result:

Team Status:
Does not exist
---
Team Status:
Has no members
---
Initial Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---
Current Team:
1. Arun
2. (to be hired)
3. (to be hired)
4. Mae
---

Related Topic

• Example 6-32
6.10.6 COUNT Collection Method

COUNT is a function that returns the number of elements in the collection (ignoring deleted elements, even if DELETE kept placeholders for them).

Topics
- COUNT Method for Varray
- COUNT Method for Nested Table

6.10.6.1 COUNT Method for Varray

For a varray, COUNT always equals LAST. If you increase or decrease the size of a varray (with the EXTEND or TRIM method), the value of COUNT changes.

Example 6-31  COUNT and LAST Values for Varray

This example shows the values of COUNT and LAST for a varray after initialization with four elements, after EXTEND(3), and after TRIM(5).

```sql
DECLARE
    TYPE NumList IS VARRAY(10) OF INTEGER;
    n NumList := NumList(1,3,5,7);

PROCEDURE print_count_and_last IS
    BEGIN
        DBMS_OUTPUT.PUT('n.COUNT = ' || n.COUNT || ', ');
        DBMS_OUTPUT.PUT_LINE('n.LAST = ' || n.LAST);
    END  print_count_and_last;
BEGIN
    print_count_and_last;
    n.EXTEND(3);
    print_count_and_last;
    n.TRIM(5);
    print_count_and_last;
END;
/
```

Result:

```
n.COUNT = 4, n.LAST = 4
n.COUNT = 7, n.LAST = 7
n.COUNT = 2, n.LAST = 2
```

6.10.6.2 COUNT Method for Nested Table

For a nested table, COUNT equals LAST unless you delete elements from the middle of the nested table, in which case COUNT is smaller than LAST.

Example 6-32  COUNT and LAST Values for Nested Table

This example shows the values of COUNT and LAST for a nested table after initialization with four elements, after deleting the third element, and after adding two null elements to the end. Finally, the example prints the status of elements 1 through 8.
DECLARE
    TYPE NumList IS TABLE OF INTEGER;
    n NumList := NumList(1,3,5,7);

PROCEDURE print_count_and_last IS
    BEGIN
        DBMS_OUTPUT.PUT('n.COUNT = ' || n.COUNT || ', ');
        DBMS_OUTPUT.PUT_LINE('n.LAST = ' || n.LAST);
    END print_count_and_last;
BEGIN
    print_count_and_last;
    n.DELETE(3); -- Delete third element
    print_count_and_last;
    n.EXTEND(2); -- Add two null elements to end
    print_count_and_last;
    FOR i IN 1..8 LOOP
        IF n.EXISTS(i) THEN
            IF n(i) IS NOT NULL THEN
                DBMS_OUTPUT.PUT_LINE('n(' || i || ') = ' || n(i));
            ELSE
                DBMS_OUTPUT.PUT_LINE('n(' || i || ') = NULL');
            END IF;
        ELSE
            DBMS_OUTPUT.PUT_LINE('n(' || i || ') does not exist');
        END IF;
    END LOOP;
END;
/

Result:

n.COUNT = 4, n.LAST = 4
n.COUNT = 3, n.LAST = 4
n.COUNT = 5, n.LAST = 6
n(1) = 1
n(2) = 3
n(3) does not exist
n(4) = 7
n(5) = NULL
n(6) = NULL
n(7) does not exist
n(8) does not exist

6.10.7 LIMIT Collection Method

LIMIT is a function that returns the maximum number of elements that the collection can have. If the collection has no maximum number of elements, LIMIT returns NULL. Only a varray has a maximum size.

Example 6-33 LIMIT and COUNT Values for Different Collection Types

This example prints the values of LIMIT and COUNT for an associative array with four elements, a varray with two elements, and a nested table with three elements.

DECLARE
    TYPE aa_type IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
    aa aa_type; -- associative array
TYPE va_type IS VARRAY(4) OF INTEGER;
  va  := va_type(2,4);   -- varray

TYPE nt_type IS TABLE OF INTEGER;
  nt := nt_type(1,3,5);  -- nested table

BEGIN
  aa(1):=3; aa(2):=6; aa(3):=9; aa(4):= 12;
  DBMS_OUTPUT.PUT('aa.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.COUNT), 'NULL'));
  DBMS_OUTPUT.PUT('aa.LIMIT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.LIMIT), 'NULL'));
  DBMS_OUTPUT.PUT('va.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(va.COUNT), 'NULL'));
  DBMS_OUTPUT.PUT('va.LIMIT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(va.LIMIT), 'NULL'));
  DBMS_OUTPUT.PUT('nt.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.COUNT), 'NULL'));
  DBMS_OUTPUT.PUT('nt.LIMIT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.LIMIT), 'NULL'));
END;
/

Result:

aa.COUNT = 4
aa.LIMIT = NULL
va.COUNT = 2
va.LIMIT = 4
nt.COUNT = 3
nt.LIMIT = NULL

6.10.8 PRIOR and NEXT Collection Methods

PRIOR and NEXT are functions that let you move backward and forward in the collection (ignoring deleted elements, even if DELETE kept placeholders for them). These methods are useful for traversing sparse collections.

Given an index:

- PRIOR returns the index of the preceding existing element of the collection, if one exists. Otherwise, PRIOR returns NULL.
  
  For any collection c, c.PRIOR(c.FIRST) returns NULL.

- NEXT returns the index of the succeeding existing element of the collection, if one exists. Otherwise, NEXT returns NULL.
  
  For any collection c, c.NEXT(c.LAST) returns NULL.

The given index need not exist. However, if the collection c is a varray, and the index exceeds c.LIMIT, then:

- c.PRIOR(index) returns c.LAST.
c.NEXT(index) returns NULL.

For example:

DECLARE
    TYPE Arr_Type IS VARRAY(10) OF NUMBER;
    v_Numbers Arr_Type := Arr_Type();
BEGIN
    v_Numbers.EXTEND(4);
    v_Numbers (1) := 10;
    v_Numbers (2) := 20;
    v_Numbers (3) := 30;
    v_Numbers (4) := 40;

    DBMS_OUTPUT.PUT_LINE(NVL(v_Numbers.prior (3400), -1));
    DBMS_OUTPUT.PUT_LINE(NVL(v_Numbers.next (3400), -1));
END;
/

Result:

4
-1

For an associative array indexed by string, the prior and next indexes are determined by key values, which are in sorted order (for more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String"). Example 6-1 uses FIRST, NEXT, and a WHILE LOOP statement to print the elements of an associative array.

Example 6-34 PRIOR and NEXT Methods

This example initializes a nested table with six elements, deletes the fourth element, and then shows the values of PRIOR and NEXT for elements 1 through 7. Elements 4 and 7 do not exist. Element 2 exists, despite its null value.

DECLARE
    TYPE nt_type IS TABLE OF NUMBER;
    nt nt_type := nt_type(18, NULL, 36, 45, 54, 63);
BEGIN
    nt.DELETE(4);
    DBMS_OUTPUT.PUT_LINE('nt(4) was deleted.');

    FOR i IN 1..7 LOOP
        DBMS_OUTPUT.PUT('nt.PRIOR(' || i || ' || ') = ');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.PRIOR(i)), 'NULL'));
        DBMS_OUTPUT.PUT('nt.NEXT(' || i || ' || ') = ');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.NEXT(i)), 'NULL'));
    END LOOP;
END;
/

Result:

nt(4) was deleted.
nt.PRIOR(1) = NULL
nt.NEXT(1) = 2
nt.PRIOR(2) = 1
nt.NEXT(2) = 3
nt.PRIOR(3) = 2
Example 6-35  Printing Elements of Sparse Nested Table

This example prints the elements of a sparse nested table from first to last, using FIRST and NEXT, and from last to first, using LAST and PRIOR.

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    n NumList := NumList(1, 2, NULL, NULL, 5, NULL, 7, 8, 9, NULL);
    idx INTEGER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('First to last:');
    idx := n.FIRST;
    WHILE idx IS NOT NULL LOOP
        DBMS_OUTPUT.PUT('n(' || idx || ') = ');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(n(idx)), 'NULL'));
        idx := n.NEXT(idx);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('--------------');
    DBMS_OUTPUT.PUT_LINE('Last to first:');
    idx := n.LAST;
    WHILE idx IS NOT NULL LOOP
        DBMS_OUTPUT.PUT('n(' || idx || ') = ');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(n(idx)), 'NULL'));
        idx := n.PRIOR(idx);
    END LOOP;
END;
/

Result:

First to last:
n(1) = 1
n(2) = 2
n(3) = NULL
n(4) = NULL
n(5) = 5
n(6) = NULL
n(7) = 7
n(8) = 8
n(9) = 9
n(10) = NULL
--------------

Last to first:
n(10) = NULL
n(9) = 9
n(8) = 8
n(7) = 7
n(6) = NULL
6.11 Collection Types Defined in Package Specifications

A collection type defined in a package specification is incompatible with an identically defined local or standalone collection type.

**Note:**

The examples in this topic define packages and procedures, which are explained in *PL/SQL Packages* and *PL/SQL Subprograms*, respectively.

**Example 6-36   Identically Defined Package and Local Collection Types**

In this example, the package specification and the anonymous block define the collection type *NumList* identically. The package defines a procedure, *print_numlist*, which has a *NumList* parameter. The anonymous block declares the variable *n1* of the type *pkg.NumList* (defined in the package) and the variable *n2* of the type *NumList* (defined in the block). The anonymous block can pass *n1* to *print_numlist*, but it cannot pass *n2* to *print_numlist*.

**Live SQL:**

You can view and run this example on Oracle Live SQL at Identically Defined Package and Local Collection Types

```sql
CREATE OR REPLACE PACKAGE pkg AS
    TYPE NumList IS TABLE OF NUMBER;
    PROCEDURE print_numlist (nums NumList);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
    PROCEDURE print_numlist (nums NumList) IS
    BEGIN
        FOR i IN nums.FIRST..nums.LAST LOOP
            DBMS_OUTPUT.PUT_LINE(nums(i));
        END LOOP;
    END;
END pkg;
/
DECLARE
    TYPE NumList IS TABLE OF NUMBER;  -- local type identical to package type
    n1 pkg.NumList := pkg.NumList(2,4);  -- package type
    n2     NumList :=     NumList(6,8);  -- local type
BEGIN
    pkg.print_numlist(n1);  -- succeeds
    pkg.print_numlist(n2);  -- fails
```
Example 6-37  Identically Defined Package and Standalone Collection Types

This example defines a standalone collection type `NumList` that is identical to the collection type `NumList` defined in the package specification in Example 6-36. The anonymous block declares the variable `n1` of the type `pkg.NumList` (defined in the package) and the variable `n2` of the standalone type `NumList`. The anonymous block can pass `n1` to `print_numlist`, but it cannot pass `n2` to `print_numlist`.

```
CREATE OR REPLACE TYPE NumList IS TABLE OF NUMBER;
    -- standalone collection type identical to package type
/
DECLARE
    n1 pkg.NumList := pkg.NumList(2,4);  -- package type
    n2     NumList :=     NumList(6,8);  -- standalone type
BEGIN
    pkg.print_numlist(n1);  -- succeeds
    pkg.print_numlist(n2);  -- fails
END;
/
```

**Result:**

```
pkg.print_numlist(n2);  -- fails
*
ERROR at line 7:
ORA-06550: line 7, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_NUMLIST'
ORA-06550: line 7, column 3:
PL/SQL: Statement ignored
```

6.12 Record Variables

You can create a record variable in any of these ways:

- Define a RECORD type and then declare a variable of that type.
• Use %ROWTYPE to declare a record variable that represents either a full or partial row of a database table or view.
• Use %TYPE to declare a record variable of the same type as a previously declared record variable.

For syntax and semantics, see "Record Variable Declaration".

Topics
• Initial Values of Record Variables
• Declaring Record Constants
• RECORD Types
• Declaring Items using the %ROWTYPE Attribute

6.12.1 Initial Values of Record Variables

For a record variable of a RECORD type, the initial value of each field is NULL unless you specify a different initial value for it when you define the type.

For a record variable declared with %ROWTYPE or %TYPE, the initial value of each field is NULL. The variable does not inherit the initial value of the referenced item.

6.12.2 Declaring Record Constants

When declaring a record constant, you must create a function that populates the record with its initial value and then invoke the function in the constant declaration.

Example 6-38 Declaring Record Constant

This example creates a function that populates the record with its initial value and then invoke the function in the constant declaration.

```sql
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
    TYPE My_Rec IS RECORD (a NUMBER, b NUMBER);
    FUNCTION Init_My_Rec RETURN My_Rec;
END My_Types;
/
CREATE OR REPLACE PACKAGE BODY My_Types IS
    FUNCTION Init_My_Rec RETURN My_Rec IS
        Rec My_Rec;
    BEGIN
        Rec.a := 0;
        Rec.b := 1;
        RETURN Rec;
    END Init_My_Rec;
END My_Types;
/
DECLARE
```

Live SQL:
You can view and run this example on Oracle Live SQL at Declaring Record Constant

```sql
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
    TYPE My_Rec IS RECORD (a NUMBER, b NUMBER);
    FUNCTION Init_My_Rec RETURN My_Rec;
END My_Types;
/
CREATE OR REPLACE PACKAGE BODY My_Types IS
    FUNCTION Init_My_Rec RETURN My_Rec IS
        Rec My_Rec;
    BEGIN
        Rec.a := 0;
        Rec.b := 1;
        RETURN Rec;
    END Init_My_Rec;
END My_Types;
/
DECLARE
```
r CONSTANT My_Types.My_Rec := My_Types.Init_My_Rec();
BEGIN
DBMS_OUTPUT.PUT_LINE('r.a = ' || r.a);
DBMS_OUTPUT.PUT_LINE('r.b = ' || r.b);
END;
/

Result:
r.a = 0
r.b = 1

PL/SQL procedure successfully completed.

6.12.3 RECORD Types

A RECORD type defined in a PL/SQL block is a local type. It is available only in the block, and is stored in the database only if the block is in a standalone or package subprogram.

A RECORD type defined in a package specification is a public item. You can reference it from outside the package by qualifying it with the package name (package_name.type_name). It is stored in the database until you drop the package with the DROP PACKAGE statement.

You cannot create a RECORD type at schema level. Therefore, a RECORD type cannot be an ADT attribute data type.

To define a RECORD type, specify its name and define its fields. To define a field, specify its name and data type. By default, the initial value of a field is NULL. You can specify the NOT NULL constraint for a field, in which case you must also specify a non-NULL initial value. Without the NOT NULL constraint, a non-NULL initial value is optional.

A RECORD type defined in a package specification is incompatible with an identically defined local RECORD type.

See Also:

- PL/SQL Packages
- PL/SQL Subprograms
- Nested, Package, and Standalone Subprograms
- Example 6-42, ""

Example 6-39   RECORD Type Definition and Variable Declaration

This example defines a RECORD type named DeptRecTyp, specifying an initial value for each field. Then it declares a variable of that type named dept_rec and prints its fields.
DECLARE
    TYPE DeptRecTyp IS RECORD (
        dept_id NUMBER(4) NOT NULL := 10,
        dept_name VARCHAR2(30) NOT NULL := 'Administration',
        mgr_id NUMBER(6) := 200,
        loc_id NUMBER(4) := 1700
    );

    dept_rec DeptRecTyp;
BEGIN
    DBMS_OUTPUT.PUT_LINE('dept_id:   ' || dept_rec.dept_id);
    DBMS_OUTPUT.PUT_LINE('dept_name: ' || dept_rec.dept_name);
    DBMS_OUTPUT.PUT_LINE('mgr_id:    ' || dept_rec.mgr_id);
    DBMS_OUTPUT.PUT_LINE('loc_id:    ' || dept_rec.loc_id);
END;
/

Result:
dept_id:  10
dept_name: Administration
mgr_id:  200
loc_id:  1700

Example 6-40  RECORD Type with RECORD Field (Nested Record)

This example defines two RECORD types, name_rec and contact. The type contact has a field of type name_rec.

DECLARE
    TYPE name_rec IS RECORD (
        first employees.first_name%TYPE,
        last employees.last_name%TYPE
    );

    TYPE contact IS RECORD (
        name name_rec,    -- nested record
        phone employees.phone_number%TYPE
    );

    friend contact;
BEGIN
    friend.name.first := 'John';
    friend.name.last := 'Smith';
friend.phone := '1-650-555-1234';

DBMS_OUTPUT.PUT_LINE ('friend.name.first ||', ' ', ||
friend.name.last ||',', ' ', ||
friend.phone
);
END;
/

Result:
John Smith, 1-650-555-1234

**Example 6-41    RECORD Type with Varray Field**

This defines a VARRAY type, full_name, and a RECORD type, contact. The type contact has a field of type full_name.

### Live SQL:

You can view and run this example on Oracle Live SQL at RECORD Type with Varray Field

DECLARE
  TYPE full_name IS VARRAY(2) OF VARCHAR2(20);

  TYPE contact IS RECORD
    (name full_name := full_name('John', 'Smith'), -- varray field
     phone employees.phone_number%TYPE
    );

  friend contact;
BEGIN
  friend.phone := '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE ('friend.name(1) ||', ' ', ||
friend.name(2) ||',', ' ', ||
friend.phone
);
END;
/

Result:
John Smith, 1-650-555-1234

**Example 6-42    Identically Defined Package and Local RECORD Types**

In this example, the package pkg and the anonymous block define the RECORD type rec_type identically. The package defines a procedure, print_rec_type, which has a rec_type parameter. The anonymous block declares the variable r1 of the package type (pkg.rec_type) and the variable r2 of the local type (rec_type). The anonymous block can pass r1 to print_rec_type, but it cannot pass r2 to print_rec_type.
Live SQL:

You can view and run this example on Oracle Live SQL at Identically Defined Package and Local RECORD Types

CREATE OR REPLACE PACKAGE pkg AS
  TYPE rec_type IS RECORD ( -- package RECORD type
     f1 INTEGER,
     f2 VARCHAR2(4)
  );
  PROCEDURE print_rec_type (rec rec_type);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_rec_type (rec rec_type) IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE(rec.f1);
      DBMS_OUTPUT.PUT_LINE(rec.f2);
    END;
  END pkg;
/
DECLARE
  TYPE rec_type IS RECORD ( -- local RECORD type
     f1 INTEGER,
     f2 VARCHAR2(4)
  );
  r1 pkg.rec_type;                -- package type
  r2     rec_type;                -- local type
BEGIN
  r1.f1 := 10; r1.f2 := 'abcd';
  r2.f1 := 25; r2.f2 := 'wxyz';
  pkg.print_rec_type(r1);  -- succeeds
  pkg.print_rec_type(r2);  -- fails
END;
/

Result:

pkg.print_rec_type(r2);  -- fails
*
ERROR at line 14:
ORA-06550: line 14, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_REC_TYPE'

6.12.4 Declaring Items using the %ROWTYPE Attribute

The %ROWTYPE attribute lets you declare a record variable that represents either a full or partial row of a database table or view.

For the syntax and semantics details, see %ROWTYPE Attribute.

Topics

- Declaring a Record Variable that Always Represents Full Row
• Declaring a Record Variable that Can Represent Partial Row
• %ROWTYPE Attribute and Virtual Columns
• %ROWTYPE Attribute and Invisible Columns

6.12.4.1 Declaring a Record Variable that Always Represents Full Row

To declare a record variable that always represents a full row of a database table or view, use this syntax:

```
variable_name table_or_view_name%ROWTYPE;
```

For every column of the table or view, the record has a field with the same name and data type.

See Also:

"%ROWTYPE Attribute" for more information about %ROWTYPE

Example 6-43  %ROWTYPE Variable Represents Full Database Table Row

This example declares a record variable that represents a row of the table `departments`, assigns values to its fields, and prints them. Compare this example to Example 6-39.

```
DECLARE
  dept_rec departments%ROWTYPE;
BEGIN
  -- Assign values to fields:
  dept_rec.department_id := 10;
  dept_rec.department_name := 'Administration';
  dept_rec.manager_id := 200;
  dept_rec.location_id := 1700;

  -- Print fields:
  DBMS_OUTPUT.PUT_LINE('dept_id:   ' || dept_rec.department_id);
  DBMS_OUTPUT.PUT_LINE('dept_name: ' || dept_rec.department_name);
  DBMS_OUTPUT.PUT_LINE('mgr_id:    ' || dept_rec.manager_id);
  DBMS_OUTPUT.PUT_LINE('loc_id:    ' || dept_rec.location_id);
END;
/
```

Result:

department_id: 10
department_name: Administration
Example 6-44  %ROWTYPE Variable Does Not Inherit Initial Values or Constraints

This example creates a table with two columns, each with an initial value and a \texttt{NOT NULL} constraint. Then it declares a record variable that represents a row of the table and prints its fields, showing that they did not inherit the initial values or \texttt{NOT NULL} constraints.

```sql
DROP TABLE t1;
CREATE TABLE t1 (  
c1 INTEGER DEFAULT 0 NOT NULL,
c2 INTEGER DEFAULT 1 NOT NULL
);
DECLARE
  t1_row t1%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT('t1.c1 = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(t1_row.c1), 'NULL'));
  DBMS_OUTPUT.PUT('t1.c2 = ');
  print(t1_row.c2);
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(t1_row.c2), 'NULL'));
END;
/
```

Result:

- \texttt{t1.c1 = NULL}
- \texttt{t1.c2 = NULL}

6.12.4.2 Declaring a Record Variable that Can Represent Partial Row

To declare a record variable that can represent a partial row of a database table or view, use this syntax:

```sql
variable_name cursor%ROWTYPE;
```

A cursor is associated with a query. For every column that the query selects, the record variable must have a corresponding, type-compatible field. If the query selects every column of the table or view, then the variable represents a full row; otherwise, the variable represents a partial row. The cursor must be either an explicit cursor or a strong cursor variable.
Example 6-45  %ROWTYPE Variable Represents Partial Database Table Row

This example defines an explicit cursor whose query selects only the columns `first_name`, `last_name`, and `phone_number` from the `employees` table in the sample schema `HR`. Then the example declares a record variable that has a field for each column that the cursor selects. The variable represents a partial row of `employees`. Compare this example to Example 6-40.

```
DECLARE
  CURSOR c IS
    SELECT first_name, last_name, phone_number
    FROM employees;

  friend c%ROWTYPE;
BEGIN
  friend.first_name   := 'John';
  friend.last_name    := 'Smith';
  friend.phone_number := '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE (        
      friend.first_name || ',' || 
      friend.last_name  || ',' || 
      friend.phone_number
  );
END;
/
```

Result:

```
John Smith, 1-650-555-1234
```

Example 6-46  %ROWTYPE Variable Represents Join Row

This example defines an explicit cursor whose query is a join and then declares a record variable that has a field for each column that the cursor selects.
DECLARE 
CURSOR c2 IS 
    SELECT employee_id, email, employees.manager_id, location_id 
    FROM employees, departments 
    WHERE employees.department_id = departments.department_id;

    join_rec c2%ROWTYPE; -- includes columns from two tables

BEGIN 
    NULL;
END;
/

6.12.4.3 %ROWTYPE Attribute and Virtual Columns

If you use the %ROWTYPE attribute to define a record variable that represents a full row of a table that has a virtual column, then you cannot insert that record into the table. Instead, you must insert the individual record fields into the table, excluding the virtual column.

Example 6-47    Inserting %ROWTYPE Record into Table (Wrong)

This example creates a record variable that represents a full row of a table that has a virtual column, populates the record, and inserts the record into the table, causing ORA-54013.

DROP TABLE plch_departure;

CREATE TABLE plch_departure ( 
    destination    VARCHAR2(100),
    departure_time DATE,
    delay          NUMBER(10),
    expected       GENERATED ALWAYS AS (departure_time + delay/24/60/60) 
);

DECLARE 
    dep_rec plch_departure%ROWTYPE;
BEGIN 
    dep_rec.destination := 'X';
    dep_rec.departure_time := SYSDATE;
    dep_rec.delay := 1500;

    INSERT INTO plch_departure VALUES dep_rec;
END;
/

Result: 

DECLARE 
* 
ERROR at line 1:
Example 6-48 Inserting %ROWTYPE Record into Table (Right)

This solves the problem in Example 6-47 by inserting the individual record fields into the table, excluding the virtual column.

DECLARE
    dep_rec plch_departure%rowtype;
BEGIN
    dep_rec.destination := 'X';
    dep_rec.departure_time := SYSDATE;
    dep_rec.delay := 1500;

    INSERT INTO plch_departure (destination, departure_time, delay)
    VALUES (dep_rec.destination, dep_rec.departure_time, dep_rec.delay);
end;
/

Result:
PL/SQL procedure successfully completed.

6.12.4.4 %ROWTYPE Attribute and Invisible Columns

Suppose that you use the %ROWTYPE attribute to define a record variable that represents a row of a table that has an invisible column, and then you make the invisible column visible.

If you define the record variable with a cursor, as in "Declaring a Record Variable that Can Represent Partial Row", then making the invisible column visible does not change the structure of the record variable.

However, if you define the record variable as in "Declaring a Record Variable that Always Represents Full Row" and use a SELECT * INTO statement to assign values to the record, then making the invisible column visible does change the structure of the record—see Example 6-49.

See Also:
Oracle Database SQL Language Reference for general information about invisible columns

Example 6-49 %ROWTYPE Affected by Making Invisible Column Visible

CREATE TABLE t (a INT, b INT, c INT INVISIBLE);
INSERT INTO t (a, b, c) VALUES (1, 2, 3);
COMMIT;

DECLARE
    t_rec t%ROWTYPE;  -- t_rec has fields a and b, but not c
BEGIN
    SELECT * INTO t_rec FROM t WHERE ROWNUM < 2;  -- t_rec(a)=1, t_rec(b)=2
    DBMS_OUTPUT.PUT_LINE('c = ' || t_rec.c);
END;
/
Result:

```
DBMS_OUTPUT.PUT_LINE('c = ' || t_rec.c);
```

ERROR at line 5:
ORA-06550: line 5, column 40:
`PLS-00302: component 'C' must be declared`
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored

Make invisible column visible:

```
ALTER TABLE t MODIFY (c VISIBLE);
```

Result:

Table altered.

Repeat preceding anonymous block:

```
DECLARE
  t_rec t%ROWTYPE;  -- t_rec has fields a, b, and c
BEGIN
  SELECT * INTO t_rec FROM t WHERE ROWNUM < 2;  -- t_rec(a)=1, t_rec(b)=2,
  -- t_rec(c)=3
  DBMS_OUTPUT.PUT_LINE('c = ' || t_rec.c);
END;
/
```

Result:

```
c = 3
```

PL/SQL procedure successfully completed.

### 6.13 Assigning Values to Record Variables

A record variable means either a record variable or a record component of a composite variable.

To any record variable, you can assign a value to each field individually.

You can assign values using qualified expressions (see Example 6-11).

In some cases, you can assign the value of one record variable to another record variable.

If a record variable represents a full or partial row of a database table or view, you can assign the represented row to the record variable.

**Topics**

- Assigning One Record Variable to Another
- Assigning Full or Partial Rows to Record Variables
- Assigning NULL to a Record Variable
6.13.1 Assigning One Record Variable to Another

You can assign the value of one record variable to another record variable only in these cases:

- The two variables have the same RECORD type.
- The target variable is declared with a RECORD type, the source variable is declared with %ROWTYPE, their fields match in number and order, and corresponding fields have the same data type.

For record components of composite variables, the types of the composite variables need not match.

**Example 6-50 Assigning Record to Another Record of Same RECORD Type**

In this example, name1 and name2 have the same RECORD type, so you can assign the value of name1 to name2.

```sql
DECLARE
    TYPE name_rec IS RECORD (  
        first  employees.first_name%TYPE DEFAULT 'John',  
        last   employees.last_name%TYPE DEFAULT 'Doe'
    );

    name1 name_rec;
    name2 name_rec;
BEGIN
    name1.first := 'Jane'; name1.last := 'Smith';
    DBMS_OUTPUT.PUT_LINE('name1: ' || name1.first || ' ' || name1.last);
    name2 := name1;
    DBMS_OUTPUT.PUT_LINE('name2: ' || name2.first || ' ' || name2.last);
END;
/
```

Result:

```
name1: Jane Smith
name2: Jane Smith
```

**Example 6-51 Assigning %ROWTYPE Record to RECORD Type Record**

In this example, the target variable is declared with a RECORD type, the source variable is declared with %ROWTYPE, their fields match in number and order, and corresponding fields have the same data type.

```sql
DECLARE
    TYPE name_rec IS RECORD (  
        first  employees.first_name%TYPE DEFAULT 'John',  
        last   employees.last_name%TYPE DEFAULT 'Doe'
    );

    CURSOR c IS
        SELECT first_name, last_name
        FROM employees;

    target name_rec;
    source c%ROWTYPE;
```
BEGIN
    source.first_name := 'Jane'; source.last_name := 'Smith';

    DBMS_OUTPUT.PUT_LINE ( 
        'source: ' || source.first_name || ' ' || source.last_name
    );

    target := source;

    DBMS_OUTPUT.PUT_LINE ( 
        'target: ' || target.first || ' ' || target.last
    );
    END;
/

Result:
source: Jane Smith
target: Jane Smith

Example 6-52  Assigning Nested Record to Another Record of Same RECORD Type

This example assigns the value of one nested record to another nested record. The nested records have the same RECORD type, but the records in which they are nested do not.

DECLARE
    TYPE name_rec IS RECORD ( 
        first  employees.first_name%TYPE,
        last   employees.last_name%TYPE
    );

    TYPE phone_rec IS RECORD ( 
        name name_rec, -- nested record
        phone employees.phone_number%TYPE
    );

    TYPE email_rec IS RECORD ( 
        name name_rec, -- nested record
        email employees.email%TYPE
    );

    phone_contact phone_rec;
    email_contact email_rec;

BEGIN
    phone_contact.name.first := 'John';
    phone_contact.name.last := 'Smith';
    phone_contact.phone := '1-650-555-1234';

    email_contact.name := phone_contact.name;
    email_contact.email := ( 
        email_contact.name.first || '.' || 
        email_contact.name.last || '@' || 
        'example.com'
    );

    DBMS_OUTPUT.PUT_LINE (email_contact.email);
    END;
/
6.13.2 Assigning Full or Partial Rows to Record Variables

If a record variable represents a full or partial row of a database table or view, you can assign the represented row to the record variable.

Topics

- Using SELECT INTO to Assign a Row to a Record Variable
- Using FETCH to Assign a Row to a Record Variable
- Using SQL Statements to Return Rows in PL/SQL Record Variables

6.13.2.1 Using SELECT INTO to Assign a Row to a Record Variable

The syntax of a simple SELECT INTO statement is:

```
SELECT select_list INTO record_variable_name
FROM table_or_view_name;
```

For each column in `select_list`, the record variable must have a corresponding, type-compatible field. The columns in `select_list` must appear in the same order as the record fields.

---

**See Also:**

"SELECT INTO Statement" for complete syntax

---

**Example 6-53  SELECT INTO Assigns Values to Record Variable**

In this example, the record variable `rec1` represents a partial row of the `employees` table—the columns `last_name` and `employee_id`. The `SELECT INTO` statement selects from `employees` the row for which `job_id` is 'AD_PRES' and assigns the values of the columns `last_name` and `employee_id` in that row to the corresponding fields of `rec1`.

```
DECLARE
  TYPE RecordTyp IS RECORD (
    last employees.last_name%TYPE,
    id   employees.employee_id%TYPE
  );
  rec1 RecordTyp;
BEGIN
  SELECT last_name, employee_id INTO rec1
  FROM employees
  WHERE job_id = 'AD_PRES';

  DBMS_OUTPUT.PUT_LINE ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

**Result:**

Employee #100 = King
6.13.2.2 Using FETCH to Assign a Row to a Record Variable

The syntax of a simple `FETCH` statement is:

```
FETCH cursor INTO record_variable_name;
```

A cursor is associated with a query. For every column that the query selects, the record variable must have a corresponding, type-compatible field. The cursor must be either an explicit cursor or a strong cursor variable.

See Also:

- "FETCH Statement" for complete syntax
- "Cursors Overview" for information about all cursors
- "Explicit Cursors" for information about explicit cursors
- "Cursor Variables" for information about cursor variables

Example 6-54  FETCH Assigns Values to Record that Function Returns

In this example, each variable of RECORD type `EmpRecTyp` represents a partial row of the `employees` table—the columns `employee_id` and `salary`. Both the cursor and the function return a value of type `EmpRecTyp`. In the function, a `FETCH` statement assigns the values of the columns `employee_id` and `salary` to the corresponding fields of a local variable of type `EmpRecTyp`.

```sql
DECLARE
  TYPE EmpRecTyp IS RECORD (  
    emp_id  employees.employee_id%TYPE,
    salary  employees.salary%TYPE
  );

  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

  highest_paid_emp       EmpRecTyp;
  next_highest_paid_emp  EmpRecTyp;

  FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
    emp_rec  EmpRecTyp;
    BEGIN
      OPEN desc_salary;
      FOR i IN 1..n LOOP
        FETCH desc_salary INTO emp_rec;
      END LOOP;
      CLOSE desc_salary;
      RETURN emp_rec;
    END nth_highest_salary;

  BEGIN
    highest_paid_emp := nth_highest_salary(1);
    next_highest_paid_emp := nth_highest_salary(2);
```

```sql```
DBMS_OUTPUT.PUT_LINE(
    'Highest Paid: #' ||
    highest_paid_emp.emp_id || ', $' ||
    highest_paid_emp.salary
);
DBMS_OUTPUT.PUT_LINE(
    'Next Highest Paid: #' ||
    next_highest_paid_emp.emp_id || ', $' ||
    next_highest_paid_emp.salary
);
END;
/

Result:

Highest Paid: #100, $24000
Next Highest Paid: #101, $17000

6.13.2.3 Using SQL Statements to Return Rows in PL/SQL Record Variables

The SQL statements INSERT, UPDATE, and DELETE have an optional RETURNING INTO clause that can return the affected row in a PL/SQL record variable. For information about this clause, see "RETURNING INTO Clause".

Example 6-55 UPDATE Statement Assigns Values to Record Variable

In this example, the UPDATE statement updates the salary of an employee and returns the name and new salary of the employee in a record variable.

DECLARE
    TYPE EmpRec IS RECORD (
        last_name employees.last_name%TYPE,
        salary employees.salary%TYPE
    );
    emp_info EmpRec;
    old_salary employees.salary%TYPE;
BEGIN
    SELECT salary INTO old_salary
    FROM employees
    WHERE employee_id = 100;

    UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id = 100
    RETURNING last_name, salary INTO emp_info;

    DBMS_OUTPUT.PUT_LINE ( 
        'Salary of ' || emp_info.last_name || ' raised from ' ||
        old_salary || ' to ' || emp_info.salary
    );
END;
/

Result:

Salary of King raised from 24000 to 26400
6.13.3 Assigning NULL to a Record Variable

Assigning the value NULL to a record variable assigns the value NULL to each of its fields.

This assignment is recursive; that is, if a field is a record, then its fields are also assigned the value NULL.

Example 6-56  Assigning NULL to Record Variable

This example prints the fields of a record variable (one of which is a record) before and after assigning NULL to it.

DECLARE
    TYPE age_rec IS RECORD (
        years  INTEGER DEFAULT 35,
        months INTEGER DEFAULT 6
    );

    TYPE name_rec IS RECORD (
        first  employees.first_name%TYPE DEFAULT 'John',
        last   employees.last_name%TYPE DEFAULT 'Doe',
        age    age_rec
    );

    name name_rec;

PROCEDURE print_name AS
BEGIN
    DBMS_OUTPUT.PUT(NVL(name.first, 'NULL') || ' ');
    DBMS_OUTPUT.PUT(NVL(name.last,  'NULL') || ', ');
    DBMS_OUTPUT.PUT(NVL(TO_CHAR(name.age.years), 'NULL') || ' yrs ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(name.age.months), 'NULL') || ' mos');
END;
BEGIN
    print_name;
    name := NULL;
    print_name;
END;
/

Result:

John Doe, 35 yrs 6 mos
NULL NULL, NULL yrs NULL mos

6.14 Record Comparisons

Records cannot be tested natively for nullity, equality, or inequality.

These BOOLEAN expressions are illegal:

- My_Record IS NULL
- My_Record_1 = My_Record_2
- My_Record_1 > My_Record_2
You must write your own functions to implement such tests. For information about writing functions, see PL/SQL Subprograms.

6.15 Inserting Records into Tables

The PL/SQL extension to the SQL INSERT statement lets you insert a record into a table.

The record must represent a row of the table. For more information, see "INSERT Statement Extension". For restrictions on inserting records into tables, see "Restrictions on Record Inserts and Updates".

To efficiently insert a collection of records into a table, put the INSERT statement inside a FORALL statement. For information about the FORALL statement, see "FORALL Statement".

Example 6-57    Initializing Table by Inserting Record of Default Values

This example creates the table schedule and initializes it by putting default values in a record and inserting the record into the table for each week. (The COLUMN formatting commands are from SQL*Plus.)

```
DROP TABLE schedule;
CREATE TABLE schedule (  
    week  NUMBER,  
    Mon   VARCHAR2(10),  
    Tue   VARCHAR2(10),  
    Wed   VARCHAR2(10),  
    Thu   VARCHAR2(10),  
    Fri   VARCHAR2(10),  
    Sat   VARCHAR2(10),  
    Sun   VARCHAR2(10)  
);  

DECLARE  
    default_week  schedule%ROWTYPE;  
    i             NUMBER;  
BEGIN  
    default_week.Mon := '0800-1700';  
    default_week.Tue := '0800-1700';  
    default_week.Wed := '0800-1700';  
    default_week.Thu := '0800-1700';  
    default_week.Fri := '0800-1700';  
    default_week.Sat := 'Day Off';  
    default_week.Sun := 'Day Off';  

    FOR i IN 1..6 LOOP  
        default_week.week    := i;  
        INSERT INTO schedule VALUES default_week;  
    END LOOP;  
END;
/
```

COLUMN week FORMAT 99
COLUMN Mon  FORMAT A9
COLUMN Tue  FORMAT A9
COLUMN Wed  FORMAT A9
COLUMN Thu  FORMAT A9
COLUMN Fri  FORMAT A9
SELECT * FROM schedule;

Result:

<table>
<thead>
<tr>
<th>WEEK</th>
<th>MON</th>
<th>TUE</th>
<th>WED</th>
<th>THU</th>
<th>FRI</th>
<th>SAT</th>
<th>SUN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>Day Off</td>
<td>Day Off</td>
</tr>
<tr>
<td>2</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>Day Off</td>
<td>Day Off</td>
</tr>
<tr>
<td>3</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>Day Off</td>
<td>Day Off</td>
</tr>
<tr>
<td>4</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>Day Off</td>
<td>Day Off</td>
</tr>
<tr>
<td>5</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>Day Off</td>
<td>Day Off</td>
</tr>
<tr>
<td>6</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>0800-1700</td>
<td>Day Off</td>
<td>Day Off</td>
</tr>
</tbody>
</table>

### 6.16 Updating Rows with Records

The PL/SQL extension to the SQL `UPDATE` statement lets you update one or more table rows with a record.

The record must represent a row of the table. For more information, see "UPDATE Statement Extensions".

For restrictions on updating table rows with a record, see "Restrictions on Record Inserts and Updates".

To efficiently update a set of rows with a collection of records, put the `UPDATE` statement inside a `FORALL` statement. For information about the `FORALL` statement, see "FORALL Statement".

#### Example 6-58   Updating Rows with Record

This example updates the first three weeks of the table `schedule` (defined in Example 6-57) by putting the new values in a record and updating the first three rows of the table with that record.

```sql
DECLARE
  default_week schedule%ROWTYPE;
BEGIN
  default_week.Mon := 'Day Off';
  default_week.Tue := '0900-1800';
  default_week.Wed := '0900-1800';
  default_week.Thu := '0900-1800';
  default_week.Fri := '0900-1800';
  default_week.Sat := '0900-1800';
  default_week.Sun := 'Day Off';

  FOR i IN 1..3 LOOP
    default_week.week := i;

    UPDATE schedule
    SET ROW = default_week
    WHERE week = i;
  END LOOP;
END;
/

SELECT * FROM schedule;
```
6.17 Restrictions on Record Inserts and Updates

These restrictions apply to record inserts and updates:

- Record variables are allowed only in these places:
  - On the right side of the `SET` clause in an `UPDATE` statement
  - In the `VALUES` clause of an `INSERT` statement
  - In the `INTO` subclause of a `RETURNING` clause

Record variables are not allowed in a `SELECT` list, `WHERE` clause, `GROUP BY` clause, or `ORDER BY` clause.

- The keyword `ROW` is allowed only on the left side of a `SET` clause. Also, you cannot use `ROW` with a subquery.

- In an `UPDATE` statement, only one `SET` clause is allowed if `ROW` is used.

- If the `VALUES` clause of an `INSERT` statement contains a record variable, no other variable or value is allowed in the clause.

- If the `INTO` subclause of a `RETURNING` clause contains a record variable, no other variable or value is allowed in the subclause.

- These are not supported:
  - Nested `RECORD` types
  - Functions that return a `RECORD` type
  - Record inserts and updates using the `EXECUTE IMMEDIATE` statement.
PL/SQL Static SQL

**Static SQL** is a PL/SQL feature that allows SQL syntax directly in a PL/SQL statement.

This chapter describes static SQL and explains how to use it.

Topics

- Description of Static SQL
- Cursors Overview
- Processing Query Result Sets
- Cursor Variables
- CURSOR Expressions
- Transaction Processing and Control
- Autonomous Transactions

See Also:

"Resolution of Names in Static SQL Statements"

### 7.1 Description of Static SQL

Static SQL has the same syntax as SQL, except as noted.

Topics

- Statements
- Pseudocolumns

#### 7.1.1 Statements

These are the PL/SQL static SQL statements, which have the same syntax as the corresponding SQL statements, except as noted:

- **SELECT** (this statement is also called a **query**)
  
  For the PL/SQL syntax, see "SELECT INTO Statement".

- Data manipulation language (DML) statements:
  
  - **INSERT**
    
    For the PL/SQL syntax, see "INSERT Statement Extension".
  
  - **UPDATE**
For the PL/SQL syntax, see "UPDATE Statement Extensions".

– **DELETE**

For the PL/SQL syntax, see "DELETE Statement Extension".

– **MERGE** (for syntax, see *Oracle Database SQL Language Reference*)

### Note:
*Oracle Database SQL Language Reference* defines DML differently.

### Transaction control language (TCL) statements:

– **COMMIT** (for syntax, see *Oracle Database SQL Language Reference*)

– **ROLLBACK** (for syntax, see *Oracle Database SQL Language Reference*)

– **SAVEPOINT** (for syntax, see *Oracle Database SQL Language Reference*)

– **SET TRANSACTION** (for syntax, see *Oracle Database SQL Language Reference*)

– **LOCK TABLE** (for syntax, see *Oracle Database SQL Language Reference*)

A PL/SQL static SQL statement can have a PL/SQL identifier wherever its SQL counterpart can have a placeholder for a bind variable. The PL/SQL identifier must identify either a variable or a formal parameter.

To use PL/SQL identifiers for table names, column names, and so on, use the **EXECUTE IMMEDIATE** statement, explained in "Native Dynamic SQL"

### Note:
After PL/SQL code runs a DML statement, the values of some variables are undefined. For example:

- After a **FETCH** or **SELECT** statement raises an exception, the values of the define variables after that statement are undefined.

- After a DML statement that affects zero rows, the values of the **OUT** bind variables are undefined, unless the DML statement is a **BULK** or multiple-row operation.

### Example 7-1  Static SQL Statements

In this example, a PL/SQL anonymous block declares three PL/SQL variables and uses them in the static SQL statements **INSERT**, **UPDATE**, **DELETE**. The block also uses the static SQL statement **COMMIT**.

```sql
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
   SELECT employee_id, first_name, last_name
   FROM employees;

DECLARE
   emp_id          employees_temp.employee_id%TYPE := 299;
   emp_first_name  employees_temp.first_name%TYPE  := 'Bob';
   emp_last_name   employees_temp.last_name%TYPE   := 'Henry';
```
BEGIN
  INSERT INTO employees_temp (employee_id, first_name, last_name)
  VALUES (emp_id, emp_first_name, emp_last_name);

  UPDATE employees_temp
  SET first_name = 'Robert'
  WHERE employee_id = emp_id;

  DELETE FROM employees_temp
  WHERE employee_id = emp_id
  RETURNING first_name, last_name
  INTO emp_first_name, emp_last_name;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE (emp_first_name || ' ' || emp_last_name);
END; /

Result:
Robert Henry

7.1.2 Pseudocolumns

A pseudocolumn behaves like a table column, but it is not stored in the table.

For general information about pseudocolumns, including restrictions, see Oracle Database SQL Language Reference.

Static SQL includes these SQL pseudocolumns:

• CURRVAL and NEXTVAL, described in "CURRVAL and NEXTVAL in PL/SQL".
• LEVEL, described in Oracle Database SQL Language Reference
• OBJECT_VALUE, described in Oracle Database SQL Language Reference

See Also:

"OBJECT_VALUE Pseudocolumn" for information about using OBJECT_VALUE in triggers

• ROWID, described in Oracle Database SQL Language Reference

See Also:

"Simulating CURRENT OF Clause with ROWID Pseudocolumn"

• ROWNUM, described in Oracle Database SQL Language Reference

7.1.2.1 CURRVAL and NEXTVAL in PL/SQL

After a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn, which returns the current value of the sequence, or the NEXTVAL pseudocolumn, which increments the sequence and returns the new value.
To reference these pseudocolumns, use dot notation—for example, `sequence_name.CURRVAL`.

**Note:**

Each time you reference `sequence_name.NEXTVAL`, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

You can use `sequence_name.CURRVAL` and `sequence_name.NEXTVAL` in a PL/SQL expression wherever you can use a `NUMBER` expression. However:

- Using `sequence_name.CURRVAL` or `sequence_name.NEXTVAL` to provide a default value for an ADT method parameter causes a compilation error.
- PL/SQL evaluates every occurrence of `sequence_name.CURRVAL` and `sequence_name.NEXTVAL` (unlike SQL, which evaluates a sequence expression for every row in which it appears).

**See Also:**

- *Oracle Database SQL Language Reference* for general information about sequences
- *Oracle Database SQL Language Reference* for `CURRVAL` and `NEXTVAL` complete syntax

**Example 7-2  CURRVAL and NEXTVAL Pseudocolumns**

This example generates a sequence number for the sequence `HR.EMPLOYEES_SEQ` and refers to that number in multiple statements.

```sql
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
   SELECT employee_id, first_name, last_name
   FROM employees;

DROP TABLE employees_temp2;
CREATE TABLE employees_temp2 AS
   SELECT employee_id, first_name, last_name
   FROM employees;

DECLARE
   seq_value NUMBER;
BEGIN
   -- Generate initial sequence number
   seq_value := employees_seq.NEXTVAL;
   -- Print initial sequence number:
   DBMS_OUTPUT.PUT_LINE ( 'Initial sequence value: ' || TO_CHAR(seq_value) );

   -- Use sequence values in PL/SQL expressions
   -- (e.g., for inserting new records)

   -- Insert a record using the sequence value
   INSERT INTO employees_temp (employee_id, first_name, last_name)
   VALUES (seq_value, 'John Doe', 'Doe');

   -- Query using the sequence value
   SELECT employee_id, first_name, last_name
   FROM employees_temp
   WHERE employee_id = seq_value;

   -- Update a record using the sequence value
   UPDATE employees_temp
   SET first_name = 'Jane Doe'
   WHERE employee_id = seq_value;

   -- Delete a record using the sequence value
   DELETE FROM employees_temp
   WHERE employee_id = seq_value;

END;
```

---

**Chapter 7**

**Description of Static SQL**

7-4
-- Use NEXTVAL to create unique number when inserting data:

```sql
INSERT INTO employees_temp (employee_id, first_name, last_name)
VALUES (employees_seq.NEXTVAL, 'Lynette', 'Smith');
```

-- Use CURRVAL to store same value somewhere else:

```sql
INSERT INTO employees_temp2 VALUES (employees_seq.CURRVAL, 'Morgan', 'Smith');
```

/* Because NEXTVAL values might be referenced by different users and applications, and some NEXTVAL values might not be stored in database, there might be gaps in sequence. */

-- Use CURRVAL to specify record to delete:

```
seq_value := employees_seq.CURRVAL;
```

```sql
DELETE FROM employees_temp2
WHERE employee_id = seq_value;
```

-- Update employee_id with NEXTVAL for specified record:

```sql
UPDATE employees_temp
SET employee_id = employees_seq.NEXTVAL
WHERE first_name = 'Lynette'
AND last_name = 'Smith';
```

-- Display final value of CURRVAL:

```
seq_value := employees_seq.CURRVAL;
```

```sql
DBMS_OUTPUT.PUT_LINE ( 'Ending sequence value: ' || TO_CHAR(seq_value) );
```

END;
/

### 7.2 Cursors Overview

A **cursor** is a pointer to a private SQL area that stores information about processing a specific **SELECT** or **DML** statement.

---

**Note:**

The cursors that this topic explains are session cursors. A **session cursor** lives in session memory until the session ends, when it ceases to exist.

---

A cursor that is constructed and managed by PL/SQL is an **implicit cursor**. A cursor that you construct and manage is an **explicit cursor**.

You can get information about any session cursor from its attributes (which you can reference in procedural statements, but not in SQL statements).
To list the session cursors that each user session currently has opened and parsed, query the dynamic performance view `V$OPEN_CURSOR`.

The number of cursors that a session can have open simultaneously is determined by:

- The amount of memory available to the session
- The value of the initialization parameter `OPEN_CURSORS`

**Note:**
Generally, PL/SQL parses an explicit cursor only the first time the session opens it and parses a SQL statement (creating an implicit cursor) only the first time the statement runs.

All parsed SQL statements are cached. A SQL statement is reparsed only if it is aged out of the cache by a new SQL statement. Although you must close an explicit cursor before you can reopen it, PL/SQL need not reparse the associated query. If you close and immediately reopen an explicit cursor, PL/SQL does not reparse the associated query.

Topics

- Implicit Cursors
- Explicit Cursors

**See Also:**

- *Oracle Database Reference* for information about the dynamic performance view `V$OPEN_CURSOR`
- *Oracle Database Reference* for information about the initialization parameter `OPEN_CURSORS`

### 7.2.1 Implicit Cursors

An **implicit cursor** is a session cursor that is constructed and managed by PL/SQL. PL/SQL opens an implicit cursor every time you run a `SELECT` or DML statement. You cannot control an implicit cursor, but you can get information from its attributes.

The syntax of an implicit cursor attribute value is `SQLattribute` (therefore, an implicit cursor is also called a **SQL cursor**). `SQLattribute` always refers to the most recently run `SELECT` or DML statement. If no such statement has run, the value of `SQLattribute` is `NULL`.

An implicit cursor closes after its associated statement runs; however, its attribute values remain available until another `SELECT` or DML statement runs.

The most recently run `SELECT` or DML statement might be in a different scope. To save an attribute value for later use, assign it to a local variable immediately. Otherwise,
other operations, such as subprogram invocations, might change the value of the attribute before you can test it.

The implicit cursor attributes are:

- **SQL%ISOPEN Attribute**: Is the Cursor Open?
- **SQL%FOUND Attribute**: Were Any Rows Affected?
- **SQL%NOTFOUND Attribute**: Were No Rows Affected?
- **SQL%ROWCOUNT Attribute**: How Many Rows Were Affected?
- **SQL%BULK_ROWCOUNT** (see "Getting Number of Rows Affected by FORALL Statement"
- **SQL%BULK_EXCEPTIONS** (see "Handling FORALL Exceptions After FORALL Statement Completes"

7.2.1.1 SQL%ISOPEN Attribute: Is the Cursor Open?

**SQL%ISOPEN** always returns **FALSE**, because an implicit cursor always closes after its associated statement runs.

7.2.1.2 SQL%FOUND Attribute: Were Any Rows Affected?

**SQL%FOUND** returns:

- **NULL** if no **SELECT** or DML statement has run
- **TRUE** if a **SELECT** statement returned one or more rows or a DML statement affected one or more rows
- **FALSE** otherwise

**Example 7-3** uses **SQL%FOUND** to determine if a **DELETE** statement affected any rows.

**Example 7-3**  SQL%FOUND Implicit Cursor Attribute

```sql
DROP TABLE dept_temp;
CREATE TABLE dept_temp AS
    SELECT * FROM departments;

CREATE OR REPLACE PROCEDURE p (dept_no NUMBER)
    AUTHID CURRENT_USER AS
BEGIN
    DELETE FROM dept_temp
    WHERE department_id = dept_no;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('Delete succeeded for department number ' || dept_no);
    ELSE
```

See Also:

"Implicit Cursor Attribute" for complete syntax and semantics
DBMS_OUTPUT.PUT_LINE ('No department number ' || dept_no);
END IF;
END;
/
BEGIN
    p(270);
    p(400);
END;
/

Result:
Delete succeeded for department number 270
No department number 400

7.2.1.3 SQL%NOTFOUND Attribute: Were No Rows Affected?

SQL%NOTFOUND (the logical opposite of SQL%FOUND) returns:

- NULL if no SELECT or DML statement has run
- FALSE if a SELECT statement returned one or more rows or a DML statement affected one or more rows
- TRUE otherwise

The SQL%NOTFOUND attribute is not useful with the PL/SQL SELECT INTO statement, because:

- If the SELECT INTO statement returns no rows, PL/SQL raises the predefined exception NO_DATA_FOUND immediately, before you can check SQL%NOTFOUND.
- A SELECT INTO statement that invokes a SQL aggregate function always returns a value (possibly NULL). After such a statement, the SQL%NOTFOUND attribute is always FALSE, so checking it is unnecessary.

7.2.1.4 SQL%ROWCOUNT Attribute: How Many Rows Were Affected?

SQL%ROWCOUNT returns:

- NULL if no SELECT or DML statement has run
- Otherwise, the number of rows returned by a SELECT statement or affected by a DML statement (an INTEGER)

Note:

If a server is Oracle Database 12c or later and its client is Oracle Database 11g release 2 or earlier (or the reverse), then the maximum number that SQL%ROWCOUNT returns is 4,294,967,295.

Example 7-4 uses SQL%ROWCOUNT to determine the number of rows that were deleted.

If a SELECT INTO statement without a BULK COLLECT clause returns multiple rows, PL/SQL raises the predefined exception TOO_MANY_ROWS and SQL%ROWCOUNT returns 1, not the actual number of rows that satisfy the query.
The value of SQL%ROWCOUNT attribute is unrelated to the state of a transaction. Therefore:

• When a transaction rolls back to a savepoint, the value of SQL%ROWCOUNT is not restored to the value it had before the savepoint.
• When an autonomous transaction ends, SQL%ROWCOUNT is not restored to the original value in the parent transaction.

Example 7-4    SQL%ROWCOUNT Implicit Cursor Attribute

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT * FROM employees;

DECLARE
  mgr_no NUMBER(6) := 122;
BEGIN
  DELETE FROM employees_temp WHERE manager_id = mgr_no;
  DBMS_OUTPUT.PUT_LINE ('Number of employees deleted: ' || TO_CHAR(SQ\L\%ROWCOUNT));
END;
/
```

Result:

Number of employees deleted: 8

7.2.2 Explicit Cursors

An explicit cursor is a session cursor that you construct and manage. You must declare and define an explicit cursor, giving it a name and associating it with a query (typically, the query returns multiple rows). Then you can process the query result set in either of these ways:

• Open the explicit cursor (with the OPEN statement), fetch rows from the result set (with the FETCH statement), and close the explicit cursor (with the CLOSE statement).
• Use the explicit cursor in a cursor FOR LOOP statement (see “Processing Query Result Sets With Cursor FOR LOOP Statements”).

You cannot assign a value to an explicit cursor, use it in an expression, or use it as a formal subprogram parameter or host variable. You can do those things with a cursor variable (see “Cursor Variables”).

Unlike an implicit cursor, you can reference an explicit cursor or cursor variable by its name. Therefore, an explicit cursor or cursor variable is called a named cursor.

Topics

• Declaring and Defining Explicit Cursors
• Opening and Closing Explicit Cursors
• Fetching Data with Explicit Cursors
• Variables in Explicit Cursor Queries
• When Explicit Cursor Queries Need Column Aliases
• Explicit Cursors that Accept Parameters
7.2.2.1 Declaring and Defining Explicit Cursors

You can either declare an explicit cursor first and then define it later in the same block, subprogram, or package, or declare and define it at the same time.

An **explicit cursor declaration**, which only declares a cursor, has this syntax:

```
CURSOR cursor_name [ parameter_list ] RETURN return_type;
```

An **explicit cursor definition** has this syntax:

```
CURSOR cursor_name [ parameter_list ] [ RETURN return_type ]
    IS select_statement;
```

If you declared the cursor earlier, then the explicit cursor definition defines it; otherwise, it both declares and defines it.

**Example 7-5** declares and defines three explicit cursors.

---

### See Also:

- "Explicit Cursor Declaration and Definition" for the complete syntax and semantics of explicit cursor declaration and definition
- "Explicit Cursors that Accept Parameters"

---

**Example 7-5  Explicit Cursor Declaration and Definition**

```sql
DECLARE
    CURSOR c1 RETURN departments%ROWTYPE;  -- Declare c1

    CURSOR c2 IS
        SELECT employee_id, job_id, salary FROM employees
        WHERE salary > 2000;

    CURSOR c1 RETURN departments%ROWTYPE IS  -- Define c1,
        SELECT * FROM departments
        WHERE department_id = 110;

    CURSOR c3 RETURN locations%ROWTYPE;      -- Declare c3

    CURSOR c3 IS
        SELECT * FROM locations
        WHERE country_id = 'JP';

BEGIN
    NULL;
END;
/
```

---

7.2.2.2 Opening and Closing Explicit Cursors

After declaring and defining an explicit cursor, you can open it with the **OPEN** statement, which does the following:

1. Allocates database resources to process the query
2. Processes the query; that is:

   a. Identifies the result set

      If the query references variables or cursor parameters, their values affect the result set. For details, see "Variables in Explicit Cursor Queries" and "Explicit Cursors that Accept Parameters".

   b. If the query has a FOR UPDATE clause, locks the rows of the result set

      For details, see "SELECT FOR UPDATE and FOR UPDATE Cursors".

3. Positions the cursor before the first row of the result set

   You close an open explicit cursor with the CLOSE statement, thereby allowing its resources to be reused. After closing a cursor, you cannot fetch records from its result set or reference its attributes. If you try, PL/SQL raises the predefined exception INVALID_CURSOR.

   You can reopen a closed cursor. You must close an explicit cursor before you try to reopen it. Otherwise, PL/SQL raises the predefined exception CURSOR_ALREADY_OPEN.

7.2.2.3 Fetching Data with Explicit Cursors

   After opening an explicit cursor, you can fetch the rows of the query result set with the FETCH statement. The basic syntax of a FETCH statement that returns one row is:

   ```sql
   FETCH cursor_name INTO into_clause
   ```

   The into_clause is either a list of variables or a single record variable. For each column that the query returns, the variable list or record must have a corresponding type-compatible variable or field. The %TYPE and %ROWTYPE attributes are useful for declaring variables and records for use in FETCH statements.

   The FETCH statement retrieves the current row of the result set, stores the column values of that row into the variables or record, and advances the cursor to the next row.

   Typically, you use the FETCH statement inside a LOOP statement, which you exit when the FETCH statement runs out of rows. To detect this exit condition, use the cursor attribute %NOTFOUND (described in "%NOTFOUND Attribute: Has No Row Been Fetched"). PL/SQL does not raise an exception when a FETCH statement returns no rows.

   Example 7-6 fetches the result sets of two explicit cursors one row at a time, using FETCH and %NOTFOUND inside LOOP statements. The first FETCH statement retrieves column values into variables. The second FETCH statement retrieves column values into a record. The variables and record are declared with %TYPE and %ROWTYPE, respectively.
Example 7-7 fetches the first five rows of a result set into five records, using five `FETCH` statements, each of which fetches into a different record variable. The record variables are declared with `%ROWTYPE`.

**See Also:**
- "FETCH Statement" for its complete syntax and semantics
- "FETCH Statement with BULK COLLECT Clause" for information about FETCH statements that return more than one row at a time

**Example 7-6  FETCH Statements Inside LOOP Statements**

```
DECLARE
  CURSOR c1 IS
  SELECT last_name, job_id FROM employees
  WHERE REGEXP_LIKE (job_id, '[HT]_[CLERK]')
  ORDER BY last_name;

  v_lastname employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id

  CURSOR c2 IS
  SELECT * FROM employees
  WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_[M][ANGR]')
  ORDER BY job_id;

  v_employees employees%ROWTYPE;  -- record variable for row of table

BEGIN
  OPEN c1;
  LOOP
    -- Fetches 2 columns into variables
    FETCH c1 INTO v_lastname, v_jobid;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE( '-------------------------------------' );

  OPEN c2;
  LOOP
    -- Fetches entire row into the v_employees record
    FETCH c2 INTO v_employees;
    EXIT WHEN c2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') || v_employees.job_id );
  END LOOP;
  CLOSE c2;
END;
/
```

**Result:**

```
Atkinson                 ST_CLERK
Bell                     SH_CLERK
Bissot                   ST_CLERK
...                      
Walsh                    SH_CLERK
```
Example 7-7  Fetching Same Explicit Cursor into Different Variables

DECLARE
CURSOR c IS
  SELECT e.job_id, j.job_title
  FROM employees e, jobs j
  WHERE e.job_id = j.job_id AND e.manager_id = 100
  ORDER BY last_name;

-- Record variables for rows of cursor result set:
job1 c%ROWTYPE;
job2 c%ROWTYPE;
job3 c%ROWTYPE;
job4 c%ROWTYPE;
job5 c%ROWTYPE;
BEGIN
  OPEN c;
  FETCH c INTO job1;  -- fetches first row
  FETCH c INTO job2;  -- fetches second row
  FETCH c INTO job3;  -- fetches third row
  FETCH c INTO job4;  -- fetches fourth row
  FETCH c INTO job5;  -- fetches fifth row
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE(job1.job_title || ' (' || job1.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job2.job_title || ' (' || job2.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job3.job_title || ' (' || job3.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job4.job_title || ' (' || job4.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job5.job_title || ' (' || job5.job_id || ')');
END;
/

Result:
Sales Manager (SA_MAN)
Administration Vice President (AD_VP)
Sales Manager (SA_MAN)
Stock Manager (ST_MAN)
Marketing Manager (MK_MAN)

PL/SQL procedure successfully completed.

7.2.2.4 Variables in Explicit Cursor Queries

An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.
In Example 7-8, the explicit cursor query references the variable `factor`. When the cursor opens, `factor` has the value 2. Therefore, `sal_multiple` is always 2 times `sal`, despite that `factor` is incremented after every fetch.

To change the result set, you must close the cursor, change the value of the variable, and then open the cursor again, as in Example 7-9.

**Example 7-8  Variable in Explicit Cursor Query—No Result Set Change**

```
DECLARE
    sal     employees.salary%TYPE;
    sal_multiple employees.salary%TYPE;
    factor   INTEGER := 2;
CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_';
BEGIN
    OPEN c1;  -- PL/SQL evaluates factor
    LOOP
        FETCH c1 INTO sal, sal_multiple;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
        DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
        DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
        factor := factor + 1;  -- Does not affect sal_multiple
    END LOOP;
    CLOSE c1;
END;
/
```

Result:

```
factor = 2
sal    = 4400
sal_multiple = 8800
factor = 3
sal    = 24000
sal_multiple = 48000
factor = 4
sal    = 17000
sal_multiple = 34000
factor = 5
sal    = 17000
sal_multiple = 34000
```

**Example 7-9  Variable in Explicit Cursor Query—Result Set Change**

```
DECLARE
    sal     employees.salary%TYPE;
    sal_multiple employees.salary%TYPE;
    factor   INTEGER := 2;
CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_';
BEGIN
```

```
```
DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
OPEN c1;  -- PL/SQL evaluates factor
LOOP
  FETCH c1 INTO sal, sal_multiple;
  EXIT WHEN c1%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
  DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
END LOOP;
CLOSE c1;

factor := factor + 1;

DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
OPEN c1;  -- PL/SQL evaluates factor
LOOP
  FETCH c1 INTO sal, sal_multiple;
  EXIT WHEN c1%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
  DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
END LOOP;
CLOSE c1;
END;
/

Result:
factor = 2
sal = 4400
sal_multiple = 8800
sal = 24000
sal_multiple = 48000
sal = 17000
sal_multiple = 34000
sal = 17000
sal_multiple = 34000
factor = 3
sal = 4400
sal_multiple = 13200
sal = 24000
sal_multiple = 72000
sal = 17000
sal_multiple = 51000
sal = 17000
sal_multiple = 51000

7.2.2.5 When Explicit Cursor Queries Need Column Aliases

When an explicit cursor query includes a virtual column (an expression), that column must have an alias if either of the following is true:

- You use the cursor to fetch into a record that was declared with %ROWTYPE.
- You want to reference the virtual column in your program.

In Example 7-10, the virtual column in the explicit cursor needs an alias for both of the preceding reasons.
Example 7-10  Explicit Cursor with Virtual Column that Needs Alias

DECLARE
  CURSOR c1 IS
    SELECT employee_id,
      (salary * .05) raise
    FROM employees
    WHERE job_id LIKE '%_MAN'
    ORDER BY employee_id;
  emp_rec c1%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('Raise for employee #' || emp_rec.employee_id || ' is $' || emp_rec.raise);
  END LOOP;
  CLOSE c1;
END;
/

Result:

Raise for employee #114 is $550
Raise for employee #120 is $400
Raise for employee #121 is $410
Raise for employee #122 is $395
Raise for employee #123 is $325
Raise for employee #124 is $368.445
Raise for employee #145 is $700
Raise for employee #146 is $675
Raise for employee #147 is $600
Raise for employee #148 is $550
Raise for employee #149 is $525
Raise for employee #201 is $650

7.2.2.6 Explicit Cursors that Accept Parameters

You can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference formal cursor parameters.

Tip:

To avoid confusion, use different names for formal and actual cursor parameters.
Example 7-11 creates an explicit cursor whose two formal parameters represent a job and its maximum salary. When opened with a specified job and maximum salary, the cursor query selects the employees with that job who are overpaid (for each such employee, the query selects the first and last name and amount overpaid). Next, the example creates a procedure that prints the cursor query result set (for information about procedures, see PL/SQL Subprograms). Finally, the example opens the cursor with one set of actual parameters, prints the result set, closes the cursor, opens the cursor with different actual parameters, prints the result set, and closes the cursor.

Topics
• Formal Cursor Parameters with Default Values
• Adding Formal Cursor Parameters with Default Values

See Also:
• “Explicit Cursor Declaration and Definition” for more information about formal cursor parameters
• “OPEN Statement” for more information about actual cursor parameters

Example 7-11 Explicit Cursor that Accepts Parameters

DECLARE
CURSOR c (job VARCHAR2, max_sal NUMBER) IS
SELECT last_name, first_name, (salary - max_sal) overpayment
FROM employees
WHERE job_id = job
AND salary > max_sal
ORDER BY salary;
PROCEDURE print_overpaid IS
last_name_ employees.last_name%TYPE;
first_name_ employees.first_name%TYPE;
overpayment_ employees.salary%TYPE;
BEGIN
LOOP
FETCH c INTO last_name_, first_name_, overpayment_;
EXIT WHEN c%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ || ' (by ' || overpayment_ || ')');
END LOOP;
END print_overpaid;
BEGIN
DBMS_OUTPUT.PUT_LINE('----------------------');
DBMS_OUTPUT.PUT_LINE('Overpaid Stock Clerks:');
DBMS_OUTPUT.PUT_LINE('----------------------');
OPEN c('ST_CLERK', 5000);
print_overpaid;
CLOSE c;

DBMS_OUTPUT.PUT_LINE('-------------------------------');
DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
DBMS_OUTPUT.PUT_LINE('-------------------------------');
OPEN c('SA_REP', 10000);


print_overpaid;
CLOSE c;
END;
/

Result:

----------------------
Overpaid Stock Clerks:
----------------------

Overpaid Sales Representatives:
-------------------------------
Vishney, Clara (by 500)
Abel, Ellen (by 1000)
Ozer, Lisa (by 1500)

PL/SQL procedure successfully completed.

7.2.2.6.1 Formal Cursor Parameters with Default Values

When you create an explicit cursor with formal parameters, you can specify default values for them. When a formal parameter has a default value, its corresponding actual parameter is optional. If you open the cursor without specifying the actual parameter, then the formal parameter has its default value.

**Example 7-12** creates an explicit cursor whose formal parameter represents a location ID. The default value of the parameter is the location ID of company headquarters.

**Example 7-12   Cursor Parameters with Default Values**

```
DECLARE
    CURSOR c (location NUMBER DEFAULT 1700) IS
        SELECT d.department_name,
               e.last_name manager,
               l.city
        FROM departments d, employees e, locations l
        WHERE l.location_id = location
            AND l.location_id = d.location_id
            AND d.department_id = e.department_id
        ORDER BY d.department_id;

    PROCEDURE print_depts IS
        dept_name  departments.department_name%TYPE;
        mgr_name   employees.last_name%TYPE;
        city_name  locations.city%TYPE;
    BEGIN
        LOOP
            FETCH c INTO dept_name, mgr_name, city_name;
            EXIT WHEN c%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(dept_name || ' (Manager: ' || mgr_name || ')');
        END LOOP;
    END print_depts;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS AT HEADQUARTERS:');
    DBMS_OUTPUT.PUT_LINE('--------------------------------');
    OPEN c;
    print_depts;
    DBMS_OUTPUT.PUT_LINE('--------------------------------');
    CLOSE c;
```
DBMS_OUTPUT.PUT_LINE('DEPARTMENTS IN CANADA:');
DBMS_OUTPUT.PUT_LINE('--------------------------------');
OPEN c(1800); -- Toronto
print_depts;
CLOSE c;
OPEN c(1900); -- Whitehorse
print_depts;
CLOSE c;
END;
/

Result is similar to:

DEPARTMENTS AT HEADQUARTERS:
--------------------------------
Administration (Manager: Whalen)
Purchasing (Manager: Colmenares)
Purchasing (Manager: Baida)
Purchasing (Manager: Himuro)
Purchasing (Manager: Raphaely)
Purchasing (Manager: Khoo)
Purchasing (Manager: Tobias)
Executive (Manager: Kochhar)
Executive (Manager: De Haan)
Executive (Manager: King)
Finance (Manager: Popp)
Finance (Manager: Greenberg)
Finance (Manager: Faviet)
Finance (Manager: Chen)
Finance (Manager: Urman)
Finance (Manager: Sciarra)
Accounting (Manager: Gietz)
Accounting (Manager: Higgins)
--------------------------------
DEPARTMENTS IN CANADA:
--------------------------------
Marketing (Manager: Hartstein)
Marketing (Manager: Fay)

PL/SQL procedure successfully completed.

7.2.2.6.2 Adding Formal Cursor Parameters with Default Values

If you add formal parameters to a cursor, and you specify default values for the added parameters, then you need not change existing references to the cursor. Compare Example 7-13 to Example 7-11.

Example 7-13 Adding Formal Parameter to Existing Cursor

DECLARE
CURSOR c (job VARCHAR2, max_sal NUMBER,
    hired DATE DEFAULT TO_DATE('31-DEC-1999', 'DD-MON-YYYY')) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    AND hire_date > hired
    ORDER BY salary;
PROCEDURE print_overpaid IS
  last_name_ employees.last_name%TYPE;
  first_name_ employees.first_name%TYPE;
  overpayment_ employees.salary%TYPE;
BEGIN
  LOOP
    FETCH c INTO last_name_, first_name_, overpayment_;
    EXIT WHEN c%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
      ' (by ' || overpayment_ || ')');
  END LOOP;
END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('-------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:
  -----------------------------');
  OPEN c('SA_REP', 10000);  -- existing reference
  print_overpaid;
  CLOSE c;
  DBMS_OUTPUT.PUT_LINE('-------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives Hired After 2004:
  -----------------------------');
  OPEN c('SA_REP', 10000, TO_DATE('31-DEC-2004', 'DD-MON-YYYY'));  -- new reference
  print_overpaid;
  CLOSE c;
END;
/

Result:

-------------------------------
Overpaid Sales Representatives:
-------------------------------
Vishney, Clara (by 500)
Abel, Ellen (by 1000)
Ozer, Lisa (by 1500)
-------------------------------
Overpaid Sales Representatives Hired After 2004:
-------------------------------
Vishney, Clara (by 500)
Ozer, Lisa (by 1500)
PL/SQL procedure successfully completed.

7.2.2.7 Explicit Cursor Attributes

The syntax for the value of an explicit cursor attribute is cursor_name immediately
followed by attribute (for example, c1%ISOPEN).

Note:

Explicit cursors and cursor variables (named cursors) have the same
attributes. This topic applies to all named cursors except where noted.
The explicit cursor attributes are:

- %ISOPEN Attribute: Is the Cursor Open?
- %FOUND Attribute: Has a Row Been Fetched?
- %NOTFOUND Attribute: Has No Row Been Fetched?
- %ROWCOUNT Attribute: How Many Rows Were Fetched?

If an explicit cursor is not open, referencing any attribute except %ISOPEN raises the predefined exception INVALID_CURSOR.

### See Also:

"Named Cursor Attribute" for complete syntax and semantics of named cursor (explicit cursor and cursor variable) attributes

#### 7.2.2.7.1 %ISOPEN Attribute: Is the Cursor Open?

%ISOPEN returns TRUE if its explicit cursor is open; FALSE otherwise.

%ISOPEN is useful for:

- Checking that an explicit cursor is not already open before you try to open it.
  
  If you try to open an explicit cursor that is already open, PL/SQL raises the predefined exception CURSOR_ALREADY_OPEN. You must close an explicit cursor before you can reopen it.

#### Note:

The preceding paragraph does not apply to cursor variables.

- Checking that an explicit cursor is open before you try to close it.

Example 7-14 opens the explicit cursor c1 only if it is not open and closes it only if it is open.

**Example 7-14 %ISOPEN Explicit Cursor Attribute**

```sql
DECLARE
  CURSOR c1 IS
  SELECT last_name, salary FROM employees
  WHERE ROWNUM < 11;

  the_name employees.last_name%TYPE;
  the_salary employees.salary%TYPE;
BEGIN
  IF NOT c1%ISOPEN THEN
    OPEN c1;
  END IF;

  FETCH c1 INTO the_name, the_salary;

  IF c1%ISOPEN THEN
    CLOSE c1;
```
7.2.2.7.2 %FOUND Attribute: Has a Row Been Fetched?

%FOUND returns:

• NULL after the explicit cursor is opened but before the first fetch
• TRUE if the most recent fetch from the explicit cursor returned a row
• FALSE otherwise

%FOUND is useful for determining whether there is a fetched row to process.

Example 7-15 loops through a result set, printing each fetched row and exiting when there are no more rows to fetch.

Example 7-15 %FOUND Explicit Cursor Attribute

DECLARE
CURSOR c1 IS
  SELECT last_name, salary FROM employees
  WHERE ROWNUM < 11
  ORDER BY last_name;

my_ename employees.last_name%TYPE;
my_salary employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%FOUND THEN  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE('Name = ' || my_ename || ', salary = ' || my_salary);
    ELSE  -- fetch failed
      EXIT;
    END IF;
  END LOOP;
END;
/

Result:
Name = Austin, salary = 4800
Name = De Haan, salary = 17000
Name = Ernst, salary = 6000
Name = Faviet, salary = 9000
Name = Greenberg, salary = 12008
Name = Hunold, salary = 9000
Name = King, salary = 24000
Name = Kochhar, salary = 17000
Name = Lorentz, salary = 4200
Name = Pataballa, salary = 4800

7.2.2.7.3 %NOTFOUND Attribute: Has No Row Been Fetched?

%NOTFOUND (the logical opposite of %FOUND) returns:

• NULL after the explicit cursor is opened but before the first fetch
• FALSE if the most recent fetch from the explicit cursor returned a row
%NOTFOUND is useful for exiting a loop when FETCH fails to return a row, as in Example 7-16.

**Example 7-16  %NOTFOUND Explicit Cursor Attribute**

```
DECLARE
    CURSOR c1 IS
        SELECT last_name, salary FROM employees
        WHERE ROWNUM < 11
        ORDER BY last_name;

    my_ename   employees.last_name%TYPE;
    my_salary  employees.salary%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO my_ename, my_salary;
        IF c1%NOTFOUND THEN -- fetch failed
            EXIT;
        ELSE  -- fetch succeeded
            DBMS_OUTPUT.PUT_LINE('Name = ' || my_ename || ', salary = ' || my_salary);
        END IF;
    END LOOP;
END;
/
```

*Result:*

```
Name = Austin, salary = 4800
Name = De Haan, salary = 17000
Name = Ernst, salary = 6000
Name = Faviet, salary = 9000
Name = Greenberg, salary = 12008
Name = Hunold, salary = 9000
Name = King, salary = 24000
Name = Kochhar, salary = 17000
Name = Lorentz, salary = 4200
Name = Pataballa, salary = 4800
```

7.2.2.7.4 %ROWCOUNT Attribute: How Many Rows Were Fetched?

%ROWCOUNT returns:

- Zero after the explicit cursor is opened but before the first fetch
- Otherwise, the number of rows fetched (an INTEGER)

**Note:**

If a server is Oracle Database 12c or later and its client is Oracle Database 11g2 or earlier (or the reverse), then the maximum number that SQL%ROWCOUNT returns is 4,294,967,295.

**Example 7-17** numbers and prints the rows that it fetches and prints a message after fetching the fifth row.
Example 7-17  %ROWCOUNT Explicit Cursor Attribute

DECLARE
    CURSOR c1 IS
        SELECT last_name FROM employees
        WHERE ROWNUM < 11
        ORDER BY last_name;

    name employees.last_name%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO name;
        EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
        DBMS_OUTPUT.PUT_LINE(c1%ROWCOUNT || ' ' || name);
        IF c1%ROWCOUNT = 5 THEN
            DBMS_OUTPUT.PUT_LINE('--- Fetched 5th row ---');
        END IF;
    END LOOP;
    CLOSE c1;
END;
/

Result:
1. Abel
2. Ande
3. Atkinson
4. Austin
5. Baer
--- Fetched 5th row ---
6. Baida
7. Banda
8. Bates
9. Bell
10. Bernstein

7.3 Processing Query Result Sets

In PL/SQL, as in traditional database programming, you use cursors to process query result sets. However, in PL/SQL, you can use either implicit or explicit cursors.

The former need less code, but the latter are more flexible. For example, explicit cursors can accept parameters.

The following PL/SQL statements use implicit cursors that PL/SQL defines and manages for you:

- SELECT INTO
- Implicit cursor FOR LOOP

The following PL/SQL statements use explicit cursors:

- Explicit cursor FOR LOOP
  You define the explicit cursor, but PL/SQL manages it while the statement runs.
- OPEN, FETCH, and CLOSE
  You define and manage the explicit cursor.
If a query returns no rows, PL/SQL raises the exception `NO_DATA_FOUND`.

Topics

- Processing Query Result Sets With SELECT INTO Statements
- Processing Query Result Sets With Cursor FOR LOOP Statements
- Processing Query Result Sets With Explicit Cursors, OPEN, FETCH, and CLOSE
- Processing Query Result Sets with Subqueries

See Also:

- “Explicit Cursors that Accept Parameters”
- *Oracle Database Development Guide* for information about returning result sets to clients
- “Exception Handler” for information about handling exceptions

### 7.3.1 Processing Query Result Sets With SELECT INTO Statements

Using an implicit cursor, the `SELECT INTO` statement retrieves values from one or more database tables (as the SQL `SELECT` statement does) and stores them in variables (which the SQL `SELECT` statement does not do).

Topics

- Handling Single-Row Result Sets
- Handling Large Multiple-Row Result Sets

See Also:

"SELECT INTO Statement“ for its complete syntax and semantics

#### 7.3.1.1 Handling Single-Row Result Sets

If you expect the query to return only one row, then use the `SELECT INTO` statement to store values from that row in either one or more scalar variables, or one record variable.

If the query might return multiple rows, but you care about only the *n*th row, then restrict the result set to that row with the clause `WHERE ROWNUM=n`.
7.3.1.2 Handling Large Multiple-Row Result Sets

If you must assign a large quantity of table data to variables, Oracle recommends using the SELECT INTO statement with the BULK COLLECT clause.

This statement retrieves an entire result set into one or more collection variables.
For more information, see "SELECT INTO Statement with BULK COLLECT Clause".

7.3.2 Processing Query Result Sets With Cursor FOR LOOP Statements

The cursor FOR LOOP statement lets you run a SELECT statement and then immediately loop through the rows of the result set.

This statement can use either an implicit or explicit cursor (but not a cursor variable).

If you use the SELECT statement only in the cursor FOR LOOP statement, then specify the SELECT statement inside the cursor FOR LOOP statement. This form of the cursor FOR LOOP statement uses an implicit cursor, and is called an implicit cursor FOR LOOP statement. Because the implicit cursor is internal to the statement, you cannot reference it with the name SQL.

If you use the SELECT statement multiple times in the same PL/SQL unit, then define an explicit cursor for it and specify that cursor in the cursor FOR LOOP statement. This form of the cursor FOR LOOP statement is called an explicit cursor FOR LOOP statement. You can use the same explicit cursor elsewhere in the same PL/SQL unit.

The cursor FOR LOOP statement implicitly declares its loop index as a %ROWTYPE record variable of the type that its cursor returns. This record is local to the loop and exists only during loop execution. Statements inside the loop can reference the record and its fields. They can reference virtual columns only by aliases.

After declaring the loop index record variable, the FOR LOOP statement opens the specified cursor. With each iteration of the loop, the FOR LOOP statement fetches a row from the result set and stores it in the record. When there are no more rows to fetch, the cursor FOR LOOP statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or if PL/SQL raises an exception.
Note:

When an exception is raised inside a cursor `FOR LOOP` statement, the cursor closes before the exception handler runs. Therefore, the values of explicit cursor attributes are not available in the handler.

Example 7-18  Implicit Cursor `FOR LOOP` Statement

In this example, an implicit cursor `FOR LOOP` statement prints the last name and job ID of every clerk whose manager has an ID greater than 120.

```
BEGIN
  FOR item IN (
    SELECT last_name, job_id
    FROM employees
    WHERE job_id LIKE '%CLERK%'
    AND manager_id > 120
    ORDER BY last_name
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE
    ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...  
Name = Walsh, Job = SH_CLERK

Example 7-19  Explicit Cursor `FOR LOOP` Statement

This example is like Example 7-18, except that it uses an explicit cursor `FOR LOOP` statement.

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE job_id LIKE '%CLERK%' AND manager_id > 120
    ORDER BY last_name;
BEGIN
  FOR item IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
    ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...  
Name = Walsh, Job = SH_CLERK
Example 7-20  Passing Parameters to Explicit Cursor FOR LOOP Statement

This example declares and defines an explicit cursor that accepts two parameters, and then uses it in an explicit cursor FOR LOOP statement to display the wages paid to employees who earn more than a specified wage in a specified department.

DECLARE
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
    WHERE job_id = job
    AND salary > max_wage;
BEGIN
  FOR person IN c1('ST_CLERK', 3000) LOOP
    -- process data record
    DBMS_OUTPUT.PUT_LINE ('Name = ' || person.last_name || ', salary = ' || person.salary || ', Job Id = ' || person.job_id);
  END LOOP;
END;
/

Result:
Name = Nayer, salary = 3200, Job Id = ST_CLERK
Name = Bissot, salary = 3300, Job Id = ST_CLERK
Name = Mallin, salary = 3300, Job Id = ST_CLERK
Name = Ladwig, salary = 3600, Job Id = ST_CLERK
Name = Stiles, salary = 3200, Job Id = ST_CLERK
Name = Rajs, salary = 3500, Job Id = ST_CLERK
Name = Davies, salary = 3100, Job Id = ST_CLERK

Example 7-21  Cursor FOR Loop References Virtual Columns

In this example, the implicit cursor FOR LOOP references virtual columns by their aliases, full_name and dream_salary.

BEGIN
  FOR item IN (SELECT first_name || ' ' || last_name AS full_name,
                salary * 10 AS dream_salary
    FROM employees
    WHERE ROWNUM <= 5
    ORDER BY dream_salary DESC, last_name ASC
  ) LOOP
    DBMS_OUTPUT.PUT_LINE (item.full_name || ' dreams of making ' || item.dream_salary);
  END LOOP;
END;
/

Result:
Stephen King dreams of making 240000
Lex De Haan dreams of making 170000
Neena Kochhar dreams of making 170000
Alexander Hunold dreams of making 90000
Bruce Ernst dreams of making 60000
7.3.3 Processing Query Result Sets With Explicit Cursors, OPEN, FETCH, and CLOSE

For full control over query result set processing, declare explicit cursors and manage them with the statements OPEN, FETCH, and CLOSE.

This result set processing technique is more complicated than the others, but it is also more flexible. For example, you can:

• Process multiple result sets in parallel, using multiple cursors.
• Process multiple rows in a single loop iteration, skip rows, or split the processing into multiple loops.
• Specify the query in one PL/SQL unit but retrieve the rows in another.

For instructions and examples, see "Explicit Cursors".

7.3.4 Processing Query Result Sets with Subqueries

If you process a query result set by looping through it and running another query for each row, then you can improve performance by removing the second query from inside the loop and making it a subquery of the first query.

While an ordinary subquery is evaluated for each table, a correlated subquery is evaluated for each row.

For more information about subqueries, see Oracle Database SQL Language Reference.

Example 7-22 Subquery in FROM Clause of Parent Query

This example defines explicit cursor c1 with a query whose FROM clause contains a subquery.

```
DECLARE
   CURSOR c1 IS
      SELECT t1.department_id, department_name, staff
      FROM departments t1,
         ( SELECT department_id, COUNT(*) AS staff
            FROM employees
            GROUP BY department_id
         ) t2
      WHERE (t1.department_id = t2.department_id) AND staff >= 5
      ORDER BY staff;

BEGIN
   FOR dept IN c1 LOOP
      DBMS_OUTPUT.PUT_LINE ('Department = ' || dept.department_name || ', staff = ' || dept.staff);
   END LOOP;
END;
/

Result:
Department = IT, staff = 5
Department = Finance, staff = 6
```
Example 7-23  Correlated Subquery

This example returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the table, the correlated subquery computes the average salary for the corresponding department.

DECLARE
CURSOR c1 IS
SELECT department_id, last_name, salary
FROM employees t
WHERE salary > (SELECT AVG(salary)
               FROM employees
               WHERE t.department_id = department_id)
ORDER BY department_id, last_name;
BEGIN
FOR person IN c1
LOOP
  DBMS_OUTPUT.PUT_LINE('Making above-average salary = ' || person.last_name);
END LOOP;
END;
/

Result:
Making above-average salary = Hartstein
Making above-average salary = Raphaely
Making above-average salary = Bell
...
Making above-average salary = Higgins

7.4 Cursor Variables

A cursor variable is like an explicit cursor, except that:

- It is not limited to one query.
  You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query.

- You can assign a value to it.

- You can use it in an expression.

- It can be a subprogram parameter.
  You can use cursor variables to pass query result sets between subprograms.

- It can be a host variable.
  You can use cursor variables to pass query result sets between PL/SQL stored subprograms and their clients.

- It cannot accept parameters.
  You cannot pass parameters to a cursor variable, but you can pass whole queries to it. The queries can include variables.
A cursor variable has this flexibility because it is a pointer; that is, its value is the address of an item, not the item itself.

Before you can reference a cursor variable, you must make it point to a SQL work area, either by opening it or by assigning it the value of an open PL/SQL cursor variable or open host cursor variable.

**Note:**

Cursor variables and explicit cursors are not interchangeable—you cannot use one where the other is expected.

**Topics**

- Creating Cursor Variables
- Opening and Closing Cursor Variables
- Fetching Data with Cursor Variables
- Assigning Values to Cursor Variables
- Variables in Cursor Variable Queries
- Querying a Collection
- Cursor Variable Attributes
- Cursor Variables as Subprogram Parameters
- Cursor Variables as Host Variables

**See Also:**

- "Explicit Cursors" for more information about explicit cursors
- "Restrictions on Cursor Variables"
- *Oracle Database Development Guide* for advantages of cursor variables
- *Oracle Database Development Guide* for disadvantages of cursor variables

### 7.4.1 Creating Cursor Variables

To create a cursor variable, either declare a variable of the predefined type `SYS_REFCURSOR` or define a `REF CURSOR` type and then declare a variable of that type.

**Note:**

Informally, a cursor variable is sometimes called a `REF CURSOR`.)
The basic syntax of a REF CURSOR type definition is:

```
TYPE type_name IS REF CURSOR [ RETURN return_type ]
```

For the complete syntax and semantics, see "Cursor Variable Declaration".

If you specify `return_type`, then the REF CURSOR type and cursor variables of that type are **strong**; if not, they are **weak**. SYS_REFCursor and cursor variables of that type are weak.

With a strong cursor variable, you can associate only queries that return the specified type. With a weak cursor variable, you can associate any query.

Weak cursor variables are more error-prone than strong ones, but they are also more flexible. Weak REF CURSOR types are interchangeable with each other and with the predefined type SYS_REFCursor. You can assign the value of a weak cursor variable to any other weak cursor variable.

You can assign the value of a strong cursor variable to another strong cursor variable only if both cursor variables have the same type (not merely the same return type).

---

**Note:**

You can partition weak cursor variable arguments to table functions only with the `PARTITION BY ANY` clause, not with `PARTITION BY RANGE` or `PARTITION BY HASH`.

For syntax and semantics, see "PARALLEL_ENABLE Clause".

---

### Example 7-24   Cursor Variable Declarations

This example defines strong and weak REF CURSOR types, variables of those types, and a variable of the predefined type SYS_REFCursor.

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE; -- strong type
  TYPE genericcurtyp IS REF CURSOR; -- weak type

  cursor1  empcurtyp;       -- strong cursor variable
  cursor2  genericcurtyp;   -- weak cursor variable
  my_cursor SYS_REFCursor;  -- weak cursor variable

  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE; -- strong type
  dept_cv deptcurtyp; -- strong cursor variable
BEGIN
  NULL;
END;
/
```

---

### Example 7-25   Cursor Variable with User-Defined Return Type

In this example, `EmpRecTyp` is a user-defined RECORD type.

```
DECLARE
  TYPE EmpRecTyp IS RECORD {
    employee_id NUMBER,
    last_name VARCHAR2(25),
    salary NUMBER(8,2))
```
7.4.2 Opening and Closing Cursor Variables

After declaring a cursor variable, you can open it with the OPEN FOR statement, which does the following:

1. Associates the cursor variable with a query (typically, the query returns multiple rows)
   
   The query can include placeholders for bind variables, whose values you specify in the USING clause of the OPEN FOR statement.

2. Allocates database resources to process the query

3. Processes the query; that is:
   
   a. Identifies the result set
      
      If the query references variables, their values affect the result set. For details, see "Variables in Cursor Variable Queries".
   
   b. If the query has a FOR UPDATE clause, locks the rows of the result set
      
      For details, see "SELECT FOR UPDATE and FOR UPDATE Cursors".

4. Positions the cursor before the first row of the result set

You need not close a cursor variable before reopening it (that is, using it in another OPEN FOR statement). After you reopen a cursor variable, the query previously associated with it is lost.

When you no longer need a cursor variable, close it with the CLOSE statement, thereby allowing its resources to be reused. After closing a cursor variable, you cannot fetch records from its result set or reference its attributes. If you try, PL/SQL raises the predefined exception INVALID_CURSOR.

You can reopen a closed cursor variable.

See Also:

- "OPEN FOR Statement" for its syntax and semantics
- "CLOSE Statement" for its syntax and semantics

7.4.3 Fetching Data with Cursor Variables

After opening a cursor variable, you can fetch the rows of the query result set with the FETCH statement.

The return type of the cursor variable must be compatible with the into_clause of the FETCH statement. If the cursor variable is strong, PL/SQL catches incompatibility at
If the cursor variable is weak, PL/SQL catches incompatibility at run time, raising the predefined exception \texttt{ROWTYPE\_MISMATCH} before the first fetch.

\textbf{See Also:}

- "Fetching Data with Explicit Cursors"
- "FETCH Statement" for its complete syntax and semantics
- "FETCH Statement with BULK COLLECT Clause" for information about FETCH statements that return more than one row at a time

\textbf{Example 7-26 Fetching Data with Cursor Variables}

This example uses one cursor variable to do what Example 7-6 does with two explicit cursors. The first \texttt{OPEN FOR} statement includes the query itself. The second \texttt{OPEN FOR} statement references a variable whose value is a query.

```sql
DECLARE
    cv SYS_REFCURSOR;  -- cursor variable
    v_lastname employees.last_name%TYPE;  -- variable for last_name
    v_jobid     employees.job_id%TYPE;     -- variable for job_id
    query_2 VARCHAR2(200) :=
        'SELECT * FROM employees
         WHERE REGEXP_LIKE (job_id, ''[ACADFIMKSA]_M[ANGR]'')
         ORDER BY job_id';
    v_employees employees%ROWTYPE;  -- record variable row of table
BEGIN
    OPEN cv FOR
        SELECT last_name, job_id FROM employees
        WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
        ORDER BY last_name;
    LOOP  -- Fetches 2 columns into variables
        FETCH cv INTO v_lastname, v_jobid;
        EXIT WHEN cv\_NOTFOUND;
        DBMS\_OUTPUT\_PUT\_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
    END LOOP;
    DBMS\_OUTPUT\_PUT\_LINE( '-------------------------------------' );
    OPEN cv FOR query_2;
    LOOP  -- Fetches entire row into the v_employees record
        FETCH cv INTO v_employees;
        EXIT WHEN cv\_NOTFOUND;
        DBMS\_OUTPUT\_PUT\_LINE( RPAD(v_employees.last_name, 25, ' ') ||
            v_employees.job_id );
    END LOOP;
    CLOSE cv;
END;
/"
Example 7-27  Fetching from Cursor Variable into Collections

This example fetches from a cursor variable into two collections (nested tables), using the BULK COLLECT clause of the FETCH statement.

```plsql
DECLARE
  TYPE empcurtyp IS REF CURSOR;
  TYPE namelist IS TABLE OF employees.last_name%TYPE;
  TYPE sallist IS TABLE OF employees.salary%TYPE;
  emp_cv  empcurtyp;
  names   namelist;
  sals    sallist;
BEGIN
  OPEN emp_cv FOR
    SELECT last_name, salary FROM employees
    WHERE job_id = 'SA_REP'
    ORDER BY salary DESC;
  FETCH emp_cv BULK COLLECT INTO names, sals;
  CLOSE emp_cv;
  -- loop through the names and sals collections
  FOR i IN names.FIRST .. names.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || names(i) || ', salary = ' || sals(i));
  END LOOP;
END;
/
```

Result:

Name = Ozer, salary = 11500
Name = Abel, salary = 11000
Name = Vishney, salary = 10500
... 
Name = Kumar, salary = 6100

7.4.4 Assigning Values to Cursor Variables

You can assign to a PL/SQL cursor variable the value of another PL/SQL cursor variable or host cursor variable.

The syntax is:

```
target_cursor_variable := source_cursor_variable;
```
If `source_cursor_variable` is open, then after the assignment, `target_cursor_variable` is also open. The two cursor variables point to the same SQL work area.

If `source_cursor_variable` is not open, opening `target_cursor_variable` after the assignment does not open `source_cursor_variable`.

### 7.4.5 Variables in Cursor Variable Queries

The query associated with a cursor variable can reference any variable in its scope.

When you open a cursor variable with the `OPEN` FOR statement, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

To change the result set, you must change the value of the variable and then open the cursor variable again for the same query, as in Example 7-29.

#### Example 7-28  Variable in Cursor Variable Query—No Result Set Change

This example opens a cursor variable for a query that references the variable `factor`, which has the value 2. Therefore, `sal_multiple` is always 2 times `sal`, despite that `factor` is incremented after every fetch.

```plsql
DECLARE
    sal           employees.salary%TYPE;
    sal_multiple  employees.salary%TYPE;
    factor        INTEGER := 2;
    cv SYS_REFCURSOR;
BEGIN
    OPEN cv FOR
        SELECT salary, salary*factor
        FROM employees
        WHERE job_id LIKE 'AD_%';  -- PL/SQL evaluates factor
    LOOP
        FETCH cv INTO sal, sal_multiple;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
        DBMS_OUTPUT.PUT_LINE('sal = ' || sal);
        DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
        factor := factor + 1;  -- Does not affect sal_multiple
    END LOOP;
    CLOSE cv;
END;
/  
Result:

factor = 2
sal = 4400
sal_multiple = 8800
factor = 3
sal = 24000
sal_multiple = 48000
factor = 4
sal = 17000
```
Example 7-29  Variable in Cursor Variable Query—Result Set Change

DECLARE
    sal          employees.salary%TYPE;
    sal_multiple  employees.salary%TYPE;
    factor        INTEGER := 2;
    cv SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

    OPEN cv FOR
      SELECT salary, salary*factor
      FROM employees
      WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor
    LOOP
      FETCH cv INTO sal, sal_multiple;
      EXIT WHEN cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
      DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    END LOOP;

    factor := factor + 1;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

    OPEN cv FOR
      SELECT salary, salary*factor
      FROM employees
      WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor
    LOOP
      FETCH cv INTO sal, sal_multiple;
      EXIT WHEN cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
      DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    END LOOP;

    CLOSE cv;
END;
/

Result:

factor = 2
sal   = 4400
sal_multiple = 8800
sal   = 24000
sal_multiple = 48000
sal   = 17000
sal_multiple = 34000
sal   = 17000
sal_multiple = 34000
factor = 3
7.4.6 Querying a Collection

You can query a collection if all of the following are true:

• The data type of the collection was either created at schema level or declared in a package specification.

• The data type of the collection element is either a scalar data type, a user-defined type, or a record type.

In the query FROM clause, the collection appears in table_collection_expression as the argument of the TABLE operator.

Note:

In SQL contexts, you cannot use a function whose return type was declared in a package specification.

See Also:

• Oracle Database SQL Language Reference for information about the table_collection_expression
• "CREATE PACKAGE Statement" for information about the CREATE PACKAGE statement
• "PL/SQL Collections and Records" for information about collection types and collection variables
• Example 8-9, "Querying a Collection with Native Dynamic SQL"

Example 7-30 Querying a Collection with Static SQL

In this example, the cursor variable is associated with a query on an associative array of records. The nested table type, mytab, is declared in a package specification.

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
  TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));
  TYPE mytab IS TABLE OF rec INDEX BY pls_integer;
END;

DECLARE
  v1 pkg.mytab;  -- collection of records
  v2 pkg.rec;
  c1 SYS_REFCURSOR;
BEGIN
  v1(1).f1 := 1;
  v1(1).f2 := 'one';
  OPEN c1 FOR SELECT * FROM TABLE(v1);
  FETCH c1 INTO v2;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE('Values in record are ' || v2.f1 || ' and ' || v2.f2);
END;
/

Result:
Values in record are 1 and one

7.4.7 Cursor Variable Attributes

A cursor variable has the same attributes as an explicit cursor (see Explicit Cursor Attributes). The syntax for the value of a cursor variable attribute is cursor_variable_name immediately followed by attribute (for example, cv%ISOPEN). If a cursor variable is not open, referencing any attribute except %ISOPEN raises the predefined exception INVALID_CURSOR.

7.4.8 Cursor Variables as Subprogram Parameters

You can use a cursor variable as a subprogram parameter, which makes it useful for passing query results between subprograms.

For example:

- You can open a cursor variable in one subprogram and process it in a different subprogram.
- In a multilanguage application, a PL/SQL subprogram can use a cursor variable to return a result set to a subprogram written in a different language.

Note:

The invoking and invoked subprograms must be in the same database instance. You cannot pass or return cursor variables to subprograms invoked through database links.

Caution:

Because cursor variables are pointers, using them as subprogram parameters increases the likelihood of subprogram parameter aliasing, which can have unintended results. For more information, see "Subprogram Parameter Aliasing with Cursor Variable Parameters".

When declaring a cursor variable as the formal parameter of a subprogram:

- If the subprogram opens or assigns a value to the cursor variable, then the parameter mode must be IN OUT.
If the subprogram only fetches from, or closes, the cursor variable, then the parameter mode can be either `IN` or `IN OUT`.

Corresponding formal and actual cursor variable parameters must have compatible return types. Otherwise, PL/SQL raises the predefined exception `ROWTYPE_MISMATCH`.

To pass a cursor variable parameter between subprograms in different PL/SQL units, define the `REF CURSOR` type of the parameter in a package. When the type is in a package, multiple subprograms can use it. One subprogram can declare a formal parameter of that type, and other subprograms can declare variables of that type and pass them to the first subprogram.

**See Also:**

- "Subprogram Parameters" for more information about subprogram parameters
- "CURSOR Expressions" for information about `CURSOR` expressions, which can be actual parameters for formal cursor variable parameters
- PL/SQL Packages, for more information about packages

**Example 7-31 Procedure to Open Cursor Variable for One Query**

This example defines, in a package, a `REF CURSOR` type and a procedure that opens a cursor variable parameter of that type.

```sql
CREATE OR REPLACE PACKAGE emp_data AUTHID DEFINER AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (
    emp_cv IN OUT empcurtyp
  ) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

**Example 7-32 Opening Cursor Variable for Chosen Query (Same Return Type)**

In this example, the stored procedure opens its cursor variable parameter for a chosen query. The queries have the same return type.

```sql
CREATE OR REPLACE PACKAGE emp_data AUTHID DEFINER AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
    END IF;
  END open_emp_cv;
END emp_data;
/
WHERE commission_pct IS NOT NULL;
ELSIF choice = 2 THEN
OPEN emp_cv FOR SELECT *
FROM employees
WHERE salary > 2500;
ELSIF choice = 3 THEN
OPEN emp_cv FOR SELECT *
FROM employees
WHERE department_id = 100;
END IF;
END;
END emp_data;
/

Example 7-33 Opening Cursor Variable for Chosen Query (Different Return Types)

In this example, the stored procedure opens its cursor variable parameter for a chosen query. The queries have the different return types.

CREATE OR REPLACE PACKAGE admin_data AUTHID DEFINER AS
  TYPE gencurtyp IS REF CURSOR;
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT);
END admin_data;
/
CREATE OR REPLACE PACKAGE BODY admin_data AS
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN generic_cv FOR SELECT * FROM employees;
    ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM departments;
    ELSIF choice = 3 THEN
      OPEN generic_cv FOR SELECT * FROM jobs;
    END IF;
  END;
END admin_data;
/

7.4.9 Cursor Variables as Host Variables

You can use a cursor variable as a host variable, which makes it useful for passing query results between PL/SQL stored subprograms and their clients.

When a cursor variable is a host variable, PL/SQL and the client (the host environment) share a pointer to the SQL work area that stores the result set.

To use a cursor variable as a host variable, declare the cursor variable in the host environment and then pass it as an input host variable (bind variable) to PL/SQL. Host cursor variables are compatible with any query return type (like weak PL/SQL cursor variables).

A SQL work area remains accessible while any cursor variable points to it, even if you pass the value of a cursor variable from one scope to another. For example, in Example 7-34, the Pro*C program passes a host cursor variable to an embedded PL/SQL anonymous block. After the block runs, the cursor variable still points to the SQL work area.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open
and fetch from it on the server side, and continue to fetch from it on the client side. You can also reduce network traffic with a PL/SQL anonymous block that opens or closes several host cursor variables in a single round trip. For example:

```plsql
/* PL/SQL anonymous block in host environment */
BEGIN
    OPEN :emp_cv FOR SELECT * FROM employees;
    OPEN :dept_cv FOR SELECT * FROM departments;
    OPEN :loc_cv FOR SELECT * FROM locations;
END;
/
```

Because the cursor variables still point to the SQL work areas after the PL/SQL anonymous block runs, the client program can use them. When the client program no longer needs the cursors, it can use a PL/SQL anonymous block to close them. For example:

```plsql
/* PL/SQL anonymous block in host environment */
BEGIN
    CLOSE :emp_cv;
    CLOSE :dept_cv;
    CLOSE :loc_cv;
END;
/
```

This technique is useful for populating a multiblock form, as in Oracle Forms. For example, you can open several SQL work areas in a single round trip, like this:

```plsql
/* PL/SQL anonymous block in host environment */
BEGIN
    OPEN :c1 FOR SELECT 1 FROM DUAL;
    OPEN :c2 FOR SELECT 1 FROM DUAL;
    OPEN :c3 FOR SELECT 1 FROM DUAL;
END;
/
```

**Note:**

If you bind a host cursor variable into PL/SQL from an Oracle Call Interface (OCI) client, then you cannot fetch from it on the server side unless you also open it there on the same server call.

**Example 7-34   Cursor Variable as Host Variable in Pro*C Client Program**

In this example, a Pro*C client program declares a cursor variable and a selector and passes them as host variables to a PL/SQL anonymous block, which opens the cursor variable for the selected query.

```sql
EXEC SQL BEGIN DECLARE SECTION;
    SQL_CURSOR generic_cv;  -- Declare host cursor variable.
    int choice;      -- Declare selector.
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :generic_cv;  -- Initialize host cursor variable.
                               -- Pass host cursor variable and selector to PL/SQL block.
/
EXEC SQL EXECUTE
BEGIN
IF :choice = 1 THEN
  OPEN :generic_cv FOR SELECT * FROM employees;
ELSIF :choice = 2 THEN
  OPEN :generic_cv FOR SELECT * FROM departments;
ELSIF :choice = 3 THEN
  OPEN :generic_cv FOR SELECT * FROM jobs;
END IF;
END; END-EXEC;

7.5 CURSOR Expressions

A CURSOR expression returns a nested cursor.

It has this syntax:

CURSOR ( subquery )

You can use a CURSOR expression in a SELECT statement that is not a subquery (as in Example 7-35) or pass it to a function that accepts a cursor variable parameter (see "Passing CURSOR Expressions to Pipelined Table Functions"). You cannot use a cursor expression with an implicit cursor.

See Also:

Oracle Database SQL Language Reference for more information about CURSOR expressions, including restrictions

Example 7-35    CURSOR Expression

This example declares and defines an explicit cursor for a query that includes a cursor expression. For each department in the departments table, the nested cursor returns the last name of each employee in that department (which it retrieves from the employees table).

DECLARE
  TYPE emp_cur_typ IS REF CURSOR;
  emp_cur emp_cur_typ;
  dept_name departments.department_name%TYPE;
  emp_name employees.last_name%TYPE;

CURSOR c1 IS
  SELECT department_name,
       CURSOR ( SELECT e.last_name
                FROM employees e
                WHERE e.department_id = d.department_id
                ORDER BY e.last_name
            ) employees
  FROM departments d
  WHERE department_name LIKE 'A%'
  ORDER BY department_name;
BEGIN
  OPEN c1;
  LOOP -- Process each row of query result set
    FETCH c1 INTO dept_name, emp_cur;
    -- Process 'emp_cur' here
  END LOOP;
END;
EXIT WHEN c1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE('Department: ' || dept_name);

LOOP -- Process each row of subquery result set
    FETCH emp_cur INTO emp_name;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('-- Employee: ' || emp_name);
    END LOOP;
    END LOOP;
    CLOSE c1;
END;
/

Result:
Department: Accounting
-- Employee: Gietz
-- Employee: Higgins
Department: Administration
-- Employee: Whalen

7.6 Transaction Processing and Control

Transaction processing is an Oracle Database feature that lets multiple users work on the database concurrently, and ensures that each user sees a consistent version of data and that all changes are applied in the right order.

A transaction is a sequence of one or more SQL statements that Oracle Database treats as a unit: either all of the statements are performed, or none of them are.

Different users can write to the same data structures without harming each other's data or coordinating with each other, because Oracle Database locks data structures automatically. To maximize data availability, Oracle Database locks the minimum amount of data for the minimum amount of time.

You rarely must write extra code to prevent problems with multiple users accessing data concurrently. However, if you do need this level of control, you can manually override the Oracle Database default locking mechanisms.

Topics

- COMMIT Statement
- ROLLBACK Statement
- SAVEPOINT Statement
- Implicit Rollbacks
- SET TRANSACTION Statement
- Overriding Default Locking
7.6.1 COMMIT Statement

The `COMMIT` statement ends the current transaction, making its changes permanent and visible to other users.

**Note:**
A transaction can span multiple blocks, and a block can contain multiple transactions.

The `WRITE` clause of the `COMMIT` statement specifies the priority with which Oracle Database writes to the redo log the information that the commit operation generates.

**Note:**
The default PL/SQL commit behavior for nondistributed transactions is `BATCH NOWAIT` if the `COMMIT_LOGGING` and `COMMIT_WAIT` database initialization parameters have not been set.

**See Also:**
- *Oracle Database Concepts* for more information about committing transactions
- *Oracle Database Concepts* for information about distributed transactions
- *Oracle Database SQL Language Reference* for information about the `COMMIT` statement
- *Oracle Data Guard Concepts and Administration* for information about ensuring no loss of data during a failover to a standby database
Example 7-36  COMMIT Statement with COMMENT and WRITE Clauses

In this example, a transaction transfers money from one bank account to another. It is important that the money both leaves one account and enters the other, hence the COMMIT WRITE IMMEDIATE NOWAIT statement.

```
DROP TABLE accounts;
CREATE TABLE accounts (
    account_id  NUMBER(6),
    balance     NUMBER (10,2)
);

INSERT INTO accounts (account_id, balance) VALUES (7715, 6350.00);
INSERT INTO accounts (account_id, balance) VALUES (7720, 5100.50);

CREATE OR REPLACE PROCEDURE transfer (from_acct  NUMBER,
to_acct    NUMBER,
amount     NUMBER) AUTHID CURRENT_USER AS
BEGIN
    UPDATE accounts
    SET balance = balance - amount
    WHERE account_id = from_acct;

    UPDATE accounts
    SET balance = balance + amount
    WHERE account_id = to_acct;

    COMMIT WRITE IMMEDIATE NOWAIT;
END;
/
```

Query before transfer:

```
SELECT * FROM accounts;
```

Result:

```
ACCOUNT_ID    BALANCE
---------- ----------
7715       6350
7720     5100.5
```

```
BEGIN
    transfer(7715, 7720, 250);
END;
/
```

Query after transfer:

```
SELECT * FROM accounts;
```

Result:

```
ACCOUNT_ID    BALANCE
---------- ----------
7715       5100.5
7720     5350
```
7.6.2 ROLLBACK Statement

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction.

If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because a SQL statement fails or PL/SQL raises an exception, a rollback lets you take corrective action and perhaps start over.

See Also:
Oracle Database SQL Language Reference for more information about the ROLLBACK statement

Example 7-37 ROLLBACK Statement

This example inserts information about an employee into three different tables. If an INSERT statement tries to store a duplicate employee number, PL/SQL raises the predefined exception DUP_VAL_ON_INDEX. To ensure that changes to all three tables are undone, the exception handler runs a ROLLBACK.

```
DECLARE
  emp_id        NUMBER(6);
  emp_lastname  VARCHAR2(25);
  emp_salary    NUMBER(8,2);
```

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DROP TABLE emp_sal;
CREATE TABLE emp_sal AS
  SELECT employee_id, salary
  FROM employees;

CREATE UNIQUE INDEX empsal_ix
ON emp_sal (employee_id);

DROP TABLE emp_job;
CREATE TABLE emp_job AS
  SELECT employee_id, job_id
  FROM employees;

CREATE UNIQUE INDEX empjobid_ix
ON emp_job (employee_id);

DEdeclare
  emp_id        NUMBER(6);
  emp_lastname  VARCHAR2(25);
  emp_salary    NUMBER(8,2);
```
emp_jobid      VARCHAR2(10);
BEGIN
    SELECT employee_id, last_name, salary, job_id
    INTO emp_id, emp_lastname, emp_salary, emp_jobid
    FROM employees
    WHERE employee_id = 120;

    INSERT INTO emp_name (employee_id, last_name)
    VALUES (emp_id, emp_lastname);

    INSERT INTO emp_sal (employee_id, salary)
    VALUES (emp_id, emp_salary);

    INSERT INTO emp_job (employee_id, job_id)
    VALUES (emp_id, emp_jobid);

    EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Inserts were rolled back');
    END;
/

7.6.3 SAVEPOINT Statement

The SAVEPOINT statement names and marks the current point in the processing of a transaction.

Savepoints let you roll back part of a transaction instead of the whole transaction. The number of active savepoints for each session is unlimited.

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints.

If you mark a savepoint in a recursive subprogram, new instances of the SAVEPOINT statement run at each level in the recursive descent, but you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers. Reusing a savepoint name in a transaction moves the savepoint from its old position to the current point in the transaction, which means that a rollback to the savepoint affects only the current part of the transaction.

See Also:
Oracle Database SQL Language Reference for more information about the SET TRANSACTION SQL statement.

Example 7-38  SAVEPOINT and ROLLBACK Statements

This example marks a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the employee_id column, PL/SQL raises the predefined exception DUP_VAL_ON_INDEX and the transaction rolls back to the savepoint, undoing only the INSERT statement.
DROP TABLE emp_name;
CREATE TABLE emp_name AS
    SELECT employee_id, last_name, salary
    FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
    emp_id employees.employee_id%TYPE;
    emp_lastname employees.last_name%TYPE;
    emp_salary employees.salary%TYPE;
BEGIN
    SELECT employee_id, last_name, salary
    INTO emp_id, emp_lastname, emp_salary
    FROM employees
    WHERE employee_id = 120;

    UPDATE emp_name
    SET salary = salary * 1.1
    WHERE employee_id = emp_id;

    DELETE FROM emp_name
    WHERE employee_id = 130;

    SAVEPOINT do_insert;

    INSERT INTO emp_name (employee_id, last_name, salary)
    VALUES (emp_id, emp_lastname, emp_salary);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO do_insert;
        DBMS_OUTPUT.PUT_LINE('Insert was rolled back');
END;
/

Example 7-39  Reusing SAVEPOINT with ROLLBACK

DROP TABLE emp_name;
CREATE TABLE emp_name AS
    SELECT employee_id, last_name, salary
    FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
    emp_id employees.employee_id%TYPE;
    emp_lastname employees.last_name%TYPE;
    emp_salary employees.salary%TYPE;
BEGIN
    SELECT employee_id, last_name, salary
    INTO emp_id, emp_lastname, emp_salary
    FROM employees
    WHERE employee_id = 120;

    SAVEPOINT my_savepoint;
UPDATE emp_name
SET salary = salary * 1.1
WHERE employee_id = emp_id;

DELETE FROM emp_name
WHERE employee_id = 130;

SAVEPOINT my_savepoint;

INSERT INTO emp_name (employee_id, last_name, salary)
VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
  ROLLBACK TO my_savepoint;
  DBMS_OUTPUT.PUT_LINE('Transaction rolled back.');</exception>
END;
/

7.6.4 Implicit Rollbacks

Before running an INSERT, UPDATE, DELETE, or MERGE statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint.

Usually, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

The database can also roll back single SQL statements to break deadlocks. The database signals an error to a participating transaction and rolls back the current statement in that transaction.

Before running a SQL statement, the database must parse it, that is, examine it to ensure it follows syntax rules and refers to valid schema objects. Errors detected while running a SQL statement cause a rollback, but errors detected while parsing the statement do not.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to OUT parameters, and does not do any rollback.

For information about handling exceptions, see PL/SQL Error Handling

7.6.5 SET TRANSACTION Statement

You use the SET TRANSACTION statement to begin a read-only or read-write transaction, establish an isolation level, or assign your current transaction to a specified rollback segment.

Read-only transactions are useful for running multiple queries while other users update the same tables.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read-consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction.

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can appear only once in a transaction. If you set a transaction to READ
ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

Only the SELECT, OPEN, FETCH, CLOSE, LOCK TABLE, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. Queries cannot be FOR UPDATE.

See Also:

Oracle Database SQL Language Reference for more information about the SQL statement SET TRANSACTION

Example 7-40 SET TRANSACTION Statement in Read-Only Transaction

In this example, a read-only transaction gather order totals for the day, the past week, and the past month. The totals are unaffected by other users updating the database during the transaction. The orders table is in the sample schema OE.

DECLARE
    daily_order_total    NUMBER(12,2);
    weekly_order_total   NUMBER(12,2);
    monthly_order_total  NUMBER(12,2);
BEGIN
    COMMIT; -- end previous transaction
    SET TRANSACTION READ ONLY NAME 'Calculate Order Totals';

    SELECT SUM (order_total)
    INTO daily_order_total
    FROM orders
    WHERE order_date = SYSDATE;

    SELECT SUM (order_total)
    INTO weekly_order_total
    FROM orders
    WHERE order_date = SYSDATE - 7;

    SELECT SUM (order_total)
    INTO monthly_order_total
    FROM orders
    WHERE order_date = SYSDATE - 30;

    COMMIT; -- ends read-only transaction
END;
/

7.6.6 Overriding Default Locking

By default, Oracle Database locks data structures automatically, which lets different applications write to the same data structures without harming each other’s data or coordinating with each other.

If you must have exclusive access to data during a transaction, you can override default locking with these SQL statements:

• LOCK TABLE, which explicitly locks entire tables.
SELECT with the FOR UPDATE clause (SELECT FOR UPDATE), which explicitly locks specific rows of a table.

Topics

- LOCK TABLE Statement
- SELECT FOR UPDATE and FOR UPDATE Cursors
- Simulating CURRENT OF Clause with ROWID Pseudocolumn

7.6.6.1 LOCK TABLE Statement

The LOCK TABLE statement explicitly locks one or more tables in a specified lock mode so that you can share or deny access to them.

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an exclusive lock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table.

A table lock never prevents other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row does one transaction wait for the other to complete. The LOCK TABLE statement lets you specify how long to wait for another transaction to complete.

Table locks are released when the transaction that acquired them is either committed or rolled back.

See Also:

- Oracle Database Development Guide for more information about locking tables explicitly
- Oracle Database SQL Language Reference for more information about the LOCK TABLE statement

7.6.6.2 SELECT FOR UPDATE and FOR UPDATE Cursors

The SELECT statement with the FOR UPDATE clause (SELECT FOR UPDATE statement) selects the rows of the result set and locks them. SELECT FOR UPDATE lets you base an update on the existing values in the rows, because it ensures that no other user can change those values before you update them. You can also use SELECT FOR UPDATE to lock rows that you do not want to update, as in Example 10-6.

Note:

In tables compressed with Hybrid Columnar Compression (HCC), DML statements lock compression units rather than rows. HCC, a feature of certain Oracle storage systems, is described in Oracle Database Concepts.
By default, the `SELECT FOR UPDATE` statement waits until the requested row lock is acquired. To change this behavior, use the `NOWAIT`, `WAIT`, or `SKIP LOCKED` clause of the `SELECT FOR UPDATE` statement. For information about these clauses, see *Oracle Database SQL Language Reference*.

When `SELECT FOR UPDATE` is associated with an explicit cursor, the cursor is called a `FOR UPDATE` cursor. Only a `FOR UPDATE` cursor can appear in the `CURRENT OF` clause of an `UPDATE` or `DELETE` statement. (The `CURRENT OF` clause, a PL/SQL extension to the `WHERE` clause of the SQL statements `UPDATE` and `DELETE`, restricts the statement to the current row of the cursor.)

When `SELECT FOR UPDATE` queries multiple tables, it locks only rows whose columns appear in the `FOR UPDATE` clause.

### 7.6.6.3 Simulating `CURRENT OF` Clause with ROWID Pseudocolumn

The rows of the result set are locked when you open a `FOR UPDATE` cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. After the rows are unlocked, you cannot fetch from the `FOR UPDATE` cursor, as Example 7-41 shows (the result is the same if you substitute `ROLLBACK` for `COMMIT`).

The workaround is to simulate the `CURRENT OF` clause with the `ROWID` pseudocolumn (described in *Oracle Database SQL Language Reference*). Select the rowid of each row into a `UROWID` variable and use the rowid to identify the current row during subsequent updates and deletes, as in Example 7-42. (To print the value of a `UROWID` variable, convert it to `VARCHAR2`, using the `ROWIDTOCHAR` function described in *Oracle Database SQL Language Reference*.)

---

**Note:**

When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the `ROWID` of the row changes. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

---

**Caution:**

Because no `FOR UPDATE` clause locks the fetched rows, other users might unintentionally overwrite your changes.

---

**Note:**

The extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates.

---

**Example 7-41    FETCH with FOR UPDATE Cursor After COMMIT Statement**

```sql
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;
```
DECLARE
CURSOR c1 IS
    SELECT * FROM emp
    FOR UPDATE OF salary
    ORDER BY employee_id;

emp_rec emp%ROWTYPE;
BEGIN
OPEN c1;
LOOP
    FETCH c1 INTO emp_rec;  -- fails on second iteration
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('emp_rec.employee_id = ' ||
                         TO_CHAR(emp_rec.employee_id));

    UPDATE emp
    SET salary = salary * 1.05
    WHERE employee_id = 105;

    COMMIT;  -- releases locks
END LOOP;
END;
/

Result:

emp_rec.employee_id = 100
DECLARE
*
ERROR at line 1:
ORA-01002: fetch out of sequence
ORA-06512: at line 11

Example 7-42  Simulating CURRENT OF Clause with ROWID Pseudocolumn

DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
CURSOR c1 IS
    SELECT last_name, job_id, rowid
    FROM emp;  -- no FOR UPDATE clause

my_lastname employees.last_name%TYPE;
my_jobid employees.job_id%TYPE;
my_rowid UROWID;
BEGIN
OPEN c1;
LOOP
    FETCH c1 INTO my_lastname, my_jobid, my_rowid;
    EXIT WHEN c1%NOTFOUND;

    UPDATE emp
    SET salary = salary * 1.02
    WHERE rowid = my_rowid;  -- simulates WHERE CURRENT OF c1

    COMMIT;
END LOOP;
7.7 Autonomous Transactions

An autonomous transaction is an independent transaction started by another transaction, the main transaction.

Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction.

Figure 7-1 shows how control flows from the main transaction (MT) to an autonomous routine (proc2) and back again. The autonomous routine commits two autonomous transactions (AT1 and AT2).

![Figure 7-1 Transaction Control Flow](image)

**Note:** Although an autonomous transaction is started by another transaction, it is not a nested transaction, because:

- It does not share transactional resources (such as locks) with the main transaction.
- It does not depend on the main transaction.
  For example, if the main transaction rolls back, nested transactions roll back, but autonomous transactions do not.
- Its committed changes are visible to other transactions immediately.
  A nested transaction's committed changes are not visible to other transactions until the main transaction commits.
- Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.
7.7.1 Advantages of Autonomous Transactions

After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

Autonomous transactions help you build modular, reusable software components. You can encapsulate autonomous transactions in stored subprograms. An invoking application needs not know whether operations done by that stored subprogram succeeded or failed.

7.7.2 Transaction Context

The main transaction shares its context with nested routines, but not with autonomous transactions. When one autonomous routine invokes another (or itself, recursively), the routines share no transaction context. When an autonomous routine invokes a nonautonomous routine, the routines share the same transaction context.

7.7.3 Transaction Visibility

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. These changes become visible to the main transaction when it resumes, if its isolation level is set to READ COMMitted (the default).

If you set the isolation level of the main transaction to SERIALizable, changes made by its autonomous transactions are not visible to the main transaction when it resumes:

    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Note:

- Transaction properties apply only to the transaction in which they are set.
- Cursor attributes are not affected by autonomous transactions.

7.7.4 Declaring Autonomous Routines

To declare an autonomous routine, use the AUTONOMOUS_TRANSACTION pragma.

For information about this pragma, see "AUTONOMOUS_TRANSACTION Pragma".

Tip:

For readability, put the AUTONOMOUS_TRANSACTION pragma at the top of the declarative section. (The pragma is allowed anywhere in the declarative section.)

You cannot apply the AUTONOMOUS_TRANSACTION pragma to an entire package or ADT, but you can apply it to each subprogram in a package or each method of an ADT.

Example 7-43  Declaring Autonomous Function in Package

This example marks a package function as autonomous.

```sql
CREATE OR REPLACE PACKAGE emp_actions AUTHID DEFINER AS  -- package specification
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER) RETURN NUMBER;
END emp_actions;
/
CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- package body
  -- code for function raise_salary
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER) RETURN NUMBER IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    new_sal NUMBER(8,2);
    BEGIN
      UPDATE employees SET salary =
          salary + sal_raise WHERE employee_id = emp_id;
      COMMIT;
      SELECT salary INTO new_sal FROM employees
          WHERE employee_id = emp_id;
      RETURN new_sal;
      END raise_salary;
END emp_actions;
/
```

Example 7-44  Declaring Autonomous Standalone Procedure

This example marks a standalone subprogram as autonomous.

```sql
CREATE OR REPLACE PROCEDURE lower_salary
  (emp_id NUMBER, amount NUMBER)
  ``
Example 7-45    Declaring Autonomous PL/SQL Block

This example marks a schema-level PL/SQL block as autonomous. (A nested PL/SQL block cannot be autonomous.)

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
    emp_id NUMBER(6)   := 200;
    amount NUMBER(6,2) := 200;
BEGIN
    UPDATE employees
    SET salary = salary - amount
    WHERE employee_id = emp_id;

    COMMIT;
END;
/```

7.7.5 Controlling Autonomous Transactions

The first SQL statement in an autonomous routine begins a transaction. When one transaction ends, the next SQL statement begins another transaction. All SQL statements run since the last commit or rollback comprise the current transaction. To control autonomous transactions, use these statements, which apply only to the current (active) transaction:

- COMMIT
- ROLLBACK [TO savepoint_name]
- SAVEPOINT savepoint_name
- SET TRANSACTION

Topics

- Entering and Exiting Autonomous Routines
- Committing and Rolling Back Autonomous Transactions
- Savepoints
- Avoiding Errors with Autonomous Transactions
7.7.5.1 Entering and Exiting Autonomous Routines

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes.

If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception is unhandled, or if the transaction ends because of some other unhandled exception, then the transaction rolls back.

To exit normally, the routine must explicitly commit or roll back all autonomous transactions. If the routine (or any routine invoked by it) has pending transactions, then PL/SQL raises an exception and the pending transactions roll back.

7.7.5.2 Committing and Rolling Back Autonomous Transactions

COMMIT and ROLLBACK end the active autonomous transaction but do not exit the autonomous routine. When one transaction ends, the next SQL statement begins another transaction. A single autonomous routine can contain several autonomous transactions, if it issues several COMMIT statements.

7.7.5.3 Savepoints

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. In an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does not roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

7.7.5.4 Avoiding Errors with Autonomous Transactions

To avoid some common errors, remember:

- If an autonomous transaction tries to access a resource held by the main transaction, a deadlock can occur. The database raises an exception in the autonomous transaction, which rolls back if the exception is unhandled.

- The database initialization parameter TRANSACTIONS specifies the maximum number of concurrent transactions. That number might be exceeded because an autonomous transaction runs concurrently with the main transaction.

- If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception is unhandled, the transaction rolls back.

- You cannot run a PIPE ROW statement in an autonomous routine while an autonomous transaction is open. You must close the autonomous transaction before running the PIPE ROW statement. This is normally accomplished by
committing or rolling back the autonomous transaction before running the `PIPE ROW` statement.

7.7.6 Autonomous Triggers

A trigger must be autonomous to run TCL or DDL statements.

To run DDL statements, the trigger must use native dynamic SQL.

---

**See Also:**

- [PL/SQL Triggers](#), for general information about triggers
- "[Description of Static SQL](#)" for general information about TCL statements
- [Oracle Database SQL Language Reference](#) for information about DDL statements
- "[Native Dynamic SQL](#)" for information about native dynamic SQL

---

One use of triggers is to log events transparently—for example, to log all inserts into a table, even those that roll back.

**Example 7-46   Autonomous Trigger Logs INSERT Statements**

In this example, whenever a row is inserted into the `EMPLOYEES` table, a trigger inserts the same row into a log table. Because the trigger is autonomous, it can commit changes to the log table regardless of whether they are committed to the main table.

```sql
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

-- Log table:
DROP TABLE log;
CREATE TABLE log (
    log_id NUMBER(6),
    up_date DATE,
    new_sal NUMBER(8,2),
    old_sal NUMBER(8,2)
);

-- Autonomous trigger on emp table:
CREATE OR REPLACE TRIGGER log_sal
    BEFORE UPDATE OF salary ON emp FOR EACH ROW
        DECLARE
            PRAGMA AUTONOMOUS_TRANSACTION;
        BEGIN
            INSERT INTO log (
                log_id,
                up_date,
                new_sal,
                old_sal
            )
            VALUES (:old.employee_id,
```
Example 7-47  Autonomous Trigger Uses Native Dynamic SQL for DDL

In this example, an autonomous trigger uses native dynamic SQL (an EXECUTE IMMEDIATE statement) to drop a temporary table after a row is inserted into the table log.

```
DROP TABLE temp;
CREATE TABLE temp (    temp_id NUMBER(6),    up_date DATE );

CREATE OR REPLACE TRIGGER drop_temp_table
AFTER INSERT ON log
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE temp';
    COMMIT;
END;
/
```

-- Show how trigger works
SELECT * FROM temp;

Result:

```
LOG_ID UP_DATE   NEW_SAL OLD_SAL
------- ------- --------- ---------
   115 02-OCT-12   3255       3100
   116 02-OCT-12   3045       2900
```

2 rows selected.
no rows selected

INSERT INTO log (log_id, up_date, new_sal, old_sal) VALUES (999, SYSDATE, 5000, 4500);

1 row created.

SELECT * FROM temp;

Result:

SELECT * FROM temp
  *
ERROR at line 1:
ORA-00942: table or view does not exist

7.7.7 Invoking Autonomous Functions from SQL

A function invoked from SQL statements must obey rules meant to control side effects.

By definition, an autonomous routine never reads or writes database state (that is, it neither queries nor modifies any database table).

See Also:

"Subprogram Side Effects" for more information

Example 7-48 Invoking Autonomous Function

The package function log_msg is autonomous. Therefore, when the query invokes the function, the function inserts a message into database table debug_output without violating the rule against writing database state (modifying database tables).

DROP TABLE debug_output;
CREATE TABLE debug_output (message VARCHAR2(200));

CREATE OR REPLACE PACKAGE debugging AUTHID DEFINER AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
END debugging;
/
CREATE OR REPLACE PACKAGE BODY debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
      INSERT INTO debug_output (message) VALUES (msg);
      COMMIT;
      RETURN msg;
    END;
END debugging;
/
-- Invoke package function from query
DECLARE
  my_emp_id    NUMBER(6);
  my_last_name VARCHAR2(25);
  my_count     NUMBER;
BEGIN
  my_emp_id = 12345;
  my_last_name = 'Smith';
  my_count = 10;
  log_msg ('Hello');
  SELECT * FROM debug_output;
END;
my_emp_id := 120;

SELECT debugging.log_msg(last_name)
INTO my_last_name
FROM employees
WHERE employee_id = my_emp_id;

/* Even if you roll back in this scope, 
the insert into 'debug_output' remains committed, 
because it is part of an autonomous transaction. */

ROLLBACK;
END;
/
Dynamic SQL is a programming methodology for generating and running SQL statements at run time.

It is useful when writing general-purpose and flexible programs like ad hoc query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compile time the full text of a SQL statement or the number or data types of its input and output variables.

PL/SQL provides two ways to write dynamic SQL:

- Native dynamic SQL, a PL/SQL language (that is, native) feature for building and running dynamic SQL statements
- DBMS_SQL package, an API for building, running, and describing dynamic SQL statements

Native dynamic SQL code is easier to read and write than equivalent code that uses the DBMS_SQL package, and runs noticeably faster (especially when it can be optimized by the compiler). However, to write native dynamic SQL code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement. If you do not know this information at compile time, you must use the DBMS_SQL package. You must also use the DBMS_SQL package if you want a stored subprogram to return a query result implicitly (not through an OUT REF CURSOR parameter).

When you need both the DBMS_SQL package and native dynamic SQL, you can switch between them, using the "DBMS_SQL.TO_REFCURSOR Function" and "DBMS_SQL.TO_CURSOR_NUMBER Function".

Topics

- When You Need Dynamic SQL
- Native Dynamic SQL
- DBMS_SQL Package
- SQL Injection

8.1 When You Need Dynamic SQL

In PL/SQL, you need dynamic SQL to run:

- SQL whose text is unknown at compile time
  For example, a SELECT statement that includes an identifier that is unknown at compile time (such as a table name) or a WHERE clause in which the number of subclauses is unknown at compile time.
- SQL that is not supported as static SQL
  That is, any SQL construct not included in "Description of Static SQL".
If you do not need dynamic SQL, use static SQL, which has these advantages:

- Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects.
- Successful compilation creates schema object dependencies.
  For information about schema object dependencies, see Oracle Database Development Guide.

For information about using static SQL statements with PL/SQL, see PL/SQL Static SQL.

8.2 Native Dynamic SQL

Native dynamic SQL processes most dynamic SQL statements with the EXECUTE IMMEDIATE statement.

If the dynamic SQL statement is a SELECT statement that returns multiple rows, native dynamic SQL gives you these choices:

- Use the EXECUTE IMMEDIATE statement with the BULK COLLECT INTO clause.
- Use the OPEN FOR, FETCH, and CLOSE statements.

The SQL cursor attributes work the same way after native dynamic SQL INSERT, UPDATE, DELETE, MERGE, and single-row SELECT statements as they do for their static SQL counterparts. For more information about SQL cursor attributes, see “Cursors Overview”.

Topics

- EXECUTE IMMEDIATE Statement
- OPEN FOR, FETCH, and CLOSE Statements
- Repeated Placeholder Names in Dynamic SQL Statements

8.2.1 EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement is the means by which native dynamic SQL processes most dynamic SQL statements.

If the dynamic SQL statement is self-contained (that is, if it has no placeholders for bind variables and the only result that it can possibly return is an error), then the EXECUTE IMMEDIATE statement needs no clauses.

If the dynamic SQL statement includes placeholders for bind variables, each placeholder must have a corresponding bind variable in the appropriate clause of the EXECUTE IMMEDIATE statement, as follows:

- If the dynamic SQL statement is a SELECT statement that can return at most one row, put out-bind variables (defines) in the INTO clause and in-bind variables in the USING clause.
- If the dynamic SQL statement is a SELECT statement that can return multiple rows, put out-bind variables (defines) in the BULK COLLECT INTO clause and in-bind variables in the USING clause.
• If the dynamic SQL statement is a DML statement without a RETURNING INTO clause, other than SELECT, put all bind variables in the USING clause.

• If the dynamic SQL statement is a DML statement with a RETURNING INTO clause, put in-bind variables in the USING clause and out-bind variables in the RETURNING INTO clause.

• If the dynamic SQL statement is an anonymous PL/SQL block or a CALL statement, put all bind variables in the USING clause.

If the dynamic SQL statement invokes a subprogram, ensure that:

– The subprogram is either created at schema level or declared and defined in a package specification.

– Every bind variable that corresponds to a placeholder for a subprogram parameter has the same parameter mode as that subprogram parameter and a data type that is compatible with that of the subprogram parameter.

– No bind variable is the reserved word NULL.

To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 8-7.

– No bind variable has a data type that SQL does not support (such as associative array indexed by string).

If the data type is a collection or record type, then it must be declared in a package specification.

---

**Note:**

Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.

---

In Example 8-4, Example 8-5, and Example 8-6, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of a PL/SQL collection type. Collection types are not SQL data types. In each example, the collection type is declared in a package specification, and the subprogram is declared in the package specification and defined in the package body.
See Also:

- "CREATE FUNCTION Statement" for information about creating functions at schema level
- "CREATE PROCEDURE Statement" for information about creating procedures at schema level
- "PL/SQL Packages" for information about packages
- "CREATE PACKAGE Statement" for information about declaring subprograms in packages
- "CREATE PACKAGE BODY Statement" for information about declaring and defining subprograms in packages
- "CREATE PACKAGE Statement" for more information about declaring types in a package specification
- "EXECUTE IMMEDIATE Statement" for syntax details of the `EXECUTE IMMEDIATE` statement
- "PL/SQL Collections and Records" for information about collection types

Example 8-1  Invoking Subprogram from Dynamic PL/SQL Block

In this example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram created at schema level.

```plsql
-- Subprogram that dynamic PL/SQL block invokes:
CREATE OR REPLACE PROCEDURE create_dept (  
  deptid IN OUT NUMBER,
  dname IN VARCHAR2,
  mgrid IN NUMBER,
  locid IN NUMBER
) AUTHID DEFINER AS
BEGIN
  deptid := departments_seq.NEXTVAL;

  INSERT INTO departments (  
    department_id,
    department_name,
    manager_id,
    location_id
  )
  VALUES (deptid, dname, mgrid, locid);
END;
/

DECLARE
  plsql_block VARCHAR2(500);
  new_deptid  NUMBER(4);
  new_dname   VARCHAR2(30) := 'Advertising';
  new_mgrid   NUMBER(6)    := 200;
  new_locid   NUMBER(4)    := 1700;
BEGIN
  -- Dynamic PL/SQL block invokes subprogram:
  plsql_block := 'BEGIN create_dept(:a, :b, :c, :d); END;';

  /* Specify bind variables in USING clause.
  
  DECLARE
    bind_name VARCHAR2(30) := 'Advertising';
    bind_value NUMBER(6) := 200;
    bind_locid NUMBER(4) := 1700;
    bind_deptid NUMBER(4);
  BEGIN
    -- Execute dynamic PL/SQL block
    EXECUTE IMMEDIATE plsql_block USING bind_name, bind_value, bind_locid, bind_deptid;
  END;*/
```

/* Specify bind variables in USING clause.
Specify mode for first parameter.
Modes of other parameters are correct by default. */

EXECUTE IMMEDIATE plsql_block
    USING IN OUT new_deptid, new_dname, new_mgrid, new_locid;
END;
/

Example 8-2  Dynamically Invoking Subprogram with BOOLEAN Formal Parameter

In this example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL (but not SQL) data type BOOLEAN.

CREATE OR REPLACE PROCEDURE p (x BOOLEAN) AUTHID DEFINER AS
BEGIN
    IF x THEN
        DBMS_OUTPUT.PUT_LINE('x is true');
    END IF;
END;
/

DECLARE
    dyn_stmt VARCHAR2(200);
    b BOOLEAN := TRUE;
BEGIN
    dyn_stmt := 'BEGIN
                  p(:x)
                END;';
    EXECUTE IMMEDIATE dyn_stmt USING b;
END;
/

Result:

x is true

Example 8-3  Dynamically Invoking Subprogram with RECORD Formal Parameter

In this example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL (but not SQL) data type RECORD. The record type is declared in a package specification, and the subprogram is declared in the package specification and defined in the package body.

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

    TYPE rec IS RECORD (n1 NUMBER, n2 NUMBER);

    PROCEDURE p (x OUT rec, y NUMBER, z NUMBER);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS

    PROCEDURE p (x OUT rec, y NUMBER, z NUMBER) AS
    BEGIN
        x.n1 := y;
        x.n2 := z;
    END p;
END pkg;
/
DECLARE

Chapter 8
Native Dynamic SQL

8-5
Example 8-4  Dynamically Invoking Subprogram with Assoc. Array Formal Parameter

In this example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type associative array indexed by PLS_INTEGER.

---

Note:
An associative array type used in this context must be indexed by PLS_INTEGER.
Example 8-5  Dynamically Invoking Subprogram with Nested Table Formal Parameter

In this example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type nested table.

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

    TYPE names IS TABLE OF VARCHAR2(10);

    PROCEDURE print_names (x names);
END pkg;
/

CREATE OR REPLACE PACKAGE BODY pkg AS

    PROCEDURE print_names (x names) IS
    BEGIN
        FOR i IN x.FIRST .. x.LAST LOOP
            DBMS_OUTPUT.PUT_LINE(x(i));
        END LOOP;
    END;
END pkg;
/

DECLARE
    fruits   pkg.names;
    dyn_stmt VARCHAR2(3000);
BEGIN
    fruits := pkg.names('apple', 'banana', 'cherry');
    dyn_stmt := 'BEGIN
                  pkg.print_names(:x);
                  END;';
    EXECUTE IMMEDIATE dyn_stmt USING fruits;
END;
/

Example 8-6  Dynamically Invoking Subprogram with Varray Formal Parameter

In this example, the dynamic PL/SQL block is an anonymous PL/SQL block that invokes a subprogram that has a formal parameter of the PL/SQL collection type varray.

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS

    TYPE foursome IS VARRAY(4) OF VARCHAR2(5);

    PROCEDURE print_foursome (x foursome);
END pkg;
/

CREATE OR REPLACE PACKAGE BODY pkg AS

    PROCEDURE print_foursome (x foursome) IS
    BEGIN
        IF x.COUNT = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Empty');
        ELSE
            FOR i IN x.FIRST .. x.LAST LOOP
                DBMS_OUTPUT.PUT_LINE(x(i));
            END LOOP;
        END IF;
    END;
END pkg;
/

DECLARE
    fruits   pkg.names;
    dyn_stmt VARCHAR2(3000);
BEGIN
    fruits := pkg.names('apple', 'banana', 'cherry');
    dyn_stmt := 'BEGIN
                  pkg.print_foursome(:x);
                  END;';
    EXECUTE IMMEDIATE dyn_stmt USING fruits;
END;
DECLARE
  directions pkg.foursome;
  dyn_stmt VARCHAR2(3000);
BEGIN
  directions := pkg.foursome('north', 'south', 'east', 'west');

  dyn_stmt := 'BEGIN pkg.print_foursome(:x); END;';
  EXECUTE IMMEDIATE dyn_stmt USING directions;
END;
/

Example 8-7  Uninitialized Variable Represents NULL in USING Clause
This example uses an uninitialized variable to represent the reserved word NULL in the USING clause.

CREATE TABLE employees_temp AS SELECT * FROM EMPLOYEES;

DECLARE
  a_null  CHAR(1);  -- Set to NULL automatically at run time
BEGIN
  EXECUTE IMMEDIATE 'UPDATE employees_temp SET commission_pct = :x'
    USING a_null;
END;
/

8.2.2 OPEN FOR, FETCH, and CLOSE Statements

If the dynamic SQL statement represents a SELECT statement that returns multiple rows, you can process it with native dynamic SQL as follows:

1. Use an OPEN FOR statement to associate a cursor variable with the dynamic SQL statement. In the USING clause of the OPEN FOR statement, specify a bind variable for each placeholder in the dynamic SQL statement.

   The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 8-7.

2. Use the FETCH statement to retrieve result set rows one at a time, several at a time, or all at once.

3. Use the CLOSE statement to close the cursor variable.

The dynamic SQL statement can query a collection if the collection meets the criteria in "Querying a Collection".

See Also:

- "OPEN FOR Statement" for syntax details
- "FETCH Statement" for syntax details
- "CLOSE Statement" for syntax details
Example 8-8  Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements

This example lists all employees who are managers, retrieving result set rows one at a time.

DECLARE
  TYPE EmpCurTyp  IS REF CURSOR;
  v_emp_cursor    EmpCurTyp;
  emp_record      employees%ROWTYPE;
  v_stmt_str      VARCHAR2(200);
  v_e_job         employees.job%TYPE;
BEGIN
  -- Dynamic SQL statement with placeholder:
  v_stmt_str := 'SELECT * FROM employees WHERE job_id = :j';

  -- Open cursor & specify bind variable in USING clause:
  OPEN v_emp_cursor FOR v_stmt_str USING 'MANAGER';

  -- Fetch rows from result set one at a time:
  LOOP
    FETCH v_emp_cursor INTO emp_record;
    EXIT WHEN v_emp_cursor%NOTFOUND;
  END LOOP;

  -- Close cursor:
  CLOSE v_emp_cursor;
END;
/

Example 8-9  Querying a Collection with Native Dynamic SQL

This example is like Example 7-30 except that the collection variable v1 is a bind variable.

CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER AS
  TYPE rec IS RECORD(f1 NUMBER, f2 VARCHAR2(30));
  TYPE mytab IS TABLE OF rec INDEX BY pls_integer;
END;
/

DECLARE
  v1 pkg.mytab;  -- collection of records
  v2 pkg.rec;
  c1 SYS_REFCURSOR;
BEGIN
  OPEN c1 FOR 'SELECT * FROM TABLE(:1)' USING v1;
  FETCH c1 INTO v2;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE('Values in record are ' || v2.f1 || ' and ' || v2.f2);
END;
/
8.2.3 Repeated Placeholder Names in Dynamic SQL Statements

If you repeat placeholder names in dynamic SQL statements, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement.

Topics
- Dynamic SQL Statement is Not Anonymous Block or CALL Statement
- Dynamic SQL Statement is Anonymous Block or CALL Statement

8.2.3.1 Dynamic SQL Statement is Not Anonymous Block or CALL Statement

If the dynamic SQL statement does not represent an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is insignificant.

Placeholders are associated with bind variables in the USING clause by position, not by name.

For example, in this dynamic SQL statement, the repetition of the name :x is insignificant:

```sql
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

In the corresponding USING clause, you must supply four bind variables. They can be different; for example:

```sql
EXECUTE IMMEDIATE sql_stmt USING a, b, c, d;
```

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

```
INSERT INTO payroll VALUES (a, b, c, d)
```

To associate the same bind variable with each occurrence of :x, you must repeat that bind variable; for example:

```sql
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

```
INSERT INTO payroll VALUES (a, a, b, a)
```

8.2.3.2 Dynamic SQL Statement is Anonymous Block or CALL Statement

If the dynamic SQL statement represents an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is significant.

Each unique placeholder name must have a corresponding bind variable in the USING clause. If you repeat a placeholder name, you need not repeat its corresponding bind variable. All references to that placeholder name correspond to one bind variable in the USING clause.

Example 8-10 Repeated Placeholder Names in Dynamic PL/SQL Block

In this example, all references to the first unique placeholder name, :x, are associated with the first bind variable in the USING clause, a, and the second unique placeholder name, :y, is associated with the second bind variable in the USING clause, b.
CREATE PROCEDURE calc_stats (  
w NUMBER,  
x NUMBER,  
y NUMBER,  
z NUMBER )  
IS  
BEGIN  
  DBMS_OUTPUT.PUT_LINE(w + x + y + z);  
END;  
/  
DECLARE  
a NUMBER := 4;  
b NUMBER := 7;  
plsql_block VARCHAR2(100);  
BEGIN  
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';  
  EXECUTE IMMEDIATE plsql_block USING a, b;  
  -- calc_stats(a, a, b, a)  
END;  
/  

8.3 DBMS_SQL Package

The DBMS_SQL package defines an entity called a SQL cursor number. Because the SQL cursor number is a PL/SQL integer, you can pass it across call boundaries and store it.

You must use the DBMS_SQL package to run a dynamic SQL statement if any of the following are true:

- You do not know the SELECT list until run time.
- You do not know until run time what placeholders in a SELECT or DML statement must be bound.
- You want a stored subprogram to return a query result implicitly (not through an OUT REF CURSOR parameter), which requires the DBMS_SQL.RETURN_RESULT procedure.

In these situations, you must use native dynamic SQL instead of the DBMS_SQL package:

- The dynamic SQL statement retrieves rows into records.
- You want to use the SQL cursor attribute %FOUND, %ISOPEN, %NOTFOUND, or %ROWCOUNT after issuing a dynamic SQL statement that is an INSERT, UPDATE, DELETE, MERGE, or single-row SELECT statement.

When you need both the DBMS_SQL package and native dynamic SQL, you can switch between them, using the functions DBMS_SQL.TO_REFCURSOR and DBMS_SQL.TO_CURSOR_NUMBER.

Topics

- DBMS_SQL.RETURN_RESULT Procedure
- DBMS_SQL.GET_NEXT_RESULT Procedure
- DBMS_SQL.TO_REFCURSOR Function
- DBMS_SQL.TO_CURSOR_NUMBER Function
You can invoke DBMS_SQL subprograms remotely.

See Also:

• "Native Dynamic SQL" for information about native dynamic SQL
• Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_SQL package, including instructions for running a dynamic SQL statement that has an unknown number of input or output variables ("Method 4")

8.3.1 DBMS_SQL.RETURN_RESULT Procedure

The DBMS_SQL.RETURN_RESULT procedure lets a stored subprogram return a query result implicitly to either the client program (which invokes the subprogram indirectly) or the immediate caller of the subprogram. After DBMS_SQL.RETURN_RESULT returns the result, only the recipient can access it.

The DBMS_SQL.RETURN_RESULT has two overloads:

PROCEDURE RETURN_RESULT (rc IN OUT SYS_REFCURSOR,
                           to_client IN BOOLEAN DEFAULT TRUE);

PROCEDURE RETURN_RESULT (rc IN OUT INTEGER,
                           to_client IN BOOLEAN DEFAULT TRUE);

The rc parameter is either an open cursor variable (SYS_REFCURSOR) or the cursor number (INTEGER) of an open cursor. To open a cursor and get its cursor number, invoke the DBMS_SQL.OPEN_CURSOR function, described in Oracle Database PL/SQL Packages and Types Reference.

When the to_client parameter is TRUE (the default), the DBMS_SQL.RETURN_RESULT procedure returns the query result to the client program (which invokes the subprogram indirectly); when this parameter is FALSE, the procedure returns the query result to the subprogram's immediate caller.

See Also:

• Oracle Database PL/SQL Packages and Types Reference for more information about DBMS_SQL.RETURN_RESULT
• Oracle Call Interface Programmer's Guide for information about C and .NET support for implicit query results
• SQL*Plus User's Guide and Reference for information about SQL*Plus support for implicit query results
Example 8-11  DBMS_SQL.RETURN_RESULT Procedure

In this example, the procedure p invokes DBMS_SQL.RETURN_RESULT without the optional to_client parameter (which is TRUE by default). Therefore, DBMS_SQL.RETURN_RESULT returns the query result to the subprogram client (the anonymous block that invokes p). After p returns a result to the anonymous block, only the anonymous block can access that result.

CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  c1 SYS_REFCURSOR;
  c2 SYS_REFCURSOR;
BEGIN
  OPEN c1 FOR
    SELECT first_name, last_name
    FROM employees
    WHERE employee_id = 176;
  DBMS_SQL.RETURN_RESULT (c1);
  -- Now p cannot access the result.
  OPEN c2 FOR
    SELECT city, state_province
    FROM locations
    WHERE country_id = 'AU';
  DBMS_SQL.RETURN_RESULT (c2);
  -- Now p cannot access the result.
END;
/
BEGIN
  p;
END;
/

Result:

ResultSet #1
FIRST_NAME           LAST_NAME
-------------------- -------------------------
Jonathon             Taylor

ResultSet #2
CITY                           STATE_PROVINCE
------------------------------ -------------------------
Sydney                         New South Wales

8.3.2 DBMS_SQL.GET_NEXT_RESULT Procedure

The DBMS_SQL.GET_NEXT_RESULT procedure gets the next result that the DBMS_SQL.RETURN_RESULT procedure returned to the recipient. The two procedures return results in the same order.

The DBMS_SQL.GET_NEXT_RESULT has two overloads:

PROCEDURE GET_NEXT_RESULT (c IN INTEGER, rc OUT SYS_REFCURSOR);
PROCEDURE GET_NEXT_RESULT (c IN INTEGER, rc OUT INTEGER);
The \texttt{c} parameter is the cursor number of an open cursor that directly or indirectly invokes a subprogram that uses the \texttt{DBMS_SQL.RETURN_RESULT} procedure to return a query result implicitly.

To open a cursor and get its cursor number, invoke the \texttt{DBMS_SQL.OPEN_CURSOR} function. \texttt{DBMS_SQL.OPEN_CURSOR} has an optional parameter, \texttt{treat_as_client_for_results}. When this parameter is \texttt{FALSE} (the default), the caller that opens this cursor (to invoke a subprogram) is not treated as the client that receives query results for the client from the subprogram that uses \texttt{DBMS_SQL.RETURN_RESULT}—those query results are returned to the client in a upper tier instead. When this parameter is \texttt{TRUE}, the caller is treated as the client. For more information about the \texttt{DBMS_SQL.OPEN_CURSOR} function, see \textit{Oracle Database PL/SQL Packages and Types Reference}.

The \texttt{rc} parameter is either a cursor variable (\texttt{SYS_REFCURSOR}) or the cursor number (\texttt{INTEGER}) of an open cursor.

In Example 8-12, the procedure \texttt{get_employee_info} uses \texttt{DBMS_SQL.RETURN_RESULT} to return two query results to a client program and is invoked dynamically by the anonymous block \texttt{<<main>>}. Because \texttt{<<main>>} needs to receive the two query results that \texttt{get_employee_info} returns, \texttt{<<main>>} opens a cursor to invoke \texttt{get_employee_info} using \texttt{DBMS_SQL.OPEN_CURSOR} with the parameter \texttt{treat_as_client_for_results} set to \texttt{TRUE}. Therefore, \texttt{DBMS_SQL.GET_NEXT_RESULT} returns its results to \texttt{<<main>>}, which uses the cursor \texttt{rc} to fetch them.

**Example 8-12 \ DBMS_SQL.GET_NEXT_RESULT Procedure**

```sql
CREATE OR REPLACE PROCEDURE get_employee_info (id IN VARCHAR2) AUTHID DEFINER AS
  rc SYS_REFCURSOR;
BEGIN
  -- Return employee info
  OPEN rc FOR SELECT first_name, last_name, email, phone_number
    FROM employees
    WHERE employee_id = id;
  DBMS_SQL.RETURN_RESULT(rc);

  -- Return employee job history
  OPEN rc FOR SELECT job_title, start_date, end_date
    FROM job_history jh, jobs j
    WHERE jh.employee_id = id AND
          jh.job_id = j.job_id
    ORDER BY start_date DESC;
  DBMS_SQL.RETURN_RESULT(rc);
END;
/

<<main>>
DECLARE
  c INTEGER;
  rc SYS_REFCURSOR;
  n NUMBER;
  first_name VARCHAR2(20);
  last_name VARCHAR2(25);
  email VARCHAR2(25);
  phone_number VARCHAR2(20);
  job_title VARCHAR2(35);
```

Chapter 8

DBMS_SQL Package

8-14
start_date DATE;
end_date DATE;

BEGIN
    c := DBMS_SQL.OPEN_CURSOR(true);
    DBMS_SQL.PARSE(c, 'BEGIN get_employee_info(:id); END;', DBMS_SQL.NATIVE);
    DBMS_SQL.BIND_VARIABLE(c, ':id', 176);
    n := DBMS_SQL.EXECUTE(c);

    -- Get employee info
    dbms_sql.get_next_result(c, rc);
    FETCH rc INTO first_name, last_name, email, phone_number;
    DBMS_OUTPUT.PUT_LINE('Employee: '||first_name || ' ' || last_name);
    DBMS_OUTPUT.PUT_LINE('Email: ' ||email);
    DBMS_OUTPUT.PUT_LINE('Phone: ' ||phone_number);

    -- Get employee job history
    DBMS_OUTPUT.PUT_LINE('Titles:');
    DBMS_SQL.GET_NEXT_RESULT(c, rc);
    LOOP
        FETCH rc INTO job_title, start_date, end_date;
        EXIT WHEN rc%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('- '||job_title||' ('||start_date||' - ' ||end_date||')');
    END LOOP;

    DBMS_SQL.CLOSE_CURSOR(c);
END main;
/

Result:
Employee: Jonathon Taylor
Email: JTAYLOR
Phone: 011.44.1644.429265
Titles:
- Sales Manager (01-JAN-07 - 31-DEC-07)
- Sales Representative (24-MAR-06 - 31-DEC-06)

PL/SQL procedure successfully completed.

8.3.3 DBMS_SQL.TO_REFCURSOR Function

The DBMS_SQL.TO_REFCURSOR function converts a SQL cursor number to a weak cursor variable, which you can use in native dynamic SQL statements.

Before passing a SQL cursor number to the DBMS_SQL.TO_REFCURSOR function, you must OPEN, PARSE, and EXECUTE it (otherwise an error occurs).

After you convert a SQL cursor number to a REF CURSOR variable, DBMS_SQL operations can access it only as the REF CURSOR variable, not as the SQL cursor number. For example, using the DBMS_SQL.IS_OPEN function to see if a converted SQL cursor number is still open causes an error.

Example 8-13 uses the DBMS_SQL.TO_REFCURSOR function to switch from the DBMS_SQL package to native dynamic SQL.
Example 8-13  Switching from DBMS_SQL Package to Native Dynamic SQL

CREATE OR REPLACE TYPE vc_array IS TABLE OF VARCHAR2(200);
/
CREATE OR REPLACE TYPE numlist IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE do_query_1 (
  placeholder vc_array,
  bindvars vc_array,
  sql_stmt VARCHAR2
) AUTHID DEFINER
IS
  TYPE curtype IS REF CURSOR;
  src_cur     curtype;
  curid       NUMBER;
  bindnames   vc_array;
  empnos      numlist;
  depts       numlist;
  ret         NUMBER;
  isopen      BOOLEAN;
BEGIN
  -- Open SQL cursor number:
  curid := DBMS_SQL.OPEN_CURSOR;

  -- Parse SQL cursor number:
  DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);

  bindnames := placeholder;

  -- Bind variables:
  FOR i IN 1 .. bindnames.COUNT LOOP
    DBMS_SQL.BIND_VARIABLE(curid, bindnames(i), bindvars(i));
  END LOOP;

  -- Run SQL cursor number:
  ret := DBMS_SQL.EXECUTE(curid);

  -- Switch from DBMS_SQL to native dynamic SQL:
  src_cur := DBMS_SQL.TO_REFCURSOR(curid);
  FETCH src_cur BULK COLLECT INTO empnos, depts;

  -- This would cause an error because curid was converted to a REF CURSOR:
  -- isopen := DBMS_SQL.IS_OPEN(curid);

  CLOSE src_cur;
END;
/

8.3.4 DBMS_SQL.TO_CURSOR_NUMBER Function

The `DBMS_SQL.TO_CURSOR_NUMBER` function converts a `REF CURSOR` variable (either strong or weak) to a SQL cursor number, which you can pass to `DBMS_SQL` subprograms.

Before passing a `REF CURSOR` variable to the `DBMS_SQL.TO_CURSOR_NUMBER` function, you must OPEN it.

After you convert a `REF CURSOR` variable to a SQL cursor number, native dynamic SQL operations cannot access it.
Example 8-14 uses the DBMS_SQL.TO_CURSOR_NUMBER function to switch from native dynamic SQL to the DBMS_SQL package.

Example 8-14 Switching from Native Dynamic SQL to DBMS_SQL Package

```
CREATE OR REPLACE PROCEDURE do_query_2 (    
  sql_stmt VARCHAR2  
) AUTHID DEFINER 
IS 
  TYPE curtype IS REF CURSOR;
  src_cur   curtype;
  curid     NUMBER;
  desctab   DBMS_SQL.DESC_TAB;
  colcnt    NUMBER;
  namevar   VARCHAR2(50);
  numvar    NUMBER;
  datevar   DATE;
  empno     NUMBER := 100;
BEGIN 
  -- sql_stmt := SELECT ... FROM employees WHERE employee_id = :b1';

  -- Open REF CURSOR variable:
  OPEN src_cur FOR sql_stmt USING empno;

  -- Switch from native dynamic SQL to DBMS_SQL package:
  curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
  DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);

  -- Define columns:
  FOR i IN 1 .. colcnt LOOP
    IF desctab(i).col_type = 2 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
    ELSIF desctab(i).col_type = 12 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
      -- statements
    ELSE
      DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 50);
    END IF;
  END LOOP;

  -- Fetch rows with DBMS_SQL package:
  WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
    FOR i IN 1 .. colcnt LOOP
      IF (desctab(i).col_type = 1) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
      ELSIF (desctab(i).col_type = 2) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, numvar);
      ELSE
        DBMS_SQL.COLUMN_VALUE(curid, i, datevar);
        -- statements
      END IF;
    END LOOP;
  END LOOP;

  DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

8.4 SQL Injection

SQL injection maliciously exploits applications that use client-supplied data in SQL statements, thereby gaining unauthorized access to a database to view or manipulate restricted data.

This section describes SQL injection vulnerabilities in PL/SQL and explains how to guard against them.

Topics

- SQL Injection Techniques
- Guards Against SQL Injection

Example 8-15 Setup for SQL Injection Examples

To try the examples, run these statements.

```sql
DROP TABLE secret_records;
CREATE TABLE secret_records (
    user_name    VARCHAR2(9),
    service_type VARCHAR2(12),
    value        VARCHAR2(30),
    date_created DATE
);

INSERT INTO secret_records (
    user_name, service_type, value, date_created
) VALUES ('Andy', 'Waiter', 'Serve dinner at Cafe Pete', SYSDATE);

INSERT INTO secret_records (
    user_name, service_type, value, date_created
) VALUES ('Chuck', 'Merger', 'Buy company XYZ', SYSDATE);
```

8.4.1 SQL Injection Techniques

All SQL injection techniques exploit a single vulnerability: String input is not correctly validated and is concatenated into a dynamic SQL statement.

Topics

- Statement Modification
- Statement Injection
- Data Type Conversion
8.4.1.1 Statement Modification

Statement modification means deliberately altering a dynamic SQL statement so that it runs in a way unintended by the application developer.

Typically, the user retrieves unauthorized data by changing the WHERE clause of a SELECT statement or by inserting a UNION ALL clause. The classic example of this technique is bypassing password authentication by making a WHERE clause always TRUE.

Example 8-16    Procedure Vulnerable to Statement Modification

This example creates a procedure that is vulnerable to statement modification and then invokes that procedure with and without statement modification. With statement modification, the procedure returns a supposedly secret record.

Create vulnerable procedure:

```sql
CREATE OR REPLACE PROCEDURE get_record (  
    user_name IN VARCHAR2,  
    service_type IN VARCHAR2,  
    rec OUT VARCHAR2  
) AUTHID DEFINER  
IS  
    query VARCHAR2(4000);  
BEGIN  
    -- Following SELECT statement is vulnerable to modification  
    -- because it uses concatenation to build WHERE clause.  
    query := 'SELECT value FROM secret_records WHERE user_name='''  
        || user_name  
        || ''' AND service_type='''  
        || service_type  
        || ''';  
    DBMS_OUTPUT.PUT_LINE('Query: ' || query);  
    EXECUTE IMMEDIATE query INTO rec;  
    DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);  
END;  
/
```

Demonstrate procedure without SQL injection:

```sql
SET SERVEROUTPUT ON;

DECLARE  
    record_value VARCHAR2(4000);  
BEGIN  
    get_record('Andy', 'Waiter', record_value);  
END;  
/
```

Result:
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND service_type='Waiter'
Rec: Serve dinner at Cafe Pete

Example of statement modification:

DECLARE
    record_value VARCHAR2(4000);
BEGIN
    get_record('Anybody '' OR service_type=''Merger''--',
                'Anything',
                record_value);
END;
/

Result:

Query: SELECT value FROM secret_records WHERE user_name='Anybody '' OR service_type=''Merger''--' AND service_type='Anything'
Rec: Buy company XYZ

PL/SQL procedure successfully completed.

8.4.1.2 Statement Injection

Statement injection means that a user appends one or more SQL statements to a dynamic SQL statement.

Anonymous PL/SQL blocks are vulnerable to this technique.

Example 8-17 Procedure Vulnerable to Statement Injection

This example creates a procedure that is vulnerable to statement injection and then invokes that procedure with and without statement injection. With statement injection, the procedure deletes the supposedly secret record exposed in Example 8-16.

Live SQL:

You can view and run this example on Oracle Live SQL at SQL Injection Demo

Create vulnerable procedure:

CREATE OR REPLACE PROCEDURE p (
    user_name IN VARCHAR2,
    service_type IN VARCHAR2
) AUTHID DEFINER
IS
    block1 VARCHAR2(4000);
BEGIN
    -- Following block is vulnerable to statement injection
    -- because it is built by concatenation.
    block1 :=
        'BEGIN
            DBMS_OUTPUT.PUT_LINE(''user_name: ' || user_name || ''');'
        ';
    END;
    exec block1;
    -- Query secret_records where user_name is Anybody and service_type is Merger
    SELECT value FROM secret_records WHERE user_name='Anybody' AND service_type='Merger'
    -- Rec: Buy company XYZ

PL/SQL procedure successfully completed.
Demonstrate procedure without SQL injection:

```sql
SET SERVEROUTPUT ON;
BEGIN
    p('Andy', 'Waiter');
END;
/
```

Result:

```
Block1: BEGIN
    DBMS_OUTPUT.PUT_LINE('user_name: Andy');
    DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
END;
user_name: Andy
service_type: Waiter
```

**SQL*Plus formatting command:**

```
COLUMN date_created FORMAT A12;
```

**Query:**

```
SELECT * FROM secret_records ORDER BY user_name;
```

Result:

```
USER_NAME  SERVICE_TYPE  VALUE                          DATE_CREATED
---------  ------------  ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      28-APR-10
Chuck     Merger       Buy company XYZ                28-APR-10
```

**Example of statement modification:**

```sql
BEGIN
    p('Anybody', 'Anything');
    DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
END;
/
```

Result:

```
Block1: BEGIN
    DBMS_OUTPUT.PUT_LINE('user_name: Anybody');
    DBMS_OUTPUT.PUT_LINE('service_type: Anything');
    DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
END;
user_name: Anybody
service_type: Anything
```

PL/SQL procedure successfully completed.
Query:

```sql
SELECT * FROM secret_records;
```

Result:

<table>
<thead>
<tr>
<th>USER_NAME</th>
<th>SERVICE_TYPE</th>
<th>VALUE</th>
<th>DATE_CREATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>Waiter</td>
<td>Serve dinner at Cafe Pete</td>
<td>18-MAR-09</td>
</tr>
</tbody>
</table>

1 row selected.

### 8.4.1.3 Data Type Conversion

A less known SQL injection technique uses NLS session parameters to modify or inject SQL statements.

A datetime or numeric value that is concatenated into the text of a dynamic SQL statement must be converted to the `VARCHAR2` data type. The conversion can be either implicit (when the value is an operand of the concatenation operator) or explicit (when the value is the argument of the `TO_CHAR` function). This data type conversion depends on the NLS settings of the database session that runs the dynamic SQL statement. The conversion of datetime values uses format models specified in the parameters `NLS_DATE_FORMAT`, `NLS_TIMESTAMP_FORMAT`, or `NLS_TIMESTAMP_TZ_FORMAT`, depending on the particular datetime data type. The conversion of numeric values applies decimal and group separators specified in the parameter `NLS_NUMERIC_CHARACTERS`.

One datetime format model is "text". The text is copied into the conversion result. For example, if the value of `NLS_DATE_FORMAT` is "Month: Month", then in June, `TO_CHAR(SYSDATE)` returns 'Month: June'. The datetime format model can be abused as shown in Example 8-18.

#### Example 8-18  Procedure Vulnerable to SQL Injection Through Data Type Conversion

```sql
SELECT * FROM secret_records;
```

Result:

<table>
<thead>
<tr>
<th>USER_NAME</th>
<th>SERVICE_TYPE</th>
<th>VALUE</th>
<th>DATE_CREATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>Waiter</td>
<td>Serve dinner at Cafe Pete</td>
<td>18-MAR-09</td>
</tr>
<tr>
<td>Chuck</td>
<td>Merger</td>
<td>Buy company XYZ</td>
<td>18-MAR-09</td>
</tr>
</tbody>
</table>

Create vulnerable procedure:

```sql
-- Return records not older than a month

CREATE OR REPLACE PROCEDURE get_recent_record (user_name IN VARCHAR2,
                                              service_type IN VARCHAR2,
                                              rec OUT VARCHAR2
                                        ) AUTHID DEFINER
IS
BEGIN
  /* Following SELECT statement is vulnerable to modification
   * because it uses concatenation to build WHERE clause
   * and because SYSDATE depends on the value of NLS_DATE_FORMAT. */
```
query := 'SELECT value FROM secret_records WHERE user_name='''
|| user_name
|| ''' AND service_type='''
|| service_type
|| ''' AND date_created>'''
|| (SYSDATE - 30)
|| '''

DBMS_OUTPUT.PUT_LINE('Query: ' || query);
EXECUTE IMMEDIATE query INTO rec;
DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/

Demonstrate procedure without SQL injection:

SET SERVEROUTPUT ON;
ALTERT SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';

DECLARE
    record_value VARCHAR2(4000);
BEGIN
    get_recent_record('Andy', 'Waiter', record_value);
END;
/

Result:

Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter' AND date_created>'29-MAR-2010'
Rec: Serve dinner at Cafe Pete

Example of statement modification:

ALTER SESSION SET NLS_DATE_FORMAT=''' OR service_type=''Merger''';

DECLARE
    record_value VARCHAR2(4000);
BEGIN
    get_recent_record('Anybody', 'Anything', record_value);
END;
/

Result:

Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created>'' OR service_type='Merger'
Rec: Buy company XYZ

PL/SQL procedure successfully completed.

8.4.2 Guards Against SQL Injection

If you use dynamic SQL in your PL/SQL applications, you must check the input text to
ensure that it is exactly what you expected.

You can use the following techniques:

- **Bind Variables**
8.4.2.1 Bind Variables

The most effective way to make your PL/SQL code invulnerable to SQL injection attacks is to use bind variables.

The database uses the values of bind variables exclusively and does not interpret their contents in any way. (Bind variables also improve performance.)

**Example 8-19  Bind Variables Guarding Against SQL Injection**

The procedure in this example is invulnerable to SQL injection because it builds the dynamic SQL statement with bind variables (not by concatenation as in the vulnerable procedure in Example 8-16). The same binding technique fixes the vulnerable procedure shown in Example 8-17.

Create invulnerable procedure:

```
CREATE OR REPLACE PROCEDURE get_record_2 (  
    user_name    IN  VARCHAR2,  
    service_type IN  VARCHAR2,  
    rec          OUT VARCHAR2  
) AUTHID DEFINER  
IS  
    query VARCHAR2(4000);  
BEGIN  
    query := 'SELECT value FROM secret_records  
        WHERE user_name=:a  
        AND service_type=:b';  
    DBMS_OUTPUT.PUT_LINE('Query: ' || query);  
    EXECUTE IMMEDIATE query INTO rec USING user_name, service_type;  
    DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);  
END;  
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;  
DECLARE  
    record_value VARCHAR2(4000);  
BEGIN  
    get_record_2('Andy', 'Waiter', record_value);  
END;  
/
```

Result:

```
Query: SELECT value FROM secret_records  
    WHERE user_name=:a  
    AND service_type=:b  
Rec: Serve dinner at Cafe Pete
```

PL/SQL procedure successfully completed.
Try statement modification:

DECLARE
    record_value VARCHAR2(4000);
BEGIN
    get_record_2('Anybody ' OR service_type=''Merger''--',
        'Anything',
        record_value);
END;
/

Result:

Query: SELECT value FROM secret_records
    WHERE user_name=:a
    AND service_type=:b

DECLARE

* ERROR at line 1:
  ORA-01403: no data found
  ORA-06512: at "HR.GET_RECORD_2", line 15
  ORA-06512: at line 4

8.4.2.2 Validation Checks

Always have your program validate user input to ensure that it is what is intended.

For example, if the user is passing a department number for a DELETE statement,
check the validity of this department number by selecting from the departments table.
Similarly, if a user enters the name of a table to be deleted, check that this table exists
by selecting from the static data dictionary view ALL_TABLES.

⚠️ Caution:

When checking the validity of a user name and its password, always return
the same error regardless of which item is invalid. Otherwise, a malicious
user who receives the error message "invalid password" but not "invalid user
name" (or the reverse) can realize that he or she has guessed one of these
correctly.

In validation-checking code, the subprograms in the DBMS_ASSERT package are often
useful. For example, you can use the DBMS_ASSERT.ENQUOTE_LITERAL function to
enclose a string literal in quotation marks, as Example 8-20 does. This prevents a
malicious user from injecting text between an opening quotation mark and its
corresponding closing quotation mark.

⚠️ Caution:

Although the DBMS_ASSERT subprograms are useful in validation code, they
do not replace it. For example, an input string can be a qualified SQL name
(verified by DBMS_ASSERT.QUALIFIED_SQL_NAME) and still be a fraudulent
password.
Example 8-20  Validation Checks Guarding Against SQL Injection

In this example, the procedure `raise_emp_salary` checks the validity of the column name that was passed to it before it updates the `employees` table, and then the anonymous block invokes the procedure from both a dynamic PL/SQL block and a dynamic SQL statement.

```sql
CREATE OR REPLACE PROCEDURE raise_emp_salary (column_value  NUMBER, emp_column    VARCHAR2, amount NUMBER ) AUTHID DEFINER
IS
  v_column  VARCHAR2(30);
  sql_stmt  VARCHAR2(200);
BEGIN
  -- Check validity of column name that was given as input:
  SELECT column_name INTO v_column
  FROM USER_TAB_COLS
  WHERE TABLE_NAME = 'EMPLOYEES'
  AND COLUMN_NAME = emp_column;
  sql_stmt := 'UPDATE employees SET salary = salary + :1 WHERE ' || DBMS_ASSERT.ENQUOTE_NAME(v_column, FALSE) || ' = :2';
  EXECUTE IMMEDIATE sql_stmt USING amount, column_value;
  -- If column name is valid:
  IF SQL%ROWCOUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Salaries were updated for: ' || emp_column || ' = ' || column_value);
  END IF;
  -- If column name is not valid:
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Invalid Column: ' || emp_column);
  END raise_emp_salary;
/

DECLARE
  plsql_block  VARCHAR2(500);
BEGIN
  -- Invoke raise_emp_salary from a dynamic PL/SQL block:
  plsql_block := 'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;';
  EXECUTE IMMEDIATE plsql_block USING 110, 'DEPARTMENT_ID', 10;
  -- Invoke raise_emp_salary from a dynamic SQL statement:
  EXECUTE IMMEDIATE 'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;' USING 112, 'EMPLOYEE_ID', 10;
```
Salaries were updated for: DEPARTMENT_ID = 110
Salaries were updated for: EMPLOYEE_ID = 112

8.4.2.3 Explicit Format Models

Using explicit locale-independent format models to construct SQL is recommended not only from a security perspective, but also to ensure that the dynamic SQL statement runs correctly in any globalization environment.

If you use datetime and numeric values that are concatenated into the text of a SQL or PL/SQL statement, and you cannot pass them as bind variables, convert them to text using explicit format models that are independent from the values of the NLS parameters of the running session. Ensure that the converted values have the format of SQL datetime or numeric literals.

**Example 8-21  Explicit Format Models Guarding Against SQL Injection**

This procedure is invulnerable to SQL injection because it converts the datetime parameter value, `SYSDATE - 30`, to a `VARCHAR2` value explicitly, using the `TO_CHAR` function and a locale-independent format model (not implicitly, as in the vulnerable procedure in Example 8-18).

Create invulnerable procedure:

```sql
CREATE OR REPLACE PROCEDURE get_recent_record
           (user_name     IN  VARCHAR2,
            service_type  IN  VARCHAR2,
            rec           OUT VARCHAR2)
             AUTHID DEFINER
          IS
             query VARCHAR2(4000);
          BEGIN
              query := 'SELECT value FROM secret_records WHERE user_name=''
                                || user_name
                                || '' AND service_type=''
                                || service_type
                                || '' AND date_created> DATE ''
                                || TO_CHAR(SYSDATE - 30,'YYYY-MM-DD')
                                || '''';
              DBMS_OUTPUT.PUT_LINE('Query: ' || query);
              EXECUTE IMMEDIATE query INTO rec;
              DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
          END;
```

Try statement modification:

```sql
ALTER SESSION SET NLS_DATE_FORMAT='' OR service_type=''Merger'';
```

DECLARE
record_value VARCHAR2(4000);
BEGIN
    get_recent_record('Anybody', 'Anything', record_value);
END;
/

Result:

Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND service_type='Anything' AND date_created> DATE '2010-03-29'
DECLARE
  *
  ERROR at line 1:
  ORA-01403: no data found
  ORA-06512: at "SYS.GET_RECENT_RECORD", line 21
  ORA-06512: at line 4
PL/SQL Subprograms

A PL/SQL subprogram is a named PL/SQL block that can be invoked repeatedly. If the subprogram has parameters, their values can differ for each invocation.

A subprogram is either a procedure or a function. Typically, you use a procedure to perform an action and a function to compute and return a value.

Topics

• Reasons to Use Subprograms
• Nested, Package, and Standalone Subprograms
• Subprogram Invocations
• Subprogram Properties
• Subprogram Parts
• Forward Declaration
• Subprogram Parameters
• Subprogram Invocation Resolution
• Overloaded Subprograms
• Recursive Subprograms
• Subprogram Side Effects
• PL/SQL Function Result Cache
• PL/SQL Functions that SQL Statements Can Invoke
• Invoker’s Rights and Definer’s Rights (AUTHID Property)
• External Subprograms

9.1 Reasons to Use Subprograms

Subprograms support the development and maintenance of reliable, reusable code with the following features:

• **Modularity**
  Subprograms let you break a program into manageable, well-defined modules.

• **Easier Application Design**
  When designing an application, you can defer the implementation details of the subprograms until you have tested the main program, and then refine them one step at a time. (To define a subprogram without implementation details, use the NULL statement, as in Example 5-30.)

• **Maintainability**
You can change the implementation details of a subprogram without changing its invokers.

- **Packageability**
  Subprograms can be grouped into packages, whose advantages are explained in "Reasons to Use Packages".

- **Reusability**
  Any number of applications, in many different environments, can use the same package subprogram or standalone subprogram.

- **Better Performance**
  Each subprogram is compiled and stored in executable form, which can be invoked repeatedly. Because stored subprograms run in the database server, a single invocation over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and invocation overhead.

Subprograms are an important component of other maintainability features, such as packages (explained in PL/SQL Packages) and Abstract Data Types (explained in "Abstract Data Types").

### 9.2 Nested, Package, and Standalone Subprograms

You can create a subprogram either inside a PL/SQL block (which can be another subprogram), inside a package, or at schema level.

A subprogram created inside a PL/SQL block is a **nested subprogram**. You can either declare and define it at the same time, or you can declare it first and then define it later in the same block (see "Forward Declaration"). A nested subprogram is stored in the database only if it is nested in a standalone or package subprogram.

A subprogram created inside a package is a **package subprogram**. You declare it in the package specification and define it in the package body. It is stored in the database until you drop the package. (Packages are described in PL/SQL Packages.)

A subprogram created at schema level is a **standalone subprogram**. You create it with the CREATE FUNCTION or CREATE PROCEDURE statement. It is stored in the database until you drop it with the DROP FUNCTION or DROP PROCEDURE statement. (These statements are described in SQL Statements for Stored PL/SQL Units.)

A **stored subprogram** is either a package subprogram or a standalone subprogram. A stored subprogram is affected by the AUTHID and ACCESSIBLE BY clauses, which can appear in the CREATE FUNCTION, CREATE PROCEDURE, and CREATE PACKAGE statements. The AUTHID clause affects the name resolution and privilege checking of SQL statements that the subprogram issues at run time (for more information, see "Invoker's Rights and Definer's Rights (AUTHID Property)"). The ACCESSIBLE BY clause specifies a white list of PL/SQL units that can access the subprogram.

### 9.3 Subprogram Invocations

A subprogram invocation has this form:

```
subprogram_name [ ( [ parameter [, parameter]... ] ) ]
```
If the subprogram has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

A procedure invocation is a PL/SQL statement. For example:

```plsql```
raise_salary(employee_id, amount);
```

A function invocation is an expression. For example:

```plsql```
new_salary := get_salary(employee_id);
```

IF salary_ok(new_salary, new_title) THEN ... 

See Also:

"Subprogram Parameters" for more information about specifying parameters in subprogram invocations

9.4 Subprogram Properties

Each subprogram property can appear only once in the subprogram declaration. The properties can appear in any order. Properties appear before the IS or AS keyword in the subprogram heading. The properties cannot appear in nested subprograms.

Only the ACCESSIBLE BY property can appear in package subprograms. Standalone subprograms may have the following properties in their declaration.

- ACCESSIBLE BY Clause
- DEFAULT COLLATION Clause
- Invoker's Rights and Definer's Rights (AUTHID Property)

9.5 Subprogram Parts

A subprogram begins with a subprogram heading, which specifies its name and (optionally) its parameter list.

Like an anonymous block, a subprogram has these parts:

- Declarative part (optional)
  - This part declares and defines local types, cursors, constants, variables, exceptions, and nested subprograms. These items cease to exist when the subprogram completes execution.
  - This part can also specify pragmas.

  Note:

  The declarative part of a subprogram does not begin with the keyword DECLARE, as the declarative part of an anonymous block does.

- Executable part (required)
This part contains one or more statements that assign values, control execution, and manipulate data. (Early in the application design process, this part might contain only a NULL statement, as in Example 5-30.)

- **Exception-handling part (optional)**
  This part contains code that handles runtime errors.

**Topics**

- Additional Parts for Functions
- RETURN Statement

---

### See Also:

- "Pragmas"
- "Procedure Declaration and Definition" for the syntax of procedure declarations and definitions
- "Subprogram Parameters" for more information about subprogram parameters

---

### Example 9-1 Declaring, Defining, and Invoking a Simple PL/SQL Procedure

In this example, an anonymous block simultaneously declares and defines a procedure and invokes it three times. The third invocation raises the exception that the exception-handling part of the procedure handles.

```sql
DECLARE
    first_name employees.first_name%TYPE;
    last_name  employees.last_name%TYPE;
    email      employees.email%TYPE;
    employer   VARCHAR2(8) := 'AcmeCorp';
BEGIN
    first_name := 'John';
    last_name  := 'Doe';
    create_email(first_name, last_name, employer); -- invocation
END;
```

---

---

---
create_email(last_name, first_name, employer);  -- invocation
DBMS_OUTPUT.PUT_LINE ('With last name first, email is: ' || email);

first_name := 'Elizabeth';
last_name  := 'MacDonald';
create_email(first_name, last_name, employer);  -- invocation
END;
/

Result:
With first name first, email is: John.Doe@AcmeCorp
With last name first, email is: Doe.John@AcmeCorp
Email address is too long.

9.5.1 Additional Parts for Functions

A function has the same structure as a procedure, except that:

- A function heading must include a `RETURN clause`, which specifies the data type of
  the value that the function returns. (A procedure heading cannot have a `RETURN`
  clause.)

- In the executable part of a function, every execution path must lead to a `RETURN`
  statement. Otherwise, the PL/SQL compiler issues a compile-time warning. (In a
  procedure, the `RETURN` statement is optional and not recommended. For details,
  see "RETURN Statement".)

- A function declaration can include these options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETERMINISTIC option</td>
<td>Helps the optimizer avoid redundant function invocations.</td>
</tr>
<tr>
<td>PARALLEL_ENABLE option</td>
<td>Enables the function for parallel execution, making it safe for use in slave sessions of parallel DML evaluations.</td>
</tr>
<tr>
<td>PIPELINED option</td>
<td>Makes a table function pipelined, for use as a row source.</td>
</tr>
<tr>
<td>RESULT_CACHE option</td>
<td>Stores function results in the PL/SQL function result cache.</td>
</tr>
</tbody>
</table>

See Also:

- "Function Declaration and Definition" for the syntax of function declarations and definitions, including descriptions of the items in the preceding table
- "PL/SQL Function Result Cache" for more information about the `RESULT_CACHE` option

Example 9-2    Declaring, Defining, and Invoking a Simple PL/SQL Function

In this example, an anonymous block simultaneously declares and defines a function and invokes it.
DECLARE
    -- Declare and define function
FUNCTION square (original NUMBER)  -- parameter list
    RETURN NUMBER  -- RETURN clause
AS  -- Declarative part begins
    original_squared NUMBER;
BEGIN  -- Executable part begins
    original_squared := original * original;
    RETURN original_squared;  -- RETURN statement
END;
BEGIN
    DBMS_OUTPUT.PUT_LINE(square(100));  -- invocation
END;
/

Result:
10000

9.5.2 RETURN Statement

The RETURN statement immediately ends the execution of the subprogram or anonymous block that contains it. A subprogram or anonymous block can contain multiple RETURN statements.

Topics
- RETURN Statement in Function
- RETURN Statement in Procedure
- RETURN Statement in Anonymous Block

See Also:
“RETURN Statement” for the syntax of the RETURN statement

9.5.2.1 RETURN Statement in Function

In a function, every execution path must lead to a RETURN statement and every RETURN statement must specify an expression. The RETURN statement assigns the value of the expression to the function identifier and returns control to the invoker, where execution resumes immediately after the invocation.

Note:
In a pipelined table function, a RETURN statement need not specify an expression. For information about the parts of a pipelined table function, see “Creating Pipelined Table Functions”. 
In Example 9-3, the anonymous block invokes the same function twice. The first time, the \texttt{RETURN} statement returns control to the inside of the invoking statement. The second time, the \texttt{RETURN} statement returns control to the statement immediately after the invoking statement.

In Example 9-4, the function has multiple \texttt{RETURN} statements, but if the parameter is not 0 or 1, then no execution path leads to a \texttt{RETURN} statement. The function compiles with warning PLW-05005: subprogram \texttt{F} returns without value at line 11.

Example 9-5 is like Example 9-4, except for the addition of the \texttt{ELSE} clause. Every execution path leads to a \texttt{RETURN} statement, and the function compiles without warning PLW-05005.

**Example 9-3  Execution Resumes After RETURN Statement in Function**

```sql
DECLARE
  x INTEGER;

FUNCTION f (n INTEGER) RETURN INTEGER IS BEGIN RETURN (n*n); END;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('f returns ' || f(2) || '. Execution returns here (1).');
  x := f(2);
  DBMS_OUTPUT.PUT_LINE('Execution returns here (2).');
END;
/
```

Result:

f returns 4. Execution returns here (1). Execution returns here (2).

**Example 9-4  Function Where Not Every Execution Path Leads to RETURN Statement**

```sql
CREATE OR REPLACE FUNCTION f (n INTEGER) RETURN INTEGER AUTHID DEFINER IS BEGIN IF n = 0 THEN RETURN 1; ELSIF n = 1 THEN RETURN n; END IF; END;
/
```

**Example 9-5  Function Where Every Execution Path Leads to RETURN Statement**

```sql
CREATE OR REPLACE FUNCTION f (n INTEGER) RETURN INTEGER AUTHID DEFINER
```
IS
BEGIN
  IF n = 0 THEN
    RETURN 1;
  ELSIF n = 1 THEN
    RETURN n;
  ELSE
    RETURN n*n;
  END IF;
END;
/
BEGIN
  FOR i IN 0 .. 3 LOOP
    DBMS_OUTPUT.PUT_LINE('f(' || i || ') = ' || f(i));
  END LOOP;
END;
/

Result:
  f(0) = 1
  f(1) = 1
  f(2) = 4
  f(3) = 9

9.5.2.2 RETURN Statement in Procedure

In a procedure, the RETURN statement returns control to the invoker, where execution resumes immediately after the invocation. The RETURN statement cannot specify an expression.

In Example 9-6, the RETURN statement returns control to the statement immediately after the invoking statement.

Example 9-6    Execution Resumes After RETURN Statement in Procedure

DECLARE
  PROCEDURE p IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Inside p');
      RETURN;
      DBMS_OUTPUT.PUT_LINE('Unreachable statement.');
    END;
    BEGIN
      p;
      DBMS_OUTPUT.PUT_LINE('Control returns here.');
    END;
/

Result:
  Inside p
  Control returns here.

9.5.2.3 RETURN Statement in Anonymous Block

In an anonymous block, the RETURN statement exits its own block and all enclosing blocks. The RETURN statement cannot specify an expression.

In Example 9-7, the RETURN statement exits both the inner and outer block.
Example 9-7  Execution Resumes After RETURN Statement in Anonymous Block

BEGIN
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside inner block. ');
    RETURN;
    DBMS_OUTPUT.PUT_LINE('Unreachable statement. ');
  END;
  DBMS_OUTPUT.PUT_LINE('Inside outer block. Unreachable statement. ');
END;
/

Result:
Inside inner block.

9.6 Forward Declaration

If nested subprograms in the same PL/SQL block invoke each other, then one requires a forward declaration, because a subprogram must be declared before it can be invoked.

A forward declaration declares a nested subprogram but does not define it. You must define it later in the same block. The forward declaration and the definition must have the same subprogram heading.

In Example 9-8, an anonymous block creates two procedures that invoke each other.

Example 9-8  Nested Subprograms Invoke Each Other

DECLARE
  -- Declare proc1 (forward declaration):
  PROCEDURE proc1(number1 NUMBER);

  -- Declare and define proc2:
  PROCEDURE proc2(number2 NUMBER) IS
    BEGIN
      proc1(number2);
    END;

  -- Define proc 1:
  PROCEDURE proc1(number1 NUMBER) IS
    BEGIN
      proc2 (number1);
    END;

BEGIN
  NULL;
END;
/

9.7 Subprogram Parameters

If a subprogram has parameters, their values can differ for each invocation.

Topics
  •  Formal and Actual Subprogram Parameters
9.7.1 Formal and Actual Subprogram Parameters

If you want a subprogram to have parameters, declare **formal parameters** in the subprogram heading. In each formal parameter declaration, specify the name and data type of the parameter, and (optionally) its mode and default value. In the execution part of the subprogram, reference the formal parameters by their names.

When invoking the subprogram, specify the **actual parameters** whose values are to be assigned to the formal parameters. Corresponding actual and formal parameters must have compatible data types.

---

**Note:**

You can declare a formal parameter of a constrained subtype, like this:

```plaintext
DECLARE
    SUBTYPE n1 IS NUMBER(1);
    SUBTYPE v1 IS VARCHAR2(1);

    PROCEDURE p (n n1, v v1) IS ...
```

But you cannot include a constraint in a formal parameter declaration, like this:

```plaintext
DECLARE
    PROCEDURE p (n NUMBER(1), v VARCHAR2(1)) IS ...
```

---

**Tip:**

To avoid confusion, use different names for formal and actual parameters.

---

**Note:**

- Actual parameters (including default values of formal parameters) can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.
- You cannot use LOB parameters in a server-to-server remote procedure call (RPC).
In Example 9-9, the procedure has formal parameters `emp_id` and `amount`. In the first procedure invocation, the corresponding actual parameters are `emp_num` and `bonus`, whose value are 120 and 100, respectively. In the second procedure invocation, the actual parameters are `emp_num` and `merit + bonus`, whose value are 120 and 150, respectively.

Topics:

• Formal Parameters of Constrained Subtypes

See Also:

• “Formal Parameter Declaration” for the syntax and semantics of a formal parameter declaration
• “function_call ::=” and “function_call” for the syntax and semantics of a function invocation
• “procedure_call ::=” and “procedure” for the syntax and semantics of a procedure invocation

Example 9-9  Formal Parameters and Actual Parameters

```sql
DECLARE
    emp_num NUMBER(6) := 120;
    bonus   NUMBER(6) := 100;
    merit   NUMBER(4) := 50;

PROCEDURE raise_salary (
    emp_id NUMBER,  -- formal parameter
    amount NUMBER   -- formal parameter
) IS
BEGIN
    UPDATE employees
    SET salary = salary + amount  -- reference to formal parameter
    WHERE employee_id = emp_id;  -- reference to formal parameter
END raise_salary;

BEGIN
    raise_salary(emp_num, bonus); -- actual parameters
    /* raise_salary runs this statement:
       UPDATE employees
       SET salary = salary + 100
       WHERE employee_id = 120;   */

    raise_salary(emp_num, merit + bonus); -- actual parameters
    /* raise_salary runs this statement:
       UPDATE employees
       SET salary = salary + 150
       WHERE employee_id = 120;   */
END;
/```
9.7.1.1 Formal Parameters of Constrained Subtypes

If the data type of a formal parameter is a constrained subtype, then:

- If the subtype has the **NOT NULL** constraint, then the actual parameter inherits it.
- If the subtype has the base type **VARCHAR2**, then the actual parameter does not inherit the size of the subtype.
- If the subtype has a numeric base type, then the actual parameter inherits the range of the subtype, but not the precision or scale.

**Note:**

In a function, the clause `RETURN datatype` declares a hidden formal parameter and the statement `RETURN value` specifies the corresponding actual parameter. Therefore, if `datatype` is a constrained data type, then the preceding rules apply to `value` (see Example 9-11).

Example 9-10 shows that an actual subprogram parameter inherits the **NOT NULL** constraint but not the size of a **VARCHAR2** subtype.

As **PL/SQL Predefined Data Types** shows, PL/SQL has many predefined data types that are constrained subtypes of other data types. For example, **INTEGER** is a constrained subtype of **NUMBER**:

```
SUBTYPE INTEGER IS NUMBER(38,0);
```

In Example 9-11, the function has both an **INTEGER** formal parameter and an **INTEGER** return type. The anonymous block invokes the function with an actual parameter that is not an integer. Because the actual parameter inherits the range but not the precision and scale of **INTEGER**, and the actual parameter is in the **INTEGER** range, the invocation succeeds. For the same reason, the `RETURN` statement succeeds in returning the noninteger value.

In Example 9-12, the function implicitly converts its formal parameter to the constrained subtype **INTEGER** before returning it.

**See Also:**

"**Constrained Subtypes**" for general information about constrained subtypes

Example 9-10  Actual Parameter Inherits Only NOT NULL from Subtype

```sql
DECLARE
    SUBTYPE License IS VARCHAR2(7) NOT NULL;
    n  License := 'DLLLDDD';

PROCEDURE p (x License) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(x);
END;
```
BEGIN
  p('1ABC123456789');  -- Succeeds; size is not inherited
  p(NULL);             -- Raises error; NOT NULL is inherited
END;
/

Result:
  p(NULL);             -- Raises error; NOT NULL is inherited
  *
  ERROR at line 12:
  ORA-06550: line 12, column 5:
  PLS-00567: cannot pass NULL to a NOT NULL constrained formal parameter
  ORA-06550: line 12, column 3:
  PL/SQL: Statement ignored

Example 9-11  Actual Parameter and Return Value Inherit Only Range From Subtype

DECLARE
  FUNCTION test (p INTEGER) RETURN INTEGER IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('p = ' || p);
    RETURN p;
  END test;

BEGIN
  DBMS_OUTPUT.PUT_LINE('test(p) = ' || test(0.66));
END;
/

Result:
  p = .66
  test(p) = .66
PL/SQL procedure successfully completed.

Example 9-12  Function Implicitly Converts Formal Parameter to Constrained Subtype

DECLARE
  FUNCTION test (p NUMBER) RETURN NUMBER IS
    q INTEGER := p;  -- Implicitly converts p to INTEGER
  BEGIN
    DBMS_OUTPUT.PUT_LINE('p = ' || q);  -- Display q, not p
    RETURN q;                           -- Return q, not p
  END test;

BEGIN
  DBMS_OUTPUT.PUT_LINE('test(p) = ' || test(0.66));
END;
/

Result:
  p = 1
  test(p) = 1
PL/SQL procedure successfully completed.
9.7.2 Subprogram Parameter Passing Methods

The PL/SQL compiler has two ways of passing an actual parameter to a subprogram:

- **By reference**
  The compiler passes the subprogram a pointer to the actual parameter. The actual and formal parameters refer to the same memory location.

- **By value**
  The compiler assigns the value of the actual parameter to the corresponding formal parameter. The actual and formal parameters refer to different memory locations.

If necessary, the compiler implicitly converts the data type of the actual parameter to the data type of the formal parameter. For information about implicit data conversion, see *Oracle Database SQL Language Reference*.

Tip:

Avoid implicit data conversion (for the reasons in *Oracle Database SQL Language Reference*), in either of these ways:

- Declare the variables that you intend to use as actual parameters with the same data types as their corresponding formal parameters (as in the declaration of variable \( x \) in Example 9-13).
- Explicitly convert actual parameters to the data types of their corresponding formal parameters, using the SQL conversion functions described in *Oracle Database SQL Language Reference* (as in the third invocation of the procedure in Example 9-13).

In Example 9-13, the procedure \( p \) has one parameter, \( n \), which is passed by value. The anonymous block invokes \( p \) three times, avoiding implicit conversion twice.

The method by which the compiler passes a specific actual parameter depends on its mode, as explained in "Subprogram Parameter Modes".

**Example 9-13  Avoiding Implicit Conversion of Actual Parameters**

```sql
CREATE OR REPLACE PROCEDURE p (n NUMBER) AUTHID DEFINER IS
  BEGIN
    NULL;
  END;
/

DECLARE
  x NUMBER := 1;
  y VARCHAR2(1) := '1';
BEGIN
  p(x);         -- No conversion needed
  p(y);         -- z implicitly converted from VARCHAR2 to NUMBER
  p(TO_NUMBER(y)); -- z explicitly converted from VARCHAR2 to NUMBER
END;
/```
9.7.3 Subprogram Parameter Modes

The **mode** of a formal parameter determines its behavior.

Table 9-1 summarizes and compares the characteristics of the subprogram parameter modes.

### Table 9-1  PL/SQL Subprogram Parameter Modes

<table>
<thead>
<tr>
<th>Parameter Mode</th>
<th>Is Default?</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Default mode</td>
<td>Passes a value to the subprogram.</td>
</tr>
<tr>
<td>OUT</td>
<td>Must be specified.</td>
<td>Returns a value to the invoker.</td>
</tr>
<tr>
<td>IN OUT</td>
<td>Must be specified.</td>
<td>Passes an initial value to the subprogram and returns an updated value to the invoker.</td>
</tr>
</tbody>
</table>

### Table 9-2  PL/SQL Subprogram Parameter Modes Characteristics

<table>
<thead>
<tr>
<th>Parameter Mode</th>
<th>Formal Parameter</th>
<th>Actual Parameter</th>
<th>Passed by Reference?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Formal parameter acts like a constant: When the subprogram begins, its value is that of either its actual parameter or default value, and the subprogram cannot change this value.</td>
<td>Actual parameter can be a constant, initialized variable, literal, or expression.</td>
<td>Actual parameter is passed by reference.</td>
</tr>
<tr>
<td>OUT</td>
<td>Formal parameter is initialized to the default value of its type. The default value of the type is <code>NULL</code> except for a record type with a non-<code>NULL</code> default value (see Example 9-16). When the subprogram begins, the formal parameter has its initial value regardless of the value of its actual parameter. Oracle recommends that the subprogram assign a value to the formal parameter.</td>
<td>If the default value of the formal parameter type is <code>NULL</code>, then the actual parameter must be a variable whose data type is not defined as NOT <code>NULL</code>.</td>
<td>By default, actual parameter is passed by value; if you specify NOCOPY, it might be passed by reference.</td>
</tr>
<tr>
<td>IN OUT</td>
<td>Formal parameter acts like an initialized variable: When the subprogram begins, its value is that of its actual parameter. Oracle recommends that the subprogram update its value.</td>
<td>Actual parameter must be a variable (typically, it is a string buffer or numeric accumulator).</td>
<td>By default, actual parameter is passed by value (in both directions); if you specify NOCOPY, it might be passed by reference.</td>
</tr>
</tbody>
</table>
Tip:

Do not use OUT and IN OUT for function parameters. Ideally, a function takes zero or more parameters and returns a single value. A function with IN OUT parameters returns multiple values and has side effects.

Note:

The specifications of many packages and types that Oracle Database supplies declare formal parameters with this notation:

```sql
i1 IN VARCHAR2 CHARACTER SET ANY_CS
i2 IN VARCHAR2 CHARACTER SET i1%CHARSET
```

Do not use this notation when declaring your own formal or actual parameters. It is reserved for Oracle implementation of the supplied packages types.

Regardless of how an OUT or IN OUT parameter is passed:

- If the subprogram exits successfully, then the value of the actual parameter is the final value assigned to the formal parameter. (The formal parameter is assigned at least one value—the initial value.)
- If the subprogram ends with an exception, then the value of the actual parameter is undefined.
- Formal OUT and IN OUT parameters can be returned in any order. In this example, the final values of `x` and `y` are undefined:

```sql
CREATE OR REPLACE PROCEDURE p (x OUT INTEGER, y OUT INTEGER) AS
BEGIN
  x := 17; y := 93;
END;
/
```

When an OUT or IN OUT parameter is passed by reference, the actual and formal parameters refer to the same memory location. Therefore, if the subprogram changes the value of the formal parameter, the change shows immediately in the actual parameter (see "Subprogram Parameter Aliasing with Parameters Passed by Reference").

In Example 9-14, the procedure `p` has two IN parameters, one OUT parameter, and one IN OUT parameter. The OUT and IN OUT parameters are passed by value (the default). The anonymous block invokes `p` twice, with different actual parameters. Before each invocation, the anonymous block prints the values of the actual parameters. The procedure `p` prints the initial values of its formal parameters. After each invocation, the anonymous block prints the values of the actual parameters again.

In Example 9-15, the anonymous block invokes procedure `p` (from Example 9-14) with an actual parameter that causes `p` to raise the predefined exception `ZERO_DIVIDE`, which `p` does not handle. The exception propagates to the anonymous block, which handles `ZERO_DIVIDE` and shows that the actual parameters for the IN and IN OUT
parameters of \( p \) have retained the values that they had before the invocation.  
(Exception propagation is explained in "Exception Propagation".)

In Example 9-16, the procedure \( p \) has three \texttt{OUT} formal parameters: \( x \), of a record type with a non-\texttt{NULL} default value; \( y \), of a record type with no non-\texttt{NULL} default value; and \( z \), which is not a record.

The corresponding actual parameters for \( x \), \( y \), and \( z \) are \( r1 \), \( r2 \), and \( s \), respectively. \( s \) is declared with an initial value. However, when \( p \) is invoked, the value of \( s \) is initialized to \texttt{NULL}. The values of \( r1 \) and \( r2 \) are initialized to the default values of their record types, 'abcde' and \texttt{NULL}, respectively.

Example 9-14  Parameter Values Before, During, and After Procedure Invocation

```sql
CREATE OR REPLACE PROCEDURE p (  
  a PLS_INTEGER, -- IN by default  
  b IN PLS_INTEGER,  
  c OUT PLS_INTEGER,  
  d IN OUT BINARY_FLOAT  
) AUTHID DEFINER IS  
BEGIN  
  -- Print values of parameters:  
  DBMS_OUTPUT.PUT_LINE('Inside procedure p:');  
  DBMS_OUTPUT.PUT('IN a = ');  
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(a), 'NULL'));  
  DBMS_OUTPUT.PUT('IN b = ');  
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(b), 'NULL'));  
  DBMS_OUTPUT.PUT('OUT c = ');  
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(c), 'NULL'));  
  DBMS_OUTPUT.PUT_LINE('IN OUT d = ' || TO_CHAR(d));  
  -- Can reference IN parameters a and b,  
  -- but cannot assign values to them.  
  c := a+10; -- Assign value to OUT parameter  
  d := 10/b; -- Assign value to IN OUT parameter  
END;  
/

DECLARE  
  aa CONSTANT PLS_INTEGER := 1;  
  bb PLS_INTEGER := 2;  
  cc PLS_INTEGER := 3;  
  dd BINARY_FLOAT := 4;  
  ee PLS_INTEGER;  
  ff BINARY_FLOAT := 5;  
BEGIN  
  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');  
  DBMS_OUTPUT.PUT('aa = ');  
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));  
  DBMS_OUTPUT.PUT('bb = ');  
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));  
  DBMS_OUTPUT.PUT('cc = ');  
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));  
END;
```
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));

DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));

p (aa, -- constant
   bb, -- initialized variable
   cc, -- initialized variable
   dd -- initialized variable
);

DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');

DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));

DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));

DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));

DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));

DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');

DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));

DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));

p (1, -- literal
   (bb+3)*4, -- expression
   ee, -- uninitialized variable
   ff -- initialized variable
);

DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');

DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));

DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));

END;
/

Result:

Before invoking procedure p:
   aa = 1
   bb = 2
   cc = 3
   dd = 4.0E+000
Inside procedure p:
   IN a = 1
   IN b = 2
   OUT c = NULL
   IN OUT d = 4.0E+000
After invoking procedure p:
   aa = 1
   bb = 2
   cc = 11
Example 9-15    OUT and IN OUT Parameter Values After Exception Handling

DECLARE
    j  PLS_INTEGER  := 10;
    k  BINARY_FLOAT := 15;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
    DBMS_OUTPUT.PUT('j = ' || TO_CHAR(j));
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(j), 'NULL'));
    DBMS_OUTPUT.PUT_LINE('k = ' || TO_CHAR(k));
    p(4, 0, j, k);  -- causes p to exit with exception ZERO_DIVIDE
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
        DBMS_OUTPUT.PUT('j = ' || TO_CHAR(j));
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(j), 'NULL'));
        DBMS_OUTPUT.PUT_LINE('k = ' || TO_CHAR(k));
END;
/

Result:
Before invoking procedure p:
    j = 10
    k = 1.5E+001
Inside procedure p:
    IN a = 4
    IN b = 0
    OUT c = NULL
    IN OUT d = 1.5E+001
After invoking procedure p:
    j = 10
    k = 1.5E+001

PL/SQL procedure successfully completed.

Example 9-16    OUT Formal Parameter of Record Type with Non-NULL Default Value

CREATE OR REPLACE PACKAGE r_types AUTHID DEFINER IS
    TYPE r_type_1 IS RECORD (f VARCHAR2(5) := 'abcde');
TYPE r_type_2 IS RECORD (f VARCHAR2(5));
END;
/

CREATE OR REPLACE PROCEDURE p (
    x OUT r_types.r_type_1,
    y OUT r_types.r_type_2,
    z OUT VARCHAR2
) AUTHID CURRENT_USER IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('x.f is ' || NVL(x.f,'NULL'));
    DBMS_OUTPUT.PUT_LINE('y.f is ' || NVL(y.f,'NULL'));
    DBMS_OUTPUT.PUT_LINE('z is ' || NVL(z,'NULL'));
END;
/

DECLARE
    r1 r_types.r_type_1;
    r2 r_types.r_type_2;
    s  VARCHAR2(5) := 'fghij';
BEGIN
    p (r1, r2, s);
    END;
/

Result:

x.f is abcde
y.f is NULL
z is NULL

PL/SQL procedure successfully completed.

9.7.4 Subprogram Parameter Aliasing

Aliasing is having two different names for the same memory location. If a stored item is visible by more than one path, and you can change the item by one path, then you can see the change by all paths.

Subprogram parameter aliasing always occurs when the compiler passes an actual parameter by reference, and can also occur when a subprogram has cursor variable parameters.

Topics

- Subprogram Parameter Aliasing with Parameters Passed by Reference
- Subprogram Parameter Aliasing with Cursor Variable Parameters

9.7.4.1 Subprogram Parameter Aliasing with Parameters Passed by Reference

When the compiler passes an actual parameter by reference, the actual and formal parameters refer to the same memory location. Therefore, if the subprogram changes the value of the formal parameter, the change shows immediately in the actual parameter.

The compiler always passes IN parameters by reference, but the resulting aliasing cannot cause problems, because subprograms cannot assign values to IN parameters.
The compiler *might* pass an OUT or IN OUT parameter by reference, if you specify NOCOPY for that parameter. NOCOPY is only a hint—each time the subprogram is invoked, the compiler decides, silently, whether to obey or ignore NOCOPY. Therefore, aliasing can occur for one invocation but not another, making subprogram results indeterminate. For example:

- If the actual parameter is a global variable, then an assignment to the formal parameter *might* show in the global parameter (see Example 9-17).
- If the same variable is the actual parameter for two formal parameters, then an assignment to either formal parameter *might* show immediately in both formal parameters (see Example 9-18).
- If the actual parameter is a package variable, then an assignment to either the formal parameter or the package variable *might* show immediately in both the formal parameter and the package variable.
- If the subprogram is exited with an unhandled exception, then an assignment to the formal parameter *might* show in the actual parameter.

See Also:

"NOCOPY" for the cases in which the compiler always ignores NOCOPY

In Example 9-17, the procedure has an IN OUT NOCOPY formal parameter, to which it assigns the value 'aardvark'. The anonymous block assigns the value 'aardwolf' to a global variable and then passes the global variable to the procedure. If the compiler obeys the NOCOPY hint, then the final value of the global variable is 'aardvark'. If the compiler ignores the NOCOPY hint, then the final value of the global variable is 'aardwolf'.

In Example 9-18, the procedure has an IN parameter, an IN OUT parameter, and an IN OUT NOCOPY parameter. The anonymous block invokes the procedure, using the same actual parameter, a global variable, for all three formal parameters. The procedure changes the value of the IN OUT parameter before it changes the value of the IN OUT NOCOPY parameter. However, if the compiler obeys the NOCOPY hint, then the latter change shows in the actual parameter immediately. The former change shows in the actual parameter after the procedure is exited successfully and control returns to the anonymous block.

Example 9-17  Aliasing from Global Variable as Actual Parameter

DECLARE
  TYPE Definition IS RECORD (
    word VARCHAR2(20),
    meaning VARCHAR2(200)
  );

  TYPE Dictionary IS VARRAY(2000) OF Definition;

  lexicon  Dictionary := Dictionary();  -- global variable

  PROCEDURE add_entry (  
    word_list IN OUT NOCOPY Dictionary  -- formal NOCOPY parameter
  ) IS
    BEGIN

word_list(1).word := 'aardvark';
END;

BEGIN
  lexicon.EXTEND;
  lexicon(1).word := 'aardwolf';
  add_entry(lexicon);  -- global variable is actual parameter
  DBMS_OUTPUT.PUT_LINE(lexicon(1).word);
END;
/

Result:
aardvark

Example 9-18    Aliasing from Same Actual Parameter for Multiple Formal Parameters

DECLARE
  n NUMBER := 10;

PROCEDURE p (  
  n1 IN NUMBER,     
  n2 IN OUT NUMBER, 
  n3 IN OUT NOCOPY NUMBER 
) IS
BEGIN
  n2 := 20;  -- actual parameter is 20 only after procedure succeeds
  DBMS_OUTPUT.put_line(n1);  -- actual parameter value is still 10
  n3 := 30;  -- might change actual parameter immediately
  DBMS_OUTPUT.put_line(n1);  -- actual parameter value is either 10 or 30
END;

BEGIN
  p(n, n, n);
  DBMS_OUTPUT.put_line;
END;
/

Result if the compiler obeys the NOCOPY hint:
10
30
20

Result if the compiler ignores the NOCOPY hint:
10
10
30

9.7.4.2 Subprogram Parameter Aliasing with Cursor Variable Parameters

Cursor variable parameters are pointers. Therefore, if a subprogram assigns one cursor variable parameter to another, they refer to the same memory location. This aliasing can have unintended results.

In Example 9-19, the procedure has two cursor variable parameters, emp_cv1 and emp_cv2. The procedure opens emp_cv1 and assigns its value (which is a pointer) to emp_cv2. Now emp_cv1 and emp_cv2 refer to the same memory location. When the
procedure closes `emp_cv1`, it also closes `emp_cv2`. Therefore, when the procedure tries to fetch from `emp_cv2`, PL/SQL raises an exception.

**Example 9-19 Aliasing from Cursor Variable Subprogram Parameters**

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    c1 EmpCurTyp;
    c2 EmpCurTyp;

    PROCEDURE get_emp_data (  
        emp_cv1 IN OUT EmpCurTyp,  
        emp_cv2 IN OUT EmpCurTyp  
    ) IS
        emp_rec employees%ROWTYPE;
    BEGIN
        OPEN emp_cv1 FOR SELECT * FROM employees;
        emp_cv2 := emp_cv1;  -- now both variables refer to same location
        FETCH emp_cv1 INTO emp_rec;  -- fetches first row of employees
        FETCH emp_cv1 INTO emp_rec;  -- fetches second row of employees
        FETCH emp_cv2 INTO emp_rec;  -- fetches third row of employees
        CLOSE emp_cv1;  -- closes both variables
        FETCH emp_cv2 INTO emp_rec; -- causes error when get_emp_data is invoked
    END;
BEGIN
    get_emp_data(c1, c2);
END;
/ 
```

Result:

```
DECLARE
    *
ERROR at line 1:
ORA-01001: invalid cursor
ORA-06512: at line 19
ORA-06512: at line 22
```

### 9.7.5 Default Values for IN Subprogram Parameters

When you declare a formal IN parameter, you can specify a default value for it. A formal parameter with a default value is called an **optional parameter**, because its corresponding actual parameter is optional in a subprogram invocation. If the actual parameter is omitted, then the invocation assigns the default value to the formal parameter. A formal parameter with no default value is called a **required parameter**, because its corresponding actual parameter is required in a subprogram invocation.

Omitting an actual parameter does not make the value of the corresponding formal parameter NULL. To make the value of a formal parameter NULL, specify NULL as either the default value or the actual parameter.

In **Example 9-20**, the procedure has one required parameter and two optional parameters.

In **Example 9-20**, the procedure invocations specify the actual parameters in the same order as their corresponding formal parameters are declared—that is, the invocations use positional notation. Positional notation does not let you omit the second parameter of `raise_salary` but specify the third; to do that, you must use either named or mixed
notation. For more information, see “Positional, Named, and Mixed Notation for Actual Parameters”.

The default value of a formal parameter can be any expression whose value can be assigned to the parameter; that is, the value and parameter must have compatible data types. If a subprogram invocation specifies an actual parameter for the formal parameter, then that invocation does not evaluate the default value.

In Example 9-21, the procedure \texttt{p} has a parameter whose default value is an invocation of the function \texttt{f}. The function \texttt{f} increments the value of a global variable. When \texttt{p} is invoked without an actual parameter, \texttt{p} invokes \texttt{f}, and \texttt{f} increments the global variable. When \texttt{p} is invoked with an actual parameter, \texttt{p} does not invoke \texttt{f}, and value of the global variable does not change.

Example 9-22 creates a procedure with two required parameters, invokes it, and then adds a third, optional parameter. Because the third parameter is optional, the original invocation remains valid.

Example 9-20  Procedure with Default Parameter Values

DECLARE  
PROCEDURE raise_salary (  
    emp_id IN employees.employee_id%TYPE,  
    amount IN employees.salary%TYPE := 100,  
    extra IN employees.salary%TYPE := 50  
) IS  
BEGIN  
UPDATE employees  
SET salary = salary + amount + extra  
WHERE employee_id = emp_id;  
END raise_salary;

BEGIN  
raise_salary(120);       -- same as raise_salary(120, 100, 50)  
raise_salary(121, 200);  -- same as raise_salary(121, 200, 50)  
END;
/

Example 9-21  Function Provides Default Parameter Value

DECLARE  
global PLS_INTEGER := 0;

FUNCTION f RETURN PLS_INTEGER IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('Inside f.');  
    global := global + 1;  
    RETURN global * 2;  
END f;

PROCEDURE p (  
x IN PLS_INTEGER := f()  
) IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE (  
    'Inside p. ' ||  
    ' global = ' || global ||  
    ', x = ' || x || ' .'  
    );  
    DBMS_OUTPUT.PUT_LINE('-----------------------');  
END p;
PROCEDURE pre_p IS
BEGIN
DBMS_OUTPUT.PUT LINE (  
'Before invoking p, global = ' || global || '.
);
DBMS_OUTPUT.PUT_LINE('Invoking p.');
END pre_p;
BEGIN
pre_p;
p();     -- default expression is evaluated
pre_p;
p(100);  -- default expression is not evaluated
pre_p;
p();     -- default expression is evaluated
END;
/

Result:
Before invoking p, global = 0.
Invoking p.
Inside f.
Inside p.   global = 1, x = 2.
----------------------------------
Before invoking p, global = 1.
Invoking p.
Inside p.   global = 1, x = 100.
----------------------------------
Before invoking p, global = 1.
Invoking p.
Inside f.
Inside p.   global = 2, x = 4.
----------------------------------

Example 9-22    Adding Subprogram Parameter Without Changing Existing Invocations

Create procedure:
CREATE OR REPLACE PROCEDURE print_name (  
  first VARCHAR2,  
  last VARCHAR2
) AUTHID DEFINER IS
BEGIN  
  DBMS_OUTPUT.PUT_LINE(first || ' ' || last);
END print_name;
/

Invoke procedure:
BEGIN
  print_name('John', 'Doe');
END;
/

Result:
John Doe
Add third parameter with default value:

```sql
CREATE OR REPLACE PROCEDURE print_name (  
    first VARCHAR2,  
    last VARCHAR2,  
    mi   VARCHAR2 := NULL
) AUTHID DEFINER IS
BEGIN  
    IF mi IS NULL THEN  
        DBMS_OUTPUT.PUT_LINE(first || ' ' || last);  
    ELSE  
        DBMS_OUTPUT.PUT_LINE(first || ' ' || mi || '. ' || last);  
    END IF;
END print_name;
/
```

Invoke procedure:

```sql
BEGIN  
    print_name('John', 'Doe');  
    print_name('John', 'Public', 'Q');  
END;
/
```

Result:

- John Doe
- John Q. Public

### 9.7.6 Positional, Named, and Mixed Notation for Actual Parameters

When invoking a subprogram, you can specify the actual parameters using either positional, named, or mixed notation. Table 9-3 summarizes and compares these notations.
Table 9-3  PL/SQL Actual Parameter Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Syntax</th>
<th>Optional parameters</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positional</td>
<td>Specify the actual parameters in the same order as the formal parameters are declared.</td>
<td>You can omit trailing optional parameters.</td>
<td>Specifying actual parameters in the wrong order can cause problems that are hard to detect, especially if the actual parameters are literals. Subprogram invocations must change if the formal parameter list changes, unless the list only acquires new trailing optional parameters (as in Example 9-22). Reduced code clarity and maintainability. Not recommended if the subprogram has a large number of parameters.</td>
<td></td>
</tr>
<tr>
<td>Named</td>
<td>Specify the actual parameters in any order, using this syntax: ( formal \rightarrow actual ) ( formal ) is the name of the formal parameter and ( actual ) is the actual parameter.</td>
<td>You can omit any optional parameters.</td>
<td>There is no wrong order for specifying actual parameters. Subprogram invocations must change only if the formal parameter list acquires new required parameters. Recommended when you invoke a subprogram defined or maintained by someone else.</td>
<td></td>
</tr>
<tr>
<td>Mixed</td>
<td>Start with positional notation, then use named notation for the remaining parameters. In the positional notation, you can omit trailing optional parameters; in the named notation, you can omit any optional parameters.</td>
<td>Convenient when you invoke a subprogram that has required parameters followed by optional parameters, and you must specify only a few of the optional parameters.</td>
<td>In the positional notation, the wrong order can cause problems that are hard to detect, especially if the actual parameters are literals. Changes to the formal parameter list might require changes in the positional notation.</td>
<td></td>
</tr>
</tbody>
</table>

In Example 9-23, the procedure invocations use different notations, but are equivalent.

In Example 9-24, the SQL SELECT statements invoke the PL/SQL function `compute_bonus`, using equivalent invocations with different notations.
Example 9-23  Equivalent Invocations with Different Notations in Anonymous Block

DECLARE
    emp_num    NUMBER(6) := 120;
    bonus      NUMBER(6) := 50;

PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER)
IS
    BEGIN
      UPDATE employees
      SET salary = salary + amount
      WHERE employee_id = emp_id;
    END raise_salary;

BEGIN
  -- Equivalent invocations:
  raise_salary(emp_num, bonus);          -- positional notation
  raise_salary(amount => bonus, emp_id => emp_num); -- named notation
  raise_salary(emp_id => emp_num, amount => bonus); -- named notation
  raise_salary(emp_num, amount => bonus);          -- mixed notation
END;
/

Example 9-24  Equivalent Invocations with Different Notations in SELECT Statements

CREATE OR REPLACE FUNCTION compute_bonus (emp_id NUMBER, bonus NUMBER)
  RETURN NUMBER
AUTHID DEFINER
IS
  emp_sal NUMBER;
BEGIN
  SELECT salary INTO emp_sal
  FROM employees
  WHERE employee_id = emp_id;

  RETURN emp_sal + bonus;
END compute_bonus;
/

SELECT compute_bonus(120, 50) FROM DUAL;          -- positional
SELECT compute_bonus(bonus => 50, emp_id => 120) FROM DUAL; -- named
SELECT compute_bonus(120, bonus => 50) FROM DUAL;          -- mixed

9.8 Subprogram Invocation Resolution

When the PL/SQL compiler encounters a subprogram invocation, it searches for a matching subprogram declaration—first in the current scope and then, if necessary, in successive enclosing scopes.

A declaration and invocation match if their subprogram names and parameter lists match. The parameter lists match if each required formal parameter in the declaration has a corresponding actual parameter in the invocation.
If the compiler finds no matching declaration for an invocation, then it generates a semantic error.

Figure 9-1 shows how the PL/SQL compiler resolves a subprogram invocation.

Figure 9-1   How PL/SQL Compiler Resolves Invocations

In Example 9-25, the function balance tries to invoke the enclosing procedure swap, using appropriate actual parameters. However, balance contains two nested procedures named swap, and neither has parameters of the same type as the enclosing procedure swap. Therefore, the invocation causes compilation error PLS-00306.

Example 9-25   Resolving PL/SQL Procedure Names

DECLARE
  PROCEDURE swap (  
    n1 NUMBER,  
    n2 NUMBER,  
  )  
BEGIN  
  n1 := n1 + n2;  
  n2 := n1 + n2;  
  n1 := n1 + n2;  
  n2 := n1 + n2;  
END;
FUNCTION balance (bal NUMBER) RETURN NUMBER IS
  x NUMBER := 10;
  PROCEDURE swap (d1 DATE, d2 DATE) IS
    BEGIN
      NULL;
    END;
  PROCEDURE swap (b1 BOOLEAN, b2 BOOLEAN) IS
    BEGIN
      NULL;
    END;
  BEGIN -- balance
    swap(num1, num2);
    RETURN x;
  END balance;
BEGIN -- enclosing procedure swap
  NULL;
END swap;
BEGIN -- anonymous block
  NULL;
END; -- anonymous block
/

Result:
  swap(num1, num2);
  *
ERROR at line 33:
ORA-06550: line 33, column 7:
PLS-00306: wrong number or types of arguments in call to 'SWAP'
ORA-06550: line 33, column 7:
PL/SQL: Statement ignored

9.9 Overloaded Subprograms

PL/SQL lets you overload nested subprograms, package subprograms, and type methods. You can use the same name for several different subprograms if their formal parameters differ in name, number, order, or data type family. (A data type family is a data type and its subtypes. For the data type families of predefined PL/SQL data types, see PL/SQL Predefined Data Types. For information about user-defined
PL/SQL subtypes, see "User-Defined PL/SQL Subtypes".) If formal parameters differ only in name, then you must use named notation to specify the corresponding actual parameters. (For information about named notation, see "Positional, Named, and Mixed Notation for Actual Parameters".)

Example 9-26 defines two subprograms with the same name, initialize. The procedures initialize different types of collections. Because the processing in the procedures is the same, it is logical to give them the same name.

You can put the two initialize procedures in the same block, subprogram, package, or type body. PL/SQL determines which procedure to invoke by checking their formal parameters. The version of initialize that PL/SQL uses depends on whether you invoke the procedure with a date_tab_typ or num_tab_typ parameter.

For an example of an overloaded procedure in a package, see Example 11-9.

Topics

- Formal Parameters that Differ Only in Numeric Data Type
- Subprograms that You Cannot Overload
- Subprogram Overload Errors

Example 9-26 Overloaded Subprogram

```plsql
DECLARE
    TYPE date_tab_typ IS TABLE OF DATE INDEX BY PLS_INTEGER;
    TYPE num_tab_typ  IS TABLE OF NUMBER INDEX BY PLS_INTEGER;

    hiredate_tab  date_tab_typ;
    sal_tab       num_tab_typ;

PROCEDURE initialize (tab OUT date_tab_typ, n INTEGER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Invoked first version');
        FOR i IN 1..n LOOP
            tab(i) := SYSDATE;
        END LOOP;
        END initialize;

PROCEDURE initialize (tab OUT num_tab_typ, n INTEGER) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Invoked second version');
        FOR i IN 1..n LOOP
            tab(i) := 0.0;
        END LOOP;
        END initialize;

BEGIN
    initialize(hiredate_tab, 50);
    initialize(sal_tab, 100);
END;
/

Result:

Invoked first version
Invoked second version
```
9.9.1 Formal Parameters that Differ Only in Numeric Data Type

You can overload subprograms if their formal parameters differ only in numeric data type. This technique is useful in writing mathematical application programming interfaces (APIs), because several versions of a function can use the same name, and each can accept a different numeric type. For example, a function that accepts `BINARY_FLOAT` might be faster, while a function that accepts `BINARY_DOUBLE` might be more precise.

To avoid problems or unexpected results when passing parameters to such overloaded subprograms:

- Ensure that the expected version of a subprogram is invoked for each set of expected parameters.
  
  For example, if you have overloaded functions that accept `BINARY_FLOAT` and `BINARY_DOUBLE`, which is invoked if you pass a `VARCHAR2` literal like '5.0'?

- Qualify numeric literals and use conversion functions to make clear what the intended parameter types are.

  For example, use literals such as `5.0f` (for `BINARY_FLOAT`), `5.0d` (for `BINARY_DOUBLE`), or conversion functions such as `TO_BINARY_FLOAT`, `TO_BINARY_DOUBLE`, and `TO_NUMBER`.

PL/SQL looks for matching numeric parameters in this order:

1. `PLS_INTEGER` (or `BINARY_INTEGER`, an identical data type)
2. `NUMBER`
3. `BINARY_FLOAT`
4. `BINARY_DOUBLE`

A `VARCHAR2` value can match a `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE` parameter.

PL/SQL uses the first overloaded subprogram that matches the supplied parameters. For example, the `SQRT` function takes a single parameter. There are overloaded versions that accept a `NUMBER`, a `BINARY_FLOAT`, or a `BINARY_DOUBLE` parameter. If you pass a `PLS_INTEGER` parameter, the first matching overload is the one with a `NUMBER` parameter.

The `SQRT` function that takes a `NUMBER` parameter is likely to be slowest. To use a faster version, use the `TO_BINARY_FLOAT` or `TO_BINARY_DOUBLE` function to convert the parameter to another data type before passing it to the `SQRT` function.

If PL/SQL must convert a parameter to another data type, it first tries to convert it to a higher data type. For example:

- The `ATAN2` function takes two parameters of the same type. If you pass parameters of different types—for example, one `PLS_INTEGER` and one `BINARY_FLOAT`—PL/SQL tries to find a match where both parameters use the higher type. In this case, that is the version of `ATAN2` that takes two `BINARY_FLOAT` parameters; the `PLS_INTEGER` parameter is converted upwards.

- A function takes two parameters of different types. One overloaded version takes a `PLS_INTEGER` and a `BINARY_FLOAT` parameter. Another overloaded version takes a `NUMBER` and a `BINARY_DOUBLE` parameter. If you invoke this function and pass two `NUMBER` parameters, PL/SQL first finds the overloaded version where the second
parameter is BINARY_FLOAT. Because this parameter is a closer match than the BINARY_DOUBLE parameter in the other overload, PL/SQL then looks downward and converts the first NUMBER parameter to PLS_INTEGER.

### 9.9.2 Subprograms that You Cannot Overload

You cannot overload these subprograms:

- Standalone subprograms
- Subprograms whose formal parameters differ only in mode; for example:
  
  ```sql
  PROCEDURE s (p IN VARCHAR2) IS ...  
  PROCEDURE s (p OUT VARCHAR2) IS ...  
  ```

- Subprograms whose formal parameters differ only in subtype; for example:
  
  ```sql
  PROCEDURE s (p INTEGER) IS ...  
  PROCEDURE s (p REAL) IS ...  
  ```

  INTEGER and REAL are subtypes of NUMBER, so they belong to the same data type family.

- Functions that differ only in return value data type, even if the data types are in different families; for example:
  
  ```sql
  FUNCTION f (p INTEGER) RETURN BOOLEAN IS ...  
  FUNCTION f (p INTEGER) RETURN INTEGER IS ...  
  ```

### 9.9.3 Subprogram Overload Errors

The PL/SQL compiler catches overload errors as soon as it determines that it cannot tell which subprogram was invoked. When subprograms have identical headings, the compiler catches the overload error when you try to compile the subprograms themselves (if they are nested) or when you try to compile the package specification that declares them. Otherwise, the compiler catches the error when you try to compile an ambiguous invocation of a subprogram.

When you try to compile the package specification in Example 9-27, which declares subprograms with identical headings, you get compile-time error PLS-00305.

Although the package specification in Example 9-28 violates the rule that you cannot overload subprograms whose formal parameters differ only in subtype, you can compile it without error.

However, when you try to compile an invocation of pkg2.s, as in Example 9-29, you get compile-time error PLS-00307.

Suppose that you correct the overload error in Example 9-28 by giving the formal parameters of the overloaded subprograms different names, as in Example 9-30.

Now you can compile an invocation of pkg2.s without error if you specify the actual parameter with named notation, as in Example 9-31. (If you specify the actual parameter with positional notation, as in Example 9-29, you still get compile-time error PLS-00307.)

The package specification in Example 9-32 violates no overload rules and compiles without error. However, you can still get compile-time error PLS-00307 when invoking its overloaded procedure, as in the second invocation in Example 9-33.
When trying to determine which subprogram was invoked, if the PL/SQL compiler implicitly converts one parameter to a matching type, then the compiler looks for other parameters that it can implicitly convert to matching types. If there is more than one match, then compile-time error PLS-00307 occurs, as in Example 9-34.

Example 9-27  Overload Error Causes Compile-Time Error

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER IS
  PROCEDURE s (p VARCHAR2);
  PROCEDURE s (p VARCHAR2);
END pkg1;
/
```

Example 9-28  Overload Error Compiles Successfully

```
CREATE OR REPLACE PACKAGE pkg2 AUTHID DEFINER IS
  SUBTYPE t1 IS VARCHAR2(10);
  SUBTYPE t2 IS VARCHAR2(10);
  PROCEDURE s (p t1);
  PROCEDURE s (p t2);
END pkg2;
/
```

Example 9-29  Invoking Subprogram in Example 9-28 Causes Compile-Time Error

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  a pkg2.t1 := 'a';
BEGIN
  pkg2.s(a);  -- Causes compile-time error PLS-00307
END p;
/
```

Example 9-30  Correcting Overload Error in Example 9-28

```
CREATE OR REPLACE PACKAGE pkg2 AUTHID DEFINER IS
  SUBTYPE t1 IS VARCHAR2(10);
  SUBTYPE t2 IS VARCHAR2(10);
  PROCEDURE s (p1 t1);
  PROCEDURE s (p2 t2);
END pkg2;
/
```

Example 9-31  Invoking Subprogram in Example 9-30

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  a pkg2.t1 := 'a';
BEGIN
  pkg2.s(p1=>'a');  -- Compiles without error
END p;
/
```

Example 9-32  Package Specification Without Overload Errors

```
CREATE OR REPLACE PACKAGE pkg3 AUTHID DEFINER IS
  PROCEDURE s (p1 VARCHAR2);
  PROCEDURE s (p1 VARCHAR2, p2 VARCHAR2 := 'p2');
END pkg3;
/
```

Chapter 9
Overloaded Subprograms
Example 9-33  Improper Invocation of Properly Overloaded Subprogram

CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
   a1 VARCHAR2(10) := 'a1';
   a2 VARCHAR2(10) := 'a2';
BEGIN
   pkg3.s(p1=>a1, p2=>a2);  -- Compiles without error
   pkg3.s(p1=>a1);          -- Causes compile-time error PLS-00307
END p;
/

Example 9-34  Implicit Conversion of Parameters Causes Overload Error

CREATE OR REPLACE PACKAGE pack1 AUTHID DEFINER AS
   PROCEDURE proc1 (a NUMBER, b VARCHAR2);
   PROCEDURE proc1 (a NUMBER, b NUMBER);
END;
/
CREATE OR REPLACE PACKAGE BODY pack1 AS
   PROCEDURE proc1 (a NUMBER, b VARCHAR2) IS BEGIN NULL; END;
   PROCEDURE proc1 (a NUMBER, b NUMBER) IS BEGIN NULL; END;
END;
/
BEGIN
   pack1.proc1(1,'2');    -- Compiles without error
   pack1.proc1(1,2);      -- Compiles without error
   pack1.proc1('1','2');  -- Causes compile-time error PLS-00307
   pack1.proc1('1',2);    -- Causes compile-time error PLS-00307
END;
/

9.10 Recursive Subprograms

A recursive subprogram invokes itself. Recursion is a powerful technique for simplifying an algorithm.

A recursive subprogram must have at least two execution paths—one leading to the recursive invocation and one leading to a terminating condition. Without the latter, recursion continues until PL/SQL runs out of memory and raises the predefined exception STORAGE_ERROR.

In Example 9-35, the function implements the following recursive definition of \( n \) factorial (\( n! \)), the product of all integers from 1 to \( n \):

\[
  n! = n \times (n - 1)!
\]

In Example 9-36, the function returns the \( n \)th Fibonacci number, which is the sum of the \( n \)-1st and \( n \)-2nd Fibonacci numbers. The first and second Fibonacci numbers are zero and one, respectively.

Note:

The function in Example 9-36 is a good candidate for result caching. For more information, see "Result-Cached Recursive Function".
Each recursive invocation of a subprogram creates an instance of each item that the subprogram declares and each SQL statement that it executes.

A recursive invocation inside a cursor FOR LOOP statement, or between an OPEN or OPEN FOR statement and a CLOSE statement, opens another cursor at each invocation, which might cause the number of open cursors to exceed the limit set by the database initialization parameter OPEN_CURSORS.

Example 9-35  Recursive Function Returns n Factorial (n!)

CREATE OR REPLACE FUNCTION factorial ( 
  n POSITIVE 
) RETURN POSITIVE 
AUTHID DEFINER 
IS 
BEGIN 
  IF n = 1 THEN 
    RETURN n; 
  ELSE 
    RETURN n * factorial(n-1); 
  END IF; 
END; 
/
BEGIN 
  FOR i IN 1..5 LOOP 
    DBMS_OUTPUT.PUT_LINE(i || '! = ' || factorial(i)); 
  END LOOP; 
END; 
/
Result:
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120

Example 9-36  Recursive Function Returns nth Fibonacci Number

CREATE OR REPLACE FUNCTION fibonacci ( 
  n PLS_INTEGER 
) RETURN PLS_INTEGER 
AUTHID DEFINER 
IS 
  fib_1 PLS_INTEGER := 0; 
  fib_2 PLS_INTEGER := 1; 
BEGIN 
  IF n = 1 THEN -- terminating condition 
    RETURN fib_1; 
  ELSIF n = 2 THEN -- terminating condition 
    RETURN fib_2; 
  ELSE 
    RETURN fibonacci(n-2) + fibonacci(n-1); -- recursive invocations 
  END IF; 
END; 
/
BEGIN 
  FOR i IN 1..10 LOOP 
    DBMS_OUTPUT.PUT(fibonacci(i)); 
    IF i < 10 THEN
DBMS_OUTPUT.PUT(', '); 
END IF;
END LOOP;

DBMS_OUTPUT.PUT_LINE(' ...');
END;
/

Result:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

9.11 Subprogram Side Effects

A subprogram has side effects if it changes anything except the values of its own local variables. For example, a subprogram that changes any of the following has side effects:

- Its own OUT or IN OUT parameter
- A global variable
- A public variable in a package
- A database table
- The database
- The external state (by invoking DBMS_OUTPUT or sending e-mail, for example)

Side effects can prevent the parallelization of a query, yield order-dependent (and therefore, indeterminate) results, or require that package state be maintained across user sessions.

Minimizing side effects is especially important when defining a result-cached function or a stored function for SQL statements to invoke.

See Also:

Oracle Database Development Guide for information about controlling side effects in PL/SQL functions invoked from SQL statements

9.12 PL/SQL Function Result Cache

When a PL/SQL function has the RESULT_CACHE option, its results are cached in the shared global area (SGA) so sessions connected to the same instance can reuse these results when available.

Oracle Database automatically detects all data sources (tables and views) that are queried while a result-cached function is running. If changes to any of these data sources are committed, the cached result becomes invalid across all instances. The best candidates for result-caching are functions that are invoked frequently but depend on information that changes infrequently or never.
9.12.1 Enabling Result-Caching for a Function

To make a function result-cached, include the `RESULT_CACHE` clause in the function declaration and definition. For syntax details, see "Function Declaration and Definition".

**Note:** For more information about configuring and managing the database server result cache, see Oracle Database Reference and Oracle Database Performance Tuning Guide.

In Example 9-37, the package `department_pkg` declares and then defines a result-cached function, `get_dept_info`, which returns a record of information about a given department. The function depends on the database tables `DEPARTMENTS` and `EMPLOYEES`.

You invoke the function `get_dept_info` as you invoke any function. For example, this invocation returns a record of information about department number 10:

```
department_pkg.get_dept_info(10);
```

This invocation returns only the name of department number 10:

```
department_pkg.get_dept_info(10).dept_name;
```

If the result for `get_dept_info(10)` is in the result cache, the result is returned from the cache; otherwise, the result is computed and added to the cache. Because `get_dept_info` depends on the `DEPARTMENTS` and `EMPLOYEES` tables, any committed change to `DEPARTMENTS` or `EMPLOYEES` invalidates all cached results for `get_dept_info`, relieving you of programming cache invalidation logic everywhere that `DEPARTMENTS` or `EMPLOYEES` might change.

**Example 9-37  Declaring and Defining Result-Cached Function**

```
CREATE OR REPLACE PACKAGE department_pkg AUTHID DEFINER IS

  TYPE dept_info_record IS RECORD (
    dept_name  departments.department_name%TYPE,
    mgr_name   employees.last_name%TYPE,
    dept_size  PLS_INTEGER
    );

  -- Function declaration
```

FUNCTION get_dept_info (dept_id NUMBER) 
    RETURN dept_info_record
       RESULT_CACHE;

END department_pkg;
/

CREATE OR REPLACE PACKAGE BODY department_pkg IS
   -- Function definition
   FUNCTION get_dept_info (dept_id NUMBER) 
       RETURN dept_info_record
       RESULT_CACHE
   IS
       rec  dept_info_record;
       BEGIN
       SELECT department_name INTO rec.dept_name
       FROM departments
       WHERE department_id = dept_id;

       SELECT e.last_name INTO rec.mgr_name
       FROM departments d, employees e
       WHERE d.department_id = dept_id
       AND d.manager_id = e.employee_id;

       SELECT COUNT(*) INTO rec.dept_size
       FROM EMPLOYEES
       WHERE department_id = dept_id;

       RETURN rec;
   END get_dept_info;
   END department_pkg;
/

9.12.2 Developing Applications with Result-Cached Functions

When developing an application that uses a result-cached function, make no assumptions about the number of times the body of the function will run for a given set of parameter values.

Some situations in which the body of a result-cached function runs are:

- The first time a session on this database instance invokes the function with these parameter values
- When the cached result for these parameter values is invalid
  When a change to any data source on which the function depends is committed, the cached result becomes invalid.
- When the cached results for these parameter values have aged out
  If the system needs memory, it might discard the oldest cached values.
- When the function bypasses the cache (see "Result Cache Bypass")

9.12.3 Requirements for Result-Cached Functions

A result-cached PL/SQL function is safe if it always produces the same output for any input that it would produce were it not marked with RESULT_CACHE. This safety is only guaranteed if these conditions are met:
• When the function is executed, it has no side effects.
  For information about side effects, see "Subprogram Side Effects".
• All tables that the function accesses are ordinary, non-SYS-owned permanent
tables in the same database as the function.
• The function’s result must be determined only by the vector of input actuals
  together with the committed content, at the current SCN, of the tables that it
  references.

It is recommended that a result-cached function also meet these criteria:
• It does not depend on session-specific settings.
  For more information, see "Making Result-Cached Functions Handle Session-
  Specific Settings".
• It does not depend on session-specific application contexts.
  For more information, see "Making Result-Cached Functions Handle Session-
  Specific Application Contexts".

For more information, see Oracle Database Performance Tuning Guide.

9.12.4 Examples of Result-Cached Functions

The best candidates for result-caching are functions that are invoked frequently but
depend on information that changes infrequently (as might be the case in the first
example). Result-caching avoids redundant computations in recursive functions.

Examples:
• Result-Cached Application Configuration Parameters
• Result-Cached Recursive Function

9.12.4.1 Result-Cached Application Configuration Parameters

Consider an application that has configuration parameters that can be set at either the
global level, the application level, or the role level. The application stores the
configuration information in these tables:

```sql
-- Global Configuration Settings
DROP TABLE global_config_params;
CREATE TABLE global_config_params
  (name VARCHAR2(20), -- parameter NAME
   val VARCHAR2(20), -- parameter VALUE
   PRIMARY KEY (name)));

-- Application-Level Configuration Settings
CREATE TABLE app_level_config_params
  (app_id VARCHAR2(20), -- application ID
   name VARCHAR2(20), -- parameter NAME
   val VARCHAR2(20), -- parameter VALUE
   PRIMARY KEY (app_id, name));

-- Role-Level Configuration Settings
CREATE TABLE role_level_config_params
  (role_id VARCHAR2(20), -- application (role) ID
   ...)
```
For each configuration parameter, the role-level setting overrides the application-level setting, which overrides the global setting. To determine which setting applies to a parameter, the application defines the PL/SQL function \texttt{get\_value}. Given a parameter name, application ID, and role ID, \texttt{get\_value} returns the setting that applies to the parameter.

The function \texttt{get\_value} is a good candidate for result-caching if it is invoked frequently and if the configuration information changes infrequently.

\textbf{Example 9-38} shows a possible definition for \texttt{get\_value}. Suppose that for one set of parameter values, the global setting determines the result of \texttt{get\_value}. While \texttt{get\_value} is running, the database detects that three tables are queried—\texttt{role\_level\_config\_params}, \texttt{app\_level\_config\_params}, and \texttt{global\_config\_params}. If a change to any of these three tables is committed, the cached result for this set of parameter values is invalidated and must be recomputed.

Now suppose that, for a second set of parameter values, the role-level setting determines the result of \texttt{get\_value}. While \texttt{get\_value} is running, the database detects that only the \texttt{role\_level\_config\_params} table is queried. If a change to \texttt{role\_level\_config\_params} is committed, the cached result for the second set of parameter values is invalidated; however, committed changes to \texttt{app\_level\_config\_params} or \texttt{global\_config\_params} do not affect the cached result.

\textbf{Example 9-38} \hspace{1em} Result-Cached Function Returns Configuration Parameter Setting

\begin{verbatim}
CREATE OR REPLACE FUNCTION get_value
    (p_param VARCHAR2,
     p_app_id  NUMBER,
     p_role_id NUMBER
    )
RETURN VARCHAR2
RESULT_CACHE
AUTHID DEFINER
IS
  answer VARCHAR2(20);
BEGIN
  -- Is parameter set at role level?
  BEGIN
    SELECT val INTO answer
    FROM role_level_config_params
    WHERE role_id = p_role_id
    AND name = p_param;
    RETURN answer; -- Found
  EXCEPTION
    WHEN no_data_found THEN
      NULL; -- Fall through to following code
  END;
  -- Is parameter set at application level?
  BEGIN
    SELECT val INTO answer
    FROM app_level_config_params
    WHERE app_id = p_app_id
    AND name = p_param;
    RETURN answer; -- Found
END;
\end{verbatim}
9.12.4.2 Result-Cached Recursive Function

A recursive function for finding the \( n \)th term of a Fibonacci series that mirrors the mathematical definition of the series might do many redundant computations. For example, to evaluate \( \text{fibonacci}(7) \), the function must compute \( \text{fibonacci}(6) \) and \( \text{fibonacci}(5) \). To compute \( \text{fibonacci}(6) \), the function must compute \( \text{fibonacci}(5) \) and \( \text{fibonacci}(4) \). Therefore, \( \text{fibonacci}(5) \) and several other terms are computed redundantly. Result-caching avoids these redundant computations.

\[ \text{fibonacci}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2) & \text{otherwise} \end{cases} \]

\( \text{fibonacci}(0) = 1 \)
\( \text{fibonacci}(1) = 1 \)

---

**Note:**

The maximum number of recursive invocations cached is 128.

---

9.12.5 Advanced Result-Cached Function Topics

**Topics**

- Rules for a Cache Hit
- Result Cache Bypass
- Making Result-Cached Functions Handle Session-Specific Settings
- Making Result-Cached Functions Handle Session-Specific Application Contexts
- Choosing Result-Caching Granularity
- Result Caches in Oracle RAC Environment
- Result Cache Management
- Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend
9.12.5.1 Rules for a Cache Hit

Each time a result-cached function is invoked with different parameter values, those parameters and their result are stored in the cache. Subsequently, when the same function is invoked with the same parameter values (that is, when there is a cache hit), the result is retrieved from the cache, instead of being recomputed.

The rules for parameter comparison for a cache hit differ from the rules for the PL/SQL "equal to" (=) operator, as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Cache Hit Rules</th>
<th>&quot;Equal To&quot; Operator Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL comparison</td>
<td>NULL equals NULL</td>
<td>NULL = NULL evaluates to NULL.</td>
</tr>
<tr>
<td>Non-null scalar</td>
<td>Non-null scalars are the same if and only if their values are identical; that is, if and only if their values have identical bit patterns on the given platform. For example, CHAR values 'AA' and 'AA ' are different. (This rule is stricter than the rule for the &quot;equal to&quot; operator.)</td>
<td>Non-null scalars can be equal even if their values do not have identical bit patterns on the given platform; for example, CHAR values 'AA' and 'AA ' are equal.</td>
</tr>
</tbody>
</table>

9.12.5.2 Result Cache Bypass

In some situations, the cache is bypassed. When the cache is bypassed:

- The function computes the result instead of retrieving it from the cache.
- The result that the function computes is not added to the cache.

Some examples of situations in which the cache is bypassed are:

- The cache is unavailable to all sessions.
  For example, the database administrator has disabled the use of the result cache during application patching (as in "Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend").
- A session is performing a DML statement on a table or view on which a result-cached function depends.
  The session bypasses the result cache for that function until the DML statement is completed—either committed or rolled back. If the statement is rolled back, the session resumes using the cache for that function.

Cache bypass ensures that:
- The user of each session sees his or her own uncommitted changes.
- The PL/SQL function result cache has only committed changes that are visible to all sessions, so that uncommitted changes in one session are not visible to other sessions.
9.12.5.3 Making Result-Cached Functions Handle Session-Specific Settings

If a function depends on settings that might vary from session to session (such as NLS_DATE_FORMAT and TIME_ZONE), make the function result-cached only if you can modify it to handle the various settings.

The function, get_hire_date, in Example 8–39 uses the TO_CHAR function to convert a DATE item to a VARCHAR item. The function get_hire_date does not specify a format mask, so the format mask defaults to the one that NLS_DATE_FORMAT specifies. If sessions that invoke get_hire_date have different NLS_DATE_FORMAT settings, cached results can have different formats. If a cached result computed by one session ages out, and another session recomputes it, the format might vary even for the same parameter value. If a session gets a cached result whose format differs from its own format, that result is probably incorrect.

Some possible solutions to this problem are:

- Change the return type of get_hire_date to DATE and have each session invoke the TO_CHAR function.
- If a common format is acceptable to all sessions, specify a format mask, removing the dependency on NLS_DATE_FORMAT. For example:

  \[\text{TO}\_\text{CHAR}(\text{date\_hired}, 'mm/dd/yy');\]

- Add a format mask parameter to get_hire_date. For example:

  \[
  \text{CREATE OR REPLACE FUNCTION get\_hire\_date (emp\_id \text{NUMBER}, \text{fmt} \text{VARCHAR})}
  \]
  \[
  \text{RETURN \text{VARCHAR}}
  \]
  \[
  \text{RESULT\_CACHE}
  \]
  \[
  \text{AUTHID DEFINER}
  \]
  \[
  \text{IS}
  \]
  \[
  \text{date\_hired DATE;}
  \]
  \[
  \text{BEGIN}
  \]
  \[
  \text{SELECT hire\_date INTO date\_hired}
  \]
  \[
  \text{FROM HR.EMPLOYEES}
  \]
  \[
  \text{WHERE EMPLOYEE\_ID = emp\_id;}
  \]
  \[
  \text{RETURN \text{TO}\_\text{CHAR}(date\_hired, fmt);}\]
  \[
  \text{END;}
  \]
  \[
  /\]

Example 9-39  Result-Cached Function Handles Session-Specific Settings

\[
\text{CREATE OR REPLACE FUNCTION get\_hire\_date (emp\_id \text{NUMBER})}
\]
\[
\text{RETURN \text{VARCHAR}}
\]
\[
\text{RESULT\_CACHE}
\]
\[
\text{AUTHID DEFINER}
\]
\[
\text{IS}
\]
\[
\text{date\_hired DATE;}
\]
\[
\text{BEGIN}
\]
\[
\text{SELECT hire\_date INTO date\_hired}
\]
\[
\text{FROM HR.EMPLOYEES}
\]
\[
\text{WHERE EMPLOYEE\_ID = emp\_id;}
\]
\[
\text{RETURN \text{TO}\_\text{CHAR}(date\_hired);}\]
\[
\text{END;}
\]
\[
/\]
9.12.5.4 Making Result-Cached Functions Handle Session-Specific Application Contexts

An **application context**, which can be either global or session-specific, is a set of attributes and their values. A PL/SQL function depends on session-specific application contexts if it does one or more of the following:

- Directly invokes the SQL function `SYS_CONTEXT`, which returns the value of a specified attribute in a specified context
- Indirectly invokes `SYS_CONTEXT` by using Virtual Private Database (VPD) mechanisms for fine-grained security

(For information about VPD, see *Oracle Database Security Guide*.)

The PL/SQL function result-caching feature does not automatically handle dependence on session-specific application contexts. If you must cache the results of a function that depends on session-specific application contexts, you must pass the application context to the function as a parameter. You can give the parameter a default value, so that not every user must specify it.

In **Example 9-40**, assume that a table, `config_tab`, has a VPD policy that translates this query:

```
SELECT value FROM config_tab WHERE name = param_name;
```

To this query:

```
SELECT value FROM config_tab 
WHERE name = param_name 
AND app_id = SYS_CONTEXT('Config', 'App_ID');
```

**Example 9-40  Result-Cached Function Handles Session-Specific Application Context**

```
CREATE OR REPLACE FUNCTION get_param_value ( 
   param_name VARCHAR, 
   appctx VARCHAR DEFAULT SYS_CONTEXT('Config', 'App_ID') 
) RETURN VARCHAR 
RESULT_CACHE 
AUTHID DEFINER
IS 
rec VARCHAR(2000); 
BEGIN 
   SELECT val INTO rec 
   FROM config_tab 
   WHERE name = param_name; 

   RETURN rec; 
END; 
/
```

9.12.5.5 Choosing Result-Caching Granularity

PL/SQL provides the function result cache, but you choose the caching granularity. To understand the concept of granularity, consider the `Product_Descriptions` table in the Order Entry (OE) sample schema:
The table has the name and description of each product in several languages. The unique key for each row is `PRODUCT_ID, LANGUAGE_ID`.

Suppose that you must define a function that takes a `PRODUCT_ID` and a `LANGUAGE_ID` and returns the associated `TRANSLATED_NAME`. You also want to cache the translated names. Some of the granularity choices for caching the names are:

- One name at a time (finer granularity)
- One language at a time (coarser granularity)

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finer</td>
<td>Each function result corresponds to one logical result. Stores only data that is needed at least once. Each data item ages out individually. Does not allow bulk loading optimizations.</td>
</tr>
<tr>
<td>Coarser</td>
<td>Each function result contains many logical subresults. Might store data that is never used. One aged-out data item ages out the whole set. Allows bulk loading optimizations.</td>
</tr>
</tbody>
</table>

In Example 9-41 and Example 9-42, the function `productName` takes a `PRODUCT_ID` and a `LANGUAGE_ID` and returns the associated `TRANSLATED_NAME`. Each version of `productName` caches translated names, but at a different granularity.

In Example 9-41, `get_product_name_1` is a result-cached function. Whenever `get_product_name_1` is invoked with a different `PRODUCT_ID` and `LANGUAGE_ID`, it caches the associated `TRANSLATED_NAME`. Each invocation of `get_product_name_1` adds at most one `TRANSLATED_NAME` to the cache.

In Example 9-42, `get_product_name_2` defines a result-cached function, `all_product_names`. Whenever `get_product_name_2` invokes `all_product_names` with a different `LANGUAGE_ID`, `all_product_names` caches every `TRANSLATED_NAME` associated with that `LANGUAGE_ID`. Each invocation of `all_product_names` adds every `TRANSLATED_NAME` of at most one `LANGUAGE_ID` to the cache.

**Example 9-41  Caching One Name at a Time (Finer Granularity)**

```sql
CREATE OR REPLACE FUNCTION get_product_name_1 (prod_id NUMBER,
lang_id VARCHAR2)
RETURN NVARCHAR2
RESULT_CACHE
AUTHID DEFINER
IS
    result_ VARCHAR2(50);
```
BEGIN
    SELECT translated_name INTO result_
    FROM OE.Product_Descriptions
    WHERE PRODUCT_ID = prod_id
    AND LANGUAGE_ID = lang_id;
RETURN result_;END;
/

Example 9-42  Caching Translated Names One Language at a Time (Coarser
Granularity)

CREATE OR REPLACE FUNCTION get_product_name_2 (prod_id NUMBER, lang_id VARCHAR2)
RETURN NVARCHAR2 AUTHID DEFINER IS
    TYPE product_names IS TABLE OF NVARCHAR2(50) INDEX BY PLS_INTEGER;
    FUNCTION all_product_names (lang_id VARCHAR2) RETURN product_names RESULT_CACHE IS
        all_names product_names;
    BEGIN
        FOR c IN (SELECT * FROM OE.Product_Descriptions WHERE LANGUAGE_ID = lang_id) LOOP
            all_names(c.PRODUCT_ID) := c.TRANSLATED_NAME;
        END LOOP;
RETURN all_names;
    END;
BEGIN
RETURN all_product_names(lang_id)(prod_id);
END;
/

9.12.5.6 Result Caches in Oracle RAC Environment

Cached results are stored in the system global area (SGA). In an Oracle RAC environment, each database instance manages its own local function result cache. However, the contents of the local result cache are accessible to sessions attached to other Oracle RAC instances. If a required result is missing from the result cache of the local instance, the result might be retrieved from the local cache of another instance, instead of being locally computed.

The access pattern and work load of an instance determine the set of results in its local cache; therefore, the local caches of different instances can have different sets of results.

Although each database instance might have its own set of cached results, the mechanisms for handling invalid results are Oracle RAC environment-wide. If results were invalidated only in the local instance’s result cache, other instances might use invalid results. For example, consider a result cache of item prices that are computed from data in database tables. If any of these database tables is updated in a way that affects the price of an item, the cached price of that item must be invalidated in every database instance in the Oracle RAC environment.
9.12.5.7 Result Cache Management

The PL/SQL function result cache shares its administrative and manageability infrastructure with the Result Cache.

The database administrator manages the server result cache by specifying the result cache initialization parameters.

The DBMS_RESULT_CACHE package allows you to administer the shared pool area part that is used by the SQL result cache and the PL/SQL function result cache.

Dynamic performance views provide information to monitor the server and client result caches.

See Also:

- Oracle Database Reference for more information about RESULT_CACHE_MAX_SIZE
- Oracle Database Reference for more information about RESULT_CACHE_MAX_RESULT
- Oracle Database Performance Tuning Guide for more information about result cache concepts
- Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_RESULT_CACHE package
- Oracle Database Reference for more information about V$RESULT_CACHE_STATISTICS
- Oracle Database Reference for more information about V$RESULT_CACHE_MEMORY
- Oracle Database Reference for more information about V$RESULT_CACHE_OBJECTS
- Oracle Database Concepts for more information about the Server Result Cache Infrastructure
- Oracle Database Reference for more information about V$RESULT_CACHE_DEPENDENCY

9.12.5.8 Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend

When you hot-patch a PL/SQL unit on which a result-cached function depends (directly or indirectly), the cached results associated with the result-cached function might not be automatically flushed in all cases.

For example, suppose that the result-cached function P1.foo() depends on the package subprogram P2.bar(). If a new version of the body of package P2 is loaded, the cached results associated with P1.foo() are not automatically flushed.

Therefore, this is the recommended procedure for hot-patching a PL/SQL unit:
Note:
To follow these steps, you must have the EXECUTE privilege on the package DBMS_RESULT_CACHE.

1. Put the result cache in bypass mode and flush existing results:
   ```plsql
   BEGIN
     DBMS_RESULT_CACHE.Bypass(TRUE);
     DBMS_RESULT_CACHE.Flush;
   END;
   /
   
   In an Oracle RAC environment, perform this step for each database instance.
   ```

2. Patch the PL/SQL code.

3. Resume using the result cache:
   ```plsql
   BEGIN
     DBMS_RESULT_CACHE.Bypass(FALSE);
   END;
   /
   
   In an Oracle RAC environment, perform this step for each database instance.
   ```

9.13 PL/SQL Functions that SQL Statements Can Invoke

To be invocable from SQL statements, a stored function (and any subprograms that it invokes) must obey the following purity rules, which are meant to control side effects:

- When invoked from a `SELECT` statement or a paralleled `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statement, the subprogram cannot modify any database tables.

- When invoked from an `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statement, the subprogram cannot query or modify any database tables modified by that statement.

If a function either queries or modifies a table, and a DML statement on that table invokes the function, then ORA-04091 (mutating-table error) occurs. There is one exception: ORA-04091 does not occur if a single-row `INSERT` statement that is not in a `FORALL` statement invokes the function in a `VALUES` clause.

- When invoked from a `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statement, the subprogram cannot execute any of the following SQL statements (unless `PRAGMA AUTONOMOUS_TRANSACTION` was specified):
  - Transaction control statements (such as `COMMIT`)
  - Session control statements (such as `SET ROLE`)
  - System control statements (such as `ALTER SYSTEM`)
  - Database definition language (DDL) statements (such as `CREATE`), which are committed automatically

(For the description of `PRAGMA AUTONOMOUS_TRANSACTION`, see "AUTONOMOUS_TRANSACTION Pragma".)
If any SQL statement in the execution part of the function violates a rule, then a runtime error occurs when that statement is parsed.

The fewer side effects a function has, the better it can be optimized in a `SELECT` statement, especially if the function is declared with the option `DETERMINISTIC` or `PARALLEL_ENABLE` (for descriptions of these options, see "DETERMINISTIC Clause" and "PARALLEL_ENABLE Clause").

See Also:

- Oracle Database Development Guide for information about restrictions on PL/SQL functions that SQL statements can invoke
- "Tune Function Invocations in Queries"

9.14 Invoker's Rights and Definer's Rights (AUTHID Property)

The `AUTHID` property of a stored PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at run time. The `AUTHID` property does not affect compilation, and has no meaning for units that have no code, such as collection types.

`AUTHID` property values are exposed in the static data dictionary view `*_PROCEDURES`. For units for which `AUTHID` has meaning, the view shows the value `CURRENT_USER` or `DEFINER`; for other units, the view shows `NULL`.

For stored PL/SQL units that you create or alter with the following statements, you can use the optional `AUTHID` clause to specify either `DEFINER` (the default, for backward compatibility) or `CURRENT_USER` (the preferred usage):

- "CREATE FUNCTION Statement"
- "CREATE PACKAGE Statement"
- "CREATE PROCEDURE Statement"
- "CREATE TYPE Statement"
- "ALTER TYPE Statement"

A unit whose `AUTHID` value is `CURRENT_USER` is called an invoker's rights unit, or IR unit. A unit whose `AUTHID` value is `DEFINER` (the default) is called a definer's rights unit, or DR unit. PL/SQL units and schema objects for which you cannot specify an `AUTHID` value behave like this:

<table>
<thead>
<tr>
<th>PL/SQL Unit or Schema Object</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anonymous block</td>
<td>IR unit</td>
</tr>
<tr>
<td>BEQUEATH CURRENT_USER view</td>
<td>Somewhat like an IR unit—see Oracle Database Security Guide.</td>
</tr>
<tr>
<td>BEQUEATH DEFINER view</td>
<td>DR unit</td>
</tr>
<tr>
<td>Trigger</td>
<td>DR unit</td>
</tr>
</tbody>
</table>
The **AUTHID** property of a unit determines whether the unit is IR or DR, and it affects both name resolution and privilege checking at run time:

- The context for name resolution is **CURRENT_SCHEMA**.
- The privileges checked are those of the **CURRENT_USER** and the enabled roles.

When a session starts, **CURRENT_SCHEMA** has the value of the schema owned by **SESSION_USER**, and **CURRENT_USER** has the same value as **SESSION_USER**. (To get the current value of **CURRENT_SCHEMA**, **CURRENT_USER**, or **SESSION_USER**, use the **SYS_CONTEXT** function, documented in *Oracle Database SQL Language Reference*.)

**CURRENT_SCHEMA** can be changed during the session with the SQL statement **ALTER SESSION SET CURRENT_SCHEMA**. **CURRENT_USER** cannot be changed programmatically, but it might change when a PL/SQL unit or a view is pushed onto, or popped from, the call stack.

**Note:**

Oracle recommends against issuing **ALTER SESSION SET CURRENT_SCHEMA** from in a stored PL/SQL unit.

During a server call, when a DR unit is pushed onto the call stack, the database stores the currently enabled roles and the current values of **CURRENT_USER** and **CURRENT_SCHEMA**. It then changes both **CURRENT_USER** and **CURRENT_SCHEMA** to the owner of the DR unit, and enables only the role **PUBLIC**. (The stored and new roles and values are not necessarily different.) When the DR unit is popped from the call stack, the database restores the stored roles and values. In contrast, when an IR unit is pushed onto, or popped from, the call stack, the values of **CURRENT_USER** and **CURRENT_SCHEMA**, and the currently enabled roles do not change (unless roles are granted to the IR unit itself—see “Granting Roles to PL/SQL Packages and Standalone Subprograms”).

For dynamic SQL statements issued by a PL/SQL unit, name resolution and privilege checking are done once, at run time. For static SQL statements, name resolution and privilege checking are done twice: first, when the PL/SQL unit is compiled, and then again at run time. At compile time, the **AUTHID** property has no effect—both DR and IR units are treated like DR units. At run time, however, the **AUTHID** property determines whether a unit is IR or DR, and the unit is treated accordingly.

Upon entry into an IR unit, the runtime system checks privileges before doing any initialization or running any code. If the unit owner has neither the **INHERIT PRIVILEGES** privilege on the invoker nor the **INHERIT ANY PRIVILEGES** privilege, then the runtime system raises error ORA-06598.
Note:

If the unit owner has the required privilege, then one of these statements granted it:

- `GRANT INHERIT PRIVILEGES ON current_user TO PUBLIC`
- `GRANT INHERIT PRIVILEGES ON current_user TO unit_owner`
- `GRANT INHERIT ANY PRIVILEGES TO unit_owner`

For information about the `GRANT` statement, see *Oracle Database SQL Language Reference*.

See Also:

- *Oracle Database Security Guide* for information about managing security for DR and IR units
- *Oracle Database Security Guide* for information about capturing privileges that are required to compile DR and IR program units

Topics

- Granting Roles to PL/SQL Packages and Standalone Subprograms
- IR Units Need Template Objects

9.14.1 Granting Roles to PL/SQL Packages and Standalone Subprograms

Using the SQL `GRANT` command, you can grant roles to PL/SQL packages and standalone subprograms. Roles granted to a PL/SQL unit do not affect compilation. They affect the privilege checking of SQL statements that the unit issues at run time: The unit runs with the privileges of both its own roles and any other currently enabled roles.

Typically, you grant roles to an IR unit, so that users with lower privileges than yours can run the unit with only the privileges needed to do so. You grant roles to a DR unit (whose invokers run it with all your privileges) only if the DR unit issues dynamic SQL, which is checked only at run time.

The basic syntax for granting roles to PL/SQL units is:

```
GRANT role [, role ]... TO unit [, unit ]...
```

For example, this command grants the roles `read` and `execute` to the function `scott.func` and the package `sys.pkg`:

```
GRANT read, execute TO FUNCTION scott.func, PACKAGE sys.pkg
```

For the complete syntax and semantics of the `GRANT` command, see *Oracle Database SQL Language Reference*. 
9.14.2 IR Units Need Template Objects

One user (that is, one schema) owns an IR unit and other users run it in their schemas. If the IR unit issues static SQL statements, then the schema objects that these statements affect must exist in the owner's schema at compile time (so that the compiler can resolve references) and in the invoker's schema at run time. The definitions of corresponding schema objects must match (for example, corresponding tables must have the same names and columns); otherwise, you get an error or unexpected results. However, the objects in the owner's schema need not contain data, because the compiler does not need it; therefore, they are called template objects.

9.14.3 Connected User Database Links in DR Units

If you include a connected user database link in a DR unit (definer's rights unit), then you must grant the user who will run the DR unit the INHERIT REMOTE PRIVILEGES privilege.

Granting the user this privilege enables the user to execute the DR unit; otherwise, the execution will fail with an ORA-25433: User does not have INHERIT REMOTE PRIVILEGES error. To include a connected user database link from within a definer's rights (DR) procedure, include `@database_link` in the procedure.

The following example shows how a DR unit can use a database link called `dblink` to access the `EMPLOYEE_ID` column of the `HR.EMPLOYEES` table:

Example 9-43    Database Link in a DR Unit

```sql
CREATE OR REPLACE PROCEDURE hr_remote_db_link 
AS
    v_employee_id VARCHAR(50);
BEGIN
    EXECUTE IMMEDIATE 'SELECT employee_id FROM employees@dblink' into v_employee_id;
    DBMS_OUTPUT.PUT_LINE('employee_id: ' || v_employee_id);
END ;
/
```

See Also:

Oracle Database Security Guide for more information about using the INHERIT REMOTE PRIVILEGES privilege, including a tutorial on how a DR unit can use a database link
9.15 External Subprograms

If a C procedure or Java method is stored in the database, you can publish it as an
external subprogram and then invoke it from PL/SQL.

To publish an external subprogram, define a stored PL/SQL subprogram with a call
specification. The call specification maps the name, parameter types, and return type
of the external subprogram to PL/SQL equivalents. Invoke the published external
subprogram by its PL/SQL name.

For example, suppose that this Java class, Adjuster, is stored in the database:

```java
import java.sql.*;
import oracle.jdbc.driver."
public class Adjuster {
    public static void raiseSalary (int empNo, float percent)
        throws SQLException {
        Connection conn = new OracleDriver().defaultConnection();
        String sql = "UPDATE employees SET salary = salary * ?
                WHERE employee_id = ?";
        try {
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setFloat(1, (1 + percent / 100));
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
            pstmt.close();
        } catch (SQLException e) {System.err.println(e.getMessage());}
    }
}
```

The Java class Adjuster has one method, raiseSalary, which raises the salary of a
specified employee by a specified percentage. Because raiseSalary is a void
method, you publish it as a PL/SQL procedure (rather than a function).

Example 9-44 publishes the stored Java method Adjuster.raiseSalary as a PL/SQL
standalone procedure, mapping the Java method name Adjuster.raiseSalary to the
PL/SQL procedure name raise_salary and the Java data types int and float to the
PL/SQL data type NUMBER. Then the anonymous block invokes raise_salary.

Example 9-45 publishes the stored Java method java.lang.Thread.sleep as a
PL/SQL standalone procedure, mapping the Java method name to the PL/SQL
procedure name java_sleep and the Java data type long to the PL/SQL data type
NUMBER. The PL/SQL standalone procedure sleep invokes java_sleep.

---

See Also:

Oracle Database Development Guide for more information about calling external programs

Example 9-44   PL/SQL Anonymous Block Invokes External Procedure

```
-- Publish Adjuster.raiseSalary as standalone PL/SQL procedure:

CREATE OR REPLACE PROCEDURE raise_salary (
```
empid NUMBER, pct NUMBER
) AS
  LANGUAGE JAVA NAME 'Adjuster.raiseSalary (int, float)'; -- call specification
/
BEGIN
  raise_salary(120, 10); -- invoke Adjuster.raiseSalary by PL/SQL name
END;
/

Example 9-45  PL/SQL Standalone Procedure Invokes External Procedure

-- Java call specification:
CREATE PROCEDURE java_sleep (
  milli_seconds IN NUMBER
) AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
/

CREATE OR REPLACE PROCEDURE sleep (
  milli_seconds IN NUMBER
) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
  java_sleep (milli_seconds);
  DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
END;
/
A trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs.

**Note:**
The database can detect only system-defined events. You cannot define your own events.

**Topics**
- Overview of Triggers
- Reasons to Use Triggers
- DML Triggers
- Correlation Names and Pseudorecords
- System Triggers
- Subprograms Invoked by Triggers
- Trigger Compilation, Invalidation, and Recompilation
- Exception Handling in Triggers
- Trigger Design Guidelines
- Trigger Restrictions
- Order in Which Triggers Fire
- Trigger Enabling and Disabling
- Trigger Changing and Debugging
- Triggers and Oracle Database Data Transfer Utilities
- Triggers for Publishing Events
- Views for Information About Triggers

### 10.1 Overview of Triggers

Like a stored procedure, a trigger is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it.

While a trigger is **enabled**, the database automatically invokes it—that is, the trigger **fires**—whenever its triggering event occurs. While a trigger is **disabled**, it does not fire.
You create a trigger with the `CREATE TRIGGER` statement. You specify the `triggering event` in terms of `triggering statements` and the item on which they act. The trigger is said to be `created on` or `defined on` the item, which is either a table, a view, a schema, or the database. You also specify the `timing point`, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects. By default, a trigger is created in the enabled state.

If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a **DML trigger**.

A **crossedition trigger** is a DML trigger for use only in edition-based redefinition.

If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a **system trigger**.

A **conditional trigger** is a DML or system trigger that has a `WHEN` clause that specifies a SQL condition that the database evaluates for each row that the triggering statement affects.

When a trigger fires, tables that the trigger references might be undergoing changes made by SQL statements in other users' transactions. SQL statements running in triggers follow the same rules that standalone SQL statements do. Specifically:

- Queries in the trigger see the current read-consistent materialized view of referenced tables and any data changed in the same transaction.
- Updates in the trigger wait for existing data locks to be released before proceeding.

An **INSTEAD OF trigger** is either:

- A DML trigger created on either a noneditioning view or a nested table column of a noneditioning view
- A system trigger defined on a `CREATE` statement

The database fires the **INSTEAD OF trigger** instead of running the triggering statement.

---

**Note:**

A trigger is often called by the name of its triggering statement (for example, `DELETE trigger` or `LOGON trigger`), the name of the item on which it is defined (for example, `DATABASE trigger` or `SCHEMA trigger`), or its timing point (for example, `BEFORE statement trigger` or `AFTER each row trigger`).
10.2 Reasons to Use Triggers

Triggers let you customize your database management system. For example, you can use triggers to:

- Automatically generate virtual column values
- Log events
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Enforce referential integrity when child and parent tables are on different nodes of a distributed database
- Publish information about database events, user events, and SQL statements to subscribing applications
- Prevent DML operations on a table after regular business hours
- Prevent invalid transactions
- Enforce complex business or referential integrity rules that you cannot define with constraints (see "How Triggers and Constraints Differ")

Caution:
Triggers are not reliable security mechanisms, because they are programmatic and easy to disable. For high-assurance security, use Oracle Database Vault, described in Oracle Database Vault Administrator's Guide.

How Triggers and Constraints Differ

Both triggers and constraints can constrain data input, but they differ significantly. A trigger always applies to new data only. For example, a trigger can prevent a DML statement from inserting a NULL value into a database column, but the column might contain NULL values that were inserted into the column before the trigger was defined or while the trigger was disabled.
A constraint can apply either to new data only (like a trigger) or to both new and existing data. Constraint behavior depends on constraint state, as explained in Oracle Database SQL Language Reference.

Constraints are easier to write and less error-prone than triggers that enforce the same rules. However, triggers can enforce some complex business rules that constraints cannot. Oracle strongly recommends that you use triggers to constrain data input only in these situations:

- To enforce referential integrity when child and parent tables are on different nodes of a distributed database
- To enforce complex business or referential integrity rules that you cannot define with constraints

See Also:

- Oracle Database Development Guide for information about using constraints to enforce business rules and prevent the entry of invalid information into tables
- “Triggers for Ensuring Referential Integrity” for information about using triggers and constraints to maintain referential integrity between parent and child tables

10.3 DML Triggers

A DML trigger is created on either a table or view, and its triggering event is composed of the DML statements DELETE, INSERT, and UPDATE.

To create a trigger that fires in response to a MERGE statement, create triggers on the INSERT and UPDATE statements to which the MERGE operation decomposes.

A DML trigger is either simple or compound.

A simple DML trigger fires at exactly one of these timing points:

- Before the triggering statement runs
  (The trigger is called a BEFORE statement trigger or statement-level BEFORE trigger.)
- After the triggering statement runs
  (The trigger is called an AFTER statement trigger or statement-level AFTER trigger.)
- Before each row that the triggering statement affects
  (The trigger is called a BEFORE each row trigger or row-level BEFORE trigger.)
- After each row that the triggering statement affects
  (The trigger is called an AFTER each row trigger or row-level AFTER trigger.)

A compound DML trigger created on a table or editioning view can fire at one, some, or all of the preceding timing points. Compound DML triggers help program an approach where you want the actions that you implement for the various timing points to share common data.
A simple or compound DML trigger that fires at row level can access the data in the row that it is processing. For details, see "Correlation Names and Pseudorecords".

An **INSTEAD OF DML trigger** is a DML trigger created on either a noneditioning view or a nested table column of a noneditioning view.

Except in an **INSTEAD OF** trigger, a triggering **UPDATE** statement can include a column list. With a column list, the trigger fires only when a specified column is updated. Without a column list, the trigger fires when any column of the associated table is updated.

**Topics**
- Conditional Predicates for Detecting Triggering DML Statement
- **INSTEAD OF** DML Triggers
- Compound DML Triggers
- Triggers for Ensuring Referential Integrity

### 10.3.1 Conditional Predicates for Detecting Triggering DML Statement

The triggering event of a DML trigger can be composed of multiple triggering statements. When one of them fires the trigger, the trigger can determine which one by using these **conditional predicates**.

#### Table 10-1 Conditional Predicates

<table>
<thead>
<tr>
<th>Conditional Predicate</th>
<th>TRUE if and only if:</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERTING</td>
<td>An INSERT statement fired the trigger.</td>
</tr>
<tr>
<td>UPDATING</td>
<td>An UPDATE statement fired the trigger.</td>
</tr>
<tr>
<td>UPDATING ('column')</td>
<td>An UPDATE statement that affected the specified column fired the trigger.</td>
</tr>
<tr>
<td>DELETING</td>
<td>A DELETE statement fired the trigger.</td>
</tr>
</tbody>
</table>

A conditional predicate can appear wherever a **BOOLEAN** expression can appear.

**Example 10-1 Trigger Uses Conditional Predicates to Detect Triggering Statement**

This example creates a DML trigger that uses conditional predicates to determine which of its four possible triggering statements fired it.

```sql
CREATE OR REPLACE TRIGGER t
BEFORE
    INSERT OR
    UPDATE OF salary, department_id OR
    DELETE
ON employees
BEGIN
    CASE
        WHEN INSERTING THEN
            DBMS_OUTPUT.PUT_LINE('Inserting');
        WHEN UPDATING('salary') THEN
            DBMS_OUTPUT.PUT_LINE('Updating salary');
        WHEN UPDATING('department_id') THEN
```
10.3.2 INSTEAD OF DML Triggers

An INSTEAD OF DML trigger is a DML trigger created on a noneditioning view, or on a nested table column of a noneditioning view. The database fires the INSTEAD OF trigger instead of running the triggering DML statement.

An INSTEAD OF trigger cannot be conditional.

An INSTEAD OF trigger is the only way to update a view that is not inherently updatable. Design the INSTEAD OF trigger to determine what operation was intended and do the appropriate DML operations on the underlying tables.

An INSTEAD OF trigger is always a row-level trigger. An INSTEAD OF trigger can read OLD and NEW values, but cannot change them.

An INSTEAD OF trigger with the NESTED TABLE clause fires only if the triggering statement operates on the elements of the specified nested table column of the view. The trigger fires for each modified nested table element.

See Also:

- Oracle Database SQL Language Reference for information about inherently updatable views
- "Compound DML Trigger Structure" for information about compound DML triggers with the INSTEAD OF EACH ROW section

Example 10-2    INSTEAD OF Trigger

This example creates the view oe.order_info to display information about customers and their orders. The view is not inherently updatable (because the primary key of the orders table, order_id, is not unique in the result set of the join view). The example creates an INSTEAD OF trigger to process INSERT statements directed to the view. The trigger inserts rows into the base tables of the view, customers and orders.

CREATE OR REPLACE VIEW order_info AS
  SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
        o.order_id, o.order_date, o.order_status
  FROM customers c, orders o
  WHERE c.customer_id = o.customer_id;

CREATE OR REPLACE TRIGGER order_info_insert
  INSTEAD OF INSERT ON order_info
DECLARE
duplicate_info EXCEPTION;
  PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
BEGIN

```
INSERT INTO customers
  (customer_id, cust_last_name, cust_first_name)
VALUES :
      :new.customer_id,
      :new.cust_last_name,
      :new.cust_first_name);

INSERT INTO orders (order_id, order_date, customer_id)
VALUES :
      :new.order_id,
      :new.order_date,
      :new.customer_id);

EXCEPTION
  WHEN duplicate_info THEN
    RAISE_APPLICATION_ERROR (
      num=> -20107,
      msg=> 'Duplicate customer or order ID');
END order_info_insert;
/

Query to show that row to be inserted does not exist:

SELECT COUNT(*) FROM order_info WHERE customer_id = 999;

Result:

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

1 row selected.

Insert row into view:

INSERT INTO order_info VALUES
  (999, 'Smith', 'John', 2500, TO_DATE('13-MAR-2001', 'DD-MON-YYYY'), 0);

Result:

1 row created.

Query to show that row has been inserted in view:

SELECT COUNT(*) FROM order_info WHERE customer_id = 999;

Result:

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1 row selected.

Query to show that row has been inserted in customers table:

SELECT COUNT(*) FROM customers WHERE customer_id = 999;

Result:

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

Chapter 10
DML Triggers
Query to show that row has been inserted in orders table:

SELECT COUNT(*) FROM orders WHERE customer_id = 999;

Result:

```
COUNT(*)
--------
  1
```

Example 10-3 INSTEAD OF Trigger on Nested Table Column of View

In this example, the view `dept_view` contains a nested table of employees, `emplist`, created by the `CAST` function (described in Oracle Database SQL Language Reference). To modify the `emplist` column, the example creates an INSTEAD OF trigger on the column.

```sql
-- Create type of nested table element:
CREATE OR REPLACE TYPE nte
AUTHID DEFINER IS
OBJECT {
    emp_id     NUMBER(6),
    lastname   VARCHAR2(25),
    job        VARCHAR2(10),
    sal        NUMBER(8,2)
};
/

-- Created type of nested table:
CREATE OR REPLACE TYPE emp_list_ IS
    TABLE OF nte;
/

-- Create view:
CREATE OR REPLACE VIEW dept_view AS
    SELECT d.department_id,
           d.department_name,
           CAST (MULTISET (SELECT e.employee_id, e.last_name, e.job_id,
                           e.salary
                       FROM employees e
                       WHERE e.department_id = d.department_id
                    )
        AS emp_list_
    ) emplist
    FROM departments d;
```
CREATE OR REPLACE TRIGGER dept_emplist_tr
   INSTEAD OF INSERT ON NESTED TABLE emplist OF dept_view
   REFERENCING NEW AS Employee
       PARENT AS Department
FOR EACH ROW
BEGIN
   -- Insert on nested table translates to insert on base table:
   INSERT INTO employees (
   employee_id,                      -- employee_id
   last_name,                    -- last_name
   email,                               -- hire_date
   hire_date,                        -- job_id
   job_id,                           -- salary
   salary,                           -- department_id
   department_id
   )
VALUES (
   :Employee.emp_id,                      -- employee_id
   :Employee.lastname,                    -- last_name
   :Employee.lastname || '@example.com',  -- email
   SYSDATE,                               -- hire_date
   :Employee.job,                         -- job_id
   :Employee.sal,                         -- salary
   :Department.department_id              -- department_id
   );
END;
/

Query view before inserting row into nested table:
SELECT emplist FROM dept_view WHERE department_id=10;

Result:

EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
----------------------------------------------
EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 4200))

1 row selected.

Query table before inserting row into nested table:
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE department_id = 10;

Result:

EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
200 Whalen                    AD_ASST          4200

1 row selected.

Insert a row into nested table:
INSERT INTO TABLE (  
SELECT d.emplist  
FROM dept_view d  
WHERE department_id = 10  
)  
VALUES (1001, 'Glenn', 'AC_MGR', 10000);

Query view after inserting row into nested table:
SELECT emplist FROM dept_view WHERE department_id=10;

Result (formatted to fit page):
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
--------------------------------------------------------------------------------
EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 4200),
            NTE(1001, 'Glenn', 'AC_MGR', 10000))
1 row selected.

Query table after inserting row into nested table:
SELECT employee_id, last_name, job_id, salary  
FROM employees  
WHERE department_id = 10;

Result:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>AD_ASST</td>
<td>4200</td>
</tr>
<tr>
<td>1001</td>
<td>Glenn</td>
<td>AC_MGR</td>
<td>10000</td>
</tr>
</tbody>
</table>

2 rows selected.

10.3.3 Compound DML Triggers

A compound DML trigger created on a table or editioning view can fire at multiple timing points. Each timing point section has its own executable part and optional exception-handling part, but all of these parts can access a common PL/SQL state. The common state is established when the triggering statement starts and is destroyed when the triggering statement completes, even when the triggering statement causes an error.

A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section.

A compound trigger can be conditional, but not autonomous.

Two common uses of compound triggers are:

- To accumulate rows destined for a second table so that you can periodically bulk-insert them
- To avoid the mutating-table error (ORA-04091)

Topics

- **Compound DML Trigger Structure**
10.3.3.1 Compound DML Trigger Structure

The optional declarative part of a compound trigger declares variables and subprograms that all of its timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run. The variables and subprograms exist for the duration of the triggering statement.

A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section. The syntax for creating the simplest compound DML trigger on a noneditioning view is:

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS BEGIN
    statement;
END INSTEAD OF EACH ROW;
```

A compound DML trigger created on a table or editioning view has at least one timing-point section in Table 10-2. If the trigger has multiple timing-point sections, they can be in any order, but no timing-point section can be repeated. If a timing-point section is absent, then nothing happens at its timing point.

### Table 10-2  Compound Trigger Timing-Point Sections

<table>
<thead>
<tr>
<th>Timing Point</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the triggering statement runs</td>
<td>BEFORE STATEMENT</td>
</tr>
<tr>
<td>After the triggering statement runs</td>
<td>AFTER STATEMENT</td>
</tr>
<tr>
<td>Before each row that the triggering statement affects</td>
<td>BEFORE EACH ROW</td>
</tr>
<tr>
<td>After each row that the triggering statement affects</td>
<td>AFTER EACH ROW</td>
</tr>
</tbody>
</table>

**See Also:**

"CREATE TRIGGER Statement" for more information about the syntax of compound triggers

A compound DML trigger does not have an initialization section, but the BEFORE STATEMENT section, which runs before any other timing-point section, can do any necessary initialization.

If a compound DML trigger has neither a BEFORE STATEMENT section nor an AFTER STATEMENT section, and its triggering statement affects no rows, then the trigger never fires.
10.3.3.2 Compound DML Trigger Restrictions

In addition to the "Trigger Restrictions"), compound DML triggers have these restrictions:

- **OLD, NEW, and PARENT** cannot appear in the declarative part, the **BEFORE STATEMENT** section, or the **AFTER STATEMENT** section.
- **Only the BEFORE EACH ROW section** can change the value of **NEW**.
- A timing-point section cannot handle exceptions raised in another timing-point section.
- If a timing-point section includes a **GOTO** statement, the target of the **GOTO** statement must be in the same timing-point section.

10.3.3.3 Performance Benefit of Compound DML Triggers

A compound DML trigger has a performance benefit when the triggering statement affects many rows.

For example, suppose that this statement triggers a compound DML trigger that has all four timing-point sections in Table 10-2:

```sql
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
  WHERE Source.c1 > 0
```

Although the **BEFORE EACH ROW** and **AFTER EACH ROW** sections of the trigger run for each row of **Source** whose column **c1** is greater than zero, the **BEFORE STATEMENT** section runs only before the **INSERT** statement runs and the **AFTER STATEMENT** section runs only after the **INSERT** statement runs.

A compound DML trigger has a greater performance benefit when it uses bulk SQL, described in "Bulk SQL and Bulk Binding".

10.3.3.4 Using Compound DML Triggers with Bulk Insertion

A compound DML trigger is useful for accumulating rows destined for a second table so that you can periodically bulk-insert them. To get the performance benefit from the compound trigger, you must specify **BULK COLLECT INTO** in the **FORALL** statement (otherwise, the **FORALL** statement does a single-row DML operation multiple times). For more information about using the **BULK COLLECT** clause with the **FORALL** statement, see "Using FORALL Statement and BULK COLLECT Clause Together".

**See Also:**

"FORALL Statement"

**Scenario:** You want to log every change to **hr.employees.salary** in a new table, **employee_salaries**. A single **UPDATE** statement updates many rows of the table **hr.employees**; therefore, bulk-inserting rows into **employee_salaries** is more efficient than inserting them individually.
Solution: Define a compound trigger on updates of the table `hr.employees`, as in Example 10-4. You do not need a BEFORE STATEMENT section to initialize `idx` or `salaries`, because they are state variables, which are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

Note:
To run Example 10-4, you must have the EXECUTE privilege on the package DBMS_LOCK.

Example 10-4  Compound Trigger Logs Changes to One Table in Another Table

```
CREATE TABLE employee_salaries (  
    employee_id NUMBER NOT NULL,  
    change_date DATE NOT NULL,  
    salary NUMBER(8,2) NOT NULL,  
    CONSTRAINT pk_employee_salaries PRIMARY KEY (employee_id, change_date),  
    CONSTRAINT fk_employee_salaries FOREIGN KEY (employee_id)  
        REFERENCES employees (employee_id)  
        ON DELETE CASCADE)  
/

CREATE OR REPLACE TRIGGER maintain_employee_salaries
    FOR UPDATE OF salary ON employees
    COMPOUND TRIGGER

    -- Declarative Part:
    -- Choose small threshold value to show how example works:
    threshhold CONSTANT SIMPLE_INTEGER := 7;

    TYPE salaries_t IS TABLE OF employee_salaries%ROWTYPE INDEX BY SIMPLE_INTEGER;
    salaries  salaries_t;
    idx       SIMPLE_INTEGER := 0;

    PROCEDURE flush_array IS  
    n CONSTANT SIMPLE_INTEGER := salaries.count();
    BEGIN
        FORALL j IN 1..n  
        INSERT INTO employee_salaries VALUES salaries(j);
        salaries.delete();
        idx := 0;
        DBMS_OUTPUT.PUT_LINE('Flushed ' || n || ' rows');
    END flush_array;

    -- AFTER EACH ROW Section:

    AFTER EACH ROW IS
    BEGIN
        idx := idx + 1;
        salaries(idx).employee_id := :NEW.employee_id;
        salaries(idx).change_date := SYSTIMESTAMP;
        salaries(idx).salary := :NEW.salary;
        IF idx >= threshhold THEN
            flush_array();
        END IF;
    END AFTER EACH ROW;

    -- AFTER STATEMENT Section:
```
AFTER STATEMENT IS
BEGIN
   flush_array();
   END AFTER STATEMENT;
END maintain_employee_salaries;
/

Increase salary of every employee in department 50 by 10%:

UPDATE employees
   SET salary = salary * 1.1
   WHERE department_id = 50
/

Result:
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 3 rows
45 rows updated.

Wait two seconds:

BEGIN
   DBMS_LOCK.SLEEP(2);
END;
/

Increase salary of every employee in department 50 by 5%:

UPDATE employees
   SET salary = salary * 1.05
   WHERE department_id = 50
/

Result:
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 7 rows
Flushed 3 rows
45 rows updated.

See changes to employees table reflected in employee_salaries table:

SELECT employee_id, count(*) c
   FROM employee_salaries
   GROUP BY employee_id
/

Result:
10.3.3.5 Using Compound DML Triggers to Avoid Mutating-Table Error

A compound DML trigger is useful for avoiding the mutating-table error (ORA-04091) explained in "Mutating-Table Restriction".

**Scenario:** A business rule states that an employee’s salary increase must not exceed 10% of the average salary for the employee’s department. This rule must be enforced by a trigger.

**Solution:** Define a compound trigger on updates of the table `hr.employees`, as in Example 10-5. The state variables are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

**Example 10-5  Compound Trigger Avoids Mutating-Table Error**

```sql
CREATE OR REPLACE TRIGGER Check_Employee_Salary_Raise
FOR UPDATE OF Salary ON Employees
COMPOUND TRIGGER

Ten_Percent                 CONSTANT NUMBER := 0.1;
TYPE Salaries_t             IS TABLE OF Employees.Salary%TYPE;
Avg_Salaries                Salaries_t;
TYPE Department_IDs_t       IS TABLE OF Employees.Department_ID%TYPE;
Department_IDs              Department_IDs_t;

-- Declare collection type and variable:
TYPE Department_Salaries_t  IS TABLE OF Employees.Salary%TYPE
INDEX BY VARCHAR2(80);
Department_Avg_Salaries     Department_Salaries_t;

BEFORE STATEMENT IS
BEGIN
SELECT AVG(e.Salary), NVL(e.Department_ID, -1)
BULK COLLECT INTO Avg_Salaries, Department_IDs
FROM Employees e
GROUP BY e.Department_ID;
FOR j IN 1..Department_IDs.COUNT() LOOP
    Department_Avg_Salaries(Department_IDs(j)) := Avg_Salaries(j);
END LOOP;
END BEFORE STATEMENT;

AFTER EACH ROW IS
BEGIN
    IF :NEW.Salary - :Old.Salary > Ten_Percent*Department_Avg_Salaries(:NEW.Department_ID)
    THEN
        Raise_Application_Error(-20000, 'Raise too big');
    END IF;
```

Chapter 10
DML Triggers
10.3.4 Triggers for Ensuring Referential Integrity

You can use triggers and constraints to maintain referential integrity between parent and child tables, as Table 10-3 shows. (For more information about constraints, see Oracle Database SQL Language Reference.)

Table 10-3  Constraints and Triggers for Ensuring Referential Integrity

<table>
<thead>
<tr>
<th>Table</th>
<th>Constraint to Declare on Table</th>
<th>Triggers to Create on Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>PRIMARY KEY or UNIQUE</td>
<td>One or more triggers that ensure that when PRIMARY KEY or UNIQUE values are updated or deleted, the desired action (RESTRICT, CASCADE, or SET NULL) occurs on corresponding FOREIGN KEY values. No action is required for inserts into the parent table, because no dependent foreign keys exist.</td>
</tr>
<tr>
<td>Child</td>
<td>FOREIGN KEY, if parent and child are in the same database. (The database does not support declarative referential constraints between tables on different nodes of a distributed database.) Disable this foreign key constraint to prevent the corresponding PRIMARY KEY or UNIQUE constraint from being dropped (except explicitly with the CASCADE option).</td>
<td>One trigger that ensures that values inserted or updated in the FOREIGN KEY correspond to PRIMARY KEY or UNIQUE values in the parent table.</td>
</tr>
</tbody>
</table>

Topics

- Foreign Key Trigger for Child Table
- UPDATE and DELETE RESTRICT Trigger for Parent Table
- UPDATE and DELETE SET NULL Trigger for Parent Table
- DELETE CASCADE Trigger for Parent Table
- UPDATE CASCADE Trigger for Parent Table
- Triggers for Complex Constraint Checking
- Triggers for Complex Security Authorizations
- Triggers for Transparent Event Logging
- Triggers for Deriving Column Values
- Triggers for Building Complex Updatable Views
- Triggers for Fine-Grained Access Control
Several triggers include statements that lock rows (SELECT FOR UPDATE). This operation is necessary to maintain concurrency while the rows are being processed.

These examples are not meant to be used exactly as written. They are provided to assist you in designing your own triggers.

10.3.4.1 Foreign Key Trigger for Child Table

The trigger in Example 10-6 ensures that before an INSERT or UPDATE statement affects a foreign key value, the corresponding value exists in the parent key. The exception ORA-04091 (mutating-table error) allows the trigger emp_dept_check to be used with the UPDATE_SET_DEFAULT and UPDATE_CASCADE triggers. This exception is unnecessary if the trigger emp_dept_check is used alone.

Example 10-6  Foreign Key Trigger for Child Table

CREATE OR REPLACE TRIGGER emp_dept_check
    BEFORE INSERT OR UPDATE OF Deptno ON emp
    FOR EACH ROW WHEN (NEW.Deptno IS NOT NULL)

    -- Before row is inserted or DEPTNO is updated in emp table,
    -- fire this trigger to verify that new foreign key value (DEPTNO)
    -- is present in dept table.

    DECLARE
        Dummy INTEGER;  -- Use for cursor fetch
        Invalid_department EXCEPTION;
        Valid_department EXCEPTION;
        Mutating_table EXCEPTION;
        PRAGMA EXCEPTION_INIT (Invalid_department, -4093);
        PRAGMA EXCEPTION_INIT (Valid_department, -4092);
        PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

        -- Cursor used to verify parent key value exists.
        -- If present, lock parent key’s row so it cannot be deleted
-- by another transaction until this transaction is
-- committed or rolled back.

CURSOR Dummy_cursor (Dn NUMBER) IS
  SELECT Deptno FROM dept
  WHERE Deptno = Dn
  FOR UPDATE OF Deptno;
BEGIN
  OPEN Dummy_cursor (:NEW.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- Verify parent key.
  -- If not found, raise user-specified error code and message.
  -- If found, close cursor before allowing triggering statement to complete:
  IF Dummy_cursor%NOTFOUND THEN
    RAISE Invalid_department;
  ELSE
    RAISE Valid_department;
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN Invalid_department THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20000, 'Invalid Department'
      || ' Number' || TO_CHAR(:NEW.deptno));
  WHEN Valid_department THEN
    CLOSE Dummy_cursor;
  WHEN Mutating_table THEN
    NULL;
END;
/

10.3.4.2 UPDATE and DELETE RESTRICT Trigger for Parent Table

The trigger in Example 10-7 enforces the UPDATE and DELETE RESTRICT referential action on the primary key of the dept table.

Caution:
The trigger in Example 10-7 does not work with self-referential tables (tables with both the primary/unique key and the foreign key). Also, this trigger does not allow triggers to cycle (such as when A fires B, which fires A).

Example 10-7  UPDATE and DELETE RESTRICT Trigger for Parent Table

CREATE OR REPLACE TRIGGER dept_restrict
  BEFORE DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

-- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
-- check for dependent foreign key values in emp;
-- if any are found, roll back.

DECLARE
  Dummy         INTEGER;  -- Use for cursor fetch
  employees_present  EXCEPTION;
  ...
employees_not_present  EXCEPTION;
PRAGMA EXCEPTION_INIT (employees_present, -4094);
PRAGMA EXCEPTION_INIT (employees_not_present, -4095);

-- Cursor used to check for dependent foreign key values.
CURSOR Dummy_cursor (Dn NUMBER) IS
  SELECT Deptno FROM emp WHERE Deptno = Dn;
BEGIN
  OPEN Dummy_cursor (:OLD.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- If dependent foreign key is found, raise user-specified
  -- error code and message. If not found, close cursor
  -- before allowing triggering statement to complete.
  IF Dummy_cursor%FOUND THEN
    RAISE employees_present;     -- Dependent rows exist
  ELSE
    RAISE employees_not_present; -- No dependent rows exist
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN employees_present THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20001, 'Employees Present in'
                         || ' Department ' || TO_CHAR(:OLD.DEPTNO));
  WHEN employees_not_present THEN
    CLOSE Dummy_cursor;
END;

10.3.4.3 UPDATE and DELETE SET NULL Trigger for Parent Table

The trigger in Example 10-8 enforces the UPDATE and DELETE SET NULL referential action on the primary key of the dept table.

Example 10-8    UPDATE and DELETE SET NULL Trigger for Parent Table

CREATE OR REPLACE TRIGGER dept_set_null
AFTER DELETE OR UPDATE OF Deptno ON dept
FOR EACH ROW
  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- set all corresponding dependent foreign key values in emp to NULL:
BEGIN
  IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
    UPDATE emp SET emp.Deptno = NULL
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/

10.3.4.4 DELETE CASCADE Trigger for Parent Table

The trigger in Example 10-9 enforces the DELETE CASCADE referential action on the primary key of the dept table.

Example 10-9    DELETE CASCADE Trigger for Parent Table

CREATE OR REPLACE TRIGGER dept_delete CASCADE
AFTER DELETE OR UPDATE OF Deptno ON dept
FOR EACH ROW
  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- set all corresponding dependent foreign key values in emp to NULL:
BEGIN
  IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
    UPDATE emp SET emp.Deptno = NULL
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
Example 10-9  DELETE CASCADE Trigger for Parent Table

CREATE OR REPLACE TRIGGER dept_del_cascade
AFTER DELETE ON dept
FOR EACH ROW

-- Before row is deleted from dept,
-- delete all rows from emp table whose DEPTNO is same as
-- DEPTNO being deleted from dept table:

BEGIN
DELETE FROM emp
WHERE emp.Deptno = :OLD.Deptno;
END;
/

10.3.4.5 UPDATE CASCADE Trigger for Parent Table

The triggers in Example 10-10 ensure that if a department number is updated in the dept table, then this change is propagated to dependent foreign keys in the emp table.

Example 10-10  UPDATE CASCADE Trigger for Parent Table

-- Generate sequence number to be used as flag
-- for determining if update occurred on column:

CREATE SEQUENCE Update_sequence
INCREMENT BY 1 MAXVALUE 5000 CYCLE;

CREATE OR REPLACE PACKAGE Integritypackage AUTHID DEFINER AS
  Updateseq NUMBER;
END Integritypackage;
/
CREATE OR REPLACE PACKAGE BODY Integritypackage AS
  END Integritypackage;
/
-- Create flag col:

ALTER TABLE emp ADD Update_id NUMBER;
CREATE OR REPLACE TRIGGER dept_cascade1
    BEFORE UPDATE OF Deptno ON dept
DECLARE
    -- Before updating dept table (this is a statement trigger),
    -- generate sequence number
    -- & assign it to public variable UPDATESEQ of
    -- user-defined package named INTEGRITYPACKAGE:
BEGIN
    Integritypackage.Updateseq := Update_sequence.NEXTVAL;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade2
    AFTER DELETE OR UPDATE OF Deptno ON dept
    FOR EACH ROW
BEGIN
    IF UPDATING THEN
        UPDATE emp
        SET Deptno = :NEW.Deptno,
            Update_id = Integritypackage.Updateseq   --from 1st
        WHERE emp.Deptno = :OLD.Deptno
            AND Update_id IS NULL;
    END IF;
    IF DELETING THEN
        -- After row is deleted from dept,
        -- delete all rows from emp table whose DEPTNO is same as
        -- DEPTNO being deleted from dept table:
        DELETE FROM emp
        WHERE emp.Deptno = :OLD.Deptno;
    END IF;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade3
    AFTER UPDATE OF Deptno ON dept
BEGIN UPDATE emp
    SET Update_id = NULL
    WHERE Update_id = Integritypackage.Updateseq;
END;
/

10.3.4.6 Triggers for Complex Constraint Checking

Triggers can enforce integrity rules other than referential integrity. The trigger in Example 10-11 does a complex check before allowing the triggering statement to run.
**Note:**

Example 10-11 needs this data structure:

```sql
CREATE TABLE Salgrade {
  Grade       NUMBER,
  Losal       NUMBER,
  Hisal       NUMBER,
  Job_classification VARCHAR2(9);
}
```

Example 10-11  Trigger Checks Complex Constraints

```sql
CREATE OR REPLACE TRIGGER salary_check
  BEFORE INSERT OR UPDATE OF Sal, Job ON Emp
  FOR EACH ROW
DECLARE
  Minsal NUMBER;
  Maxsal NUMBER;
  Salary_out_of_range EXCEPTION;
  PRAGMA EXCEPTION_INIT (Salary_out_of_range, -4096);
BEGIN
  /* Retrieve minimum & maximum salary for employee's new job classification
     from SALGRADE table into MINSAL and MAXSAL: */
  SELECT Losal, Hisal INTO Minsal, Maxsal
  FROM Salgrade
  WHERE Job_classification = :NEW.Job;

  /* If employee's new salary is less than or greater than
     job classification's limits, raise exception. Exception message is returned and pending INSERT or UPDATE statement
     that fired the trigger is rolled back: */
  IF (:NEW.Sal < Minsal OR :NEW.Sal > Maxsal) THEN
    RAISE Salary_out_of_range;
  END IF;
EXCEPTION
  WHEN Salary_out_of_range THEN
    Raise_application_error (-20300, 'Salary '|| TO_CHAR(:NEW.Sal) ||' out of range for ' || 'job classification ' ||:NEW.Job ||' for employee ' || :NEW.Ename);
  WHEN NO_DATA_FOUND THEN
    Raise_application_error(-20322, 'Invalid Job Classification');
END;
/
```

10.3.4.7 Triggers for Complex Security Authorizations

Triggers are commonly used to enforce complex security authorizations for table data. Use triggers only to enforce complex security authorizations that you cannot define using the database security features provided with the database. For example, use a
trigger to prohibit updates to the employee table during weekends and nonworking hours.

When using a trigger to enforce a complex security authorization, it is best to use a BEFORE statement trigger. Using a BEFORE statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to run, so that no wasted work is done by an unauthorized statement.
- The security check is done only for the triggering statement, not for each row affected by the triggering statement.

The trigger in Example 10-12 enforces security by raising exceptions when anyone tries to update the table employees during weekends or nonworking hours.

### See Also:

*Oracle Database Security Guide* for detailed information about database security features

---

**Example 10-12  Trigger Enforces Security Authorizations**

```sql
CREATE OR REPLACE TRIGGER Employee_permit_changes
BEFORE INSERT OR DELETE OR UPDATE ON employees
DECLARE
  Dummy             INTEGER;
  Not_on_weekends   EXCEPTION;
  Nonworking_hours  EXCEPTION;
  PRAGMA EXCEPTION_INIT (Not_on_weekends, -4097);
  PRAGMA EXCEPTION_INIT (Nonworking_hours, -4099);
BEGIN
  -- Check for weekends:
  IF (TO_CHAR(Sysdate, 'DAY') = 'SAT' OR
      TO_CHAR(Sysdate, 'DAY') = 'SUN') THEN
    RAISE Not_on_weekends;
  END IF;

  -- Check for work hours (8am to 6pm):
  IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
      TO_CHAR(Sysdate, 'HH24') > 18) THEN
    RAISE Nonworking_hours;
  END IF;

EXCEPTION
  WHEN Not_on_weekends THEN
    Raise_application_error(-20324,'Might not change ' ||
      'employee table during the weekend');
  WHEN Nonworking_hours THEN
    Raise_application_error(-20326,'Might not change ' ||
      'emp table during Nonworking hours');
END;
/
```
10.3.4.8 Triggers for Transparent Event Logging

Triggers are very useful when you want to transparently do a related change in the database following certain events.

The `REORDER` trigger example shows a trigger that reorders parts as necessary when certain conditions are met. (In other words, a triggering statement is entered, and the `PARTS_ON_HAND` value is less than the `REORDER_POINT` value.)

10.3.4.9 Triggers for Deriving Column Values

Triggers can derive column values automatically, based upon a value provided by an `INSERT` or `UPDATE` statement. This type of trigger is useful to force values in specific columns that depend on the values of other columns in the same row. `BEFORE` row triggers are necessary to complete this type of operation for these reasons:

- The dependent values must be derived before the `INSERT` or `UPDATE` occurs, so that the triggering statement can use the derived values.
- The trigger must fire for each row affected by the triggering `INSERT` or `UPDATE` statement.

The trigger in Example 10-13 derives new column values for a table whenever a row is inserted or updated.

**Note:**

Example 10-13 needs this change to this data structure:

```
ALTER TABLE Emp ADD(
    Uppername   VARCHAR2(20),
    Soundexname VARCHAR2(20));
```

**Example 10-13  Trigger Derives New Column Values**

```
CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp
/* Before updating the ENAME field, derive the values for
the UPPERNAME and SOUNDEXNAME fields. Restrict users
from updating these fields directly: */
FOR EACH ROW
BEGIN
    :NEW.Uppername := UPPER(:NEW.Ename);
    :NEW.Soundexname := SOUNDEX(:NEW.Ename);
END;
/
```

10.3.4.10 Triggers for Building Complex Updatable Views

Views are an excellent mechanism to provide logical windows over table data. However, when the view query gets complex, the system implicitly cannot translate the DML on the view into those on the underlying tables. **INSTEAD** of triggers help solve
this problem. These triggers can be defined over views, and they fire instead of the actual DML.

Consider a library system where books are arranged by title. The library consists of a collection of book type objects:

```sql
CREATE OR REPLACE TYPE Book_t AS OBJECT {
    Booknum    NUMBER,
    Title      VARCHAR2(20),
    Author     VARCHAR2(20),
    Available  CHAR(1)
};
/
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;
/
```

The table `Book_table` is created and populated like this:

```sql
DROP TABLE Book_table;
CREATE TABLE Book_table (
    Booknum    NUMBER,
    Section    VARCHAR2(20),
    Title      VARCHAR2(20),
    Author     VARCHAR2(20),
    Available  CHAR(1)
);
INSERT INTO Book_table (
    Booknum, Section, Title, Author, Available
) VALUES (
    121001, 'Classic', 'Iliad', 'Homer', 'Y'
);;
INSERT INTO Book_table (
    Booknum, Section, Title, Author, Available
) VALUES (
    121002, 'Novel', 'Gone with the Wind', 'Mitchell M', 'N'
);
SELECT * FROM Book_table ORDER BY Booknum;
```

Result:

<table>
<thead>
<tr>
<th>BOOKNUM</th>
<th>SECTION</th>
<th>TITLE</th>
<th>AUTHOR</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>121001</td>
<td>Classic</td>
<td>Iliad</td>
<td>Homer</td>
<td>Y</td>
</tr>
<tr>
<td>121002</td>
<td>Novel</td>
<td>Gone with the Wind</td>
<td>Mitchell M</td>
<td>N</td>
</tr>
</tbody>
</table>

2 rows selected.

The table `Library_table` is created and populated like this:

```sql
DROP TABLE Library_table;
CREATE TABLE Library_table (Section VARCHAR2(20));
INSERT INTO Library_table (Section)
VALUES ('Novel');
INSERT INTO Library_table (Section)
VALUES ('Classic');
```
You can define a complex view over the tables `Book_table` and `Library_table` to create a logical view of the library with sections and a collection of books in each section:

```
CREATE OR REPLACE VIEW Library_view AS
    SELECT i.Section, CAST (MULTISET {
        SELECT b.Booknum, b.Title, b.Author, b.Available
        FROM Book_table b
        WHERE b.Section = i.Section
    } AS Book_list_t) BOOKLIST
    FROM Library_table i;
```

(For information about the `CAST` function, see Oracle Database SQL Language Reference.)

Make `Library_view` updatable by defining an `INSTEAD` of trigger on it:

```
CREATE OR REPLACE TRIGGER Library_trigger
    INSTEAD OF INSERT ON Library_view
    FOR EACH ROW
    DECLARE
        Bookvar Book_t;
        i INTEGER;
    BEGIN
        INSERT INTO Library_table
            VALUES (:NEW.Section);
        FOR i IN 1..:NEW.Booklist.COUNT LOOP
            Bookvar := :NEW.Booklist(i);
            INSERT INTO Book_table
                VALUES (Booknum, Section, Title, Author, Available);
        END LOOP;
    END;
/
```

Insert a new row into `Library_view`:

```
INSERT INTO Library_view (Section, Booklist)
VALUES (  
    'History',
    Chapter 10
);```
See the effect on Library_view:

```sql
SELECT * FROM Library_view ORDER BY Section;
```

Result:

```
SECTION
--------------------
BOOKLIST(BOOKNUM, TITLE, AUTHOR, AVAILABLE)
--------------------------------------------------------------------
Classic    BOOK_LIST_T(BOOK_T(121001, 'Iliad', 'Homer', 'Y'))
History    BOOK_LIST_T(BOOK_T(121330, 'Alexander', 'Mirth', 'Y'))
Novel      BOOK_LIST_T(BOOK_T(121002, 'Gone with the Wind', 'Mitchell M', 'N'))
```

3 rows selected.

See the effect on Book_table:

```sql
SELECT * FROM Book_table ORDER BY Booknum;
```

Result:

```
BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
 121001 Classic              Iliad                Homer                Y
 121002 Novel                Gone with the Wind   Mitchell M           N
 121330 History              Alexander            Mirth                Y
```

3 rows selected.

See the effect on Library_table:

```sql
SELECT * FROM Library_table ORDER BY Section;
```

Result:

```
SECTION
--------------------
Classic              History              Novel
```

3 rows selected.

Similarly, you can also define triggers on the nested table booklist to handle modification of the nested table element.

**10.3.4.11 Triggers for Fine-Grained Access Control**

You can use LOGON triggers to run the package associated with an application context. An application context captures session-related information about the user who is
logging in to the database. From there, your application can control how much access this user has, based on his or her session information.

**Note:**

If you have very specific logon requirements, such as preventing users from logging in from outside the firewall or after work hours, consider using Oracle Database Vault instead of LOGON triggers. With Oracle Database Vault, you can create custom rules to strictly control user access.

**See Also:**

- *Oracle Database Security Guide* for information about creating a LOGON trigger to run a database session application context package
- *Oracle Database Vault Administrator’s Guide* for information about Oracle Database Vault

### 10.4 Correlation Names and Pseudorecords

**Note:**

This topic applies only to triggers that fire at row level. That is:

- Row-level simple DML triggers
- Compound DML triggers with row-level timing point sections

A trigger that fires at row level can access the data in the row that it is processing by using **correlation names**. The default correlation names are OLD, NEW, and PARENT. To change the correlation names, use the **REFERENCING clause** of the CREATE TRIGGER statement (see "referencing_clause ::=").

If the trigger is created on a nested table, then OLD and NEW refer to the current row of the nested table, and PARENT refers to the current row of the parent table. If the trigger is created on a table or view, then OLD and NEW refer to the current row of the table or view, and PARENT is undefined.

OLD, NEW, and PARENT are also called **pseudorecords**, because they have record structure, but are allowed in fewer contexts than records are. The structure of a pseudorecord is `table_name%ROWTYPE`, where `table_name` is the name of the table on which the trigger is created (for OLD and NEW) or the name of the parent table (for PARENT).

In the **trigger_body** of a simple trigger or the **tps_body** of a compound trigger, a correlation name is a placeholder for a bind variable. Reference the field of a pseudorecord with this syntax:
In the WHEN clause of a conditional trigger, a correlation name is not a placeholder for a bind variable. Therefore, omit the colon in the preceding syntax.

Table 10-4 shows the values of OLD and NEW fields for the row that the triggering statement is processing.

Table 10-4  OLD and NEW Pseudorecord Field Values

<table>
<thead>
<tr>
<th>Triggering Statement</th>
<th>OLD.field Value</th>
<th>NEW.field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>NULL</td>
<td>Post-insert value</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Pre-update value</td>
<td>Post-update value</td>
</tr>
<tr>
<td>DELETE</td>
<td>Pre-delete value</td>
<td>NULL</td>
</tr>
</tbody>
</table>

The restrictions on pseudorecords are:

- A pseudorecord cannot appear in a record-level operation.
  For example, the trigger cannot include this statement:
  \[ :NEW := NULL; \]

- A pseudorecord cannot be an actual subprogram parameter.
  (A pseudorecord field can be an actual subprogram parameter.)

- The trigger cannot change OLD field values.
  Trying to do so raises ORA-04085.

- If the triggering statement is DELETE, then the trigger cannot change NEW field values.
  Trying to do so raises ORA-04084.

- An AFTER trigger cannot change NEW field values, because the triggering statement runs before the trigger fires.
  Trying to do so raises ORA-04084.

A BEFORE trigger can change NEW field values before a triggering INSERT or UPDATE statement puts them in the table.

If a statement triggers both a BEFORE trigger and an AFTER trigger, and the BEFORE trigger changes a NEW field value, then the AFTER trigger "sees" that change.

Example 10-14  Trigger Logs Changes to EMPLOYEES.SALARY

This example creates a log table and a trigger that inserts a row in the log table after any UPDATE statement affects the SALARY column of the EMPLOYEES table, and then updates EMPLOYEES.SALARY and shows the log table.

Create log table:

```sql
CREATE TABLE Emp_log (  Emp_id NUMBER,  Log_date DATE,  New_salary NUMBER,  )
```
Create trigger that inserts row in log table after EMPLOYEES.SALARY is updated:

```sql
CREATE OR REPLACE TRIGGER log_salary_increase
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (:NEW.employee_id, SYSDATE, :NEW.salary, 'New Salary');
END;
/
```

Update EMPLOYEES.SALARY:

```sql
UPDATE employees
SET salary = salary + 1000.0
WHERE Department_id = 20;
```

Result:

2 rows updated.

Show log table:

```sql
SELECT * FROM Emp_log;
```

Result:

```
EMP_ID LOG_DATE  NEW_SALARY ACTION
---------- --------- ---------- --------------------
201 28-APR-10      13650 New Salary
202 28-APR-10       6300 New Salary
```

2 rows selected.

Example 10-15  Conditional Trigger Prints Salary Change Information

This example creates a conditional trigger that prints salary change information whenever a DELETE, INSERT, or UPDATE statement affects the EMPLOYEES table—unless that information is about the President. The database evaluates the WHEN condition for each affected row. If the WHEN condition is TRUE for an affected row, then the trigger fires for that row before the triggering statement runs. If the WHEN condition is not TRUE for an affected row, then trigger does not fire for that row, but the triggering statement still runs.

```sql
CREATE OR REPLACE TRIGGER print_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
WHEN (NEW.job_id <> 'AD_PRES')  -- do not print information about President
DECLARE
    sal_diff  NUMBER;
BEGIN
    sal_diff  := :NEW.salary  - :OLD.salary;
    DBMS_OUTPUT.PUT(:NEW.last_name || ': ');
    DBMS_OUTPUT.PUT('Old salary = ' || :OLD.salary || ', ');
    DBMS_OUTPUT.PUT('New salary = ' || :NEW.salary || ', ');
END;
```
DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);
END;
/

Query:

```
SELECT last_name, department_id, salary, job_id
FROM employees
WHERE department_id IN (10, 20, 90)
ORDER BY department_id, last_name;
```

Result:

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>4200</td>
<td>AD_ASST</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>6000</td>
<td>MK_REP</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>13000</td>
<td>MK_MAN</td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td>17000</td>
<td>AD_VP</td>
</tr>
<tr>
<td><strong>King</strong></td>
<td><strong>90</strong></td>
<td><strong>24000</strong></td>
<td><strong>AD_PRES</strong></td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>17000</td>
<td>AD_VP</td>
</tr>
</tbody>
</table>

6 rows selected.

Triggering statement:

```
UPDATE employees
SET salary = salary * 1.05
WHERE department_id IN (10, 20, 90);
```

Result:

- Whalen: Old salary = 4200, New salary = 4410, Difference: 210
- Hartstein: Old salary = 13000, New salary = 13650, Difference: 650
- Fay: Old salary = 6000, New salary = 6300, Difference: 300
- Kochhar: Old salary = 17000, New salary = 17850, Difference: 850
- De Haan: Old salary = 17000, New salary = 17850, Difference: 850

6 rows updated.

Query:

```
SELECT salary FROM employees WHERE job_id = 'AD_PRES';
```

Result:

<table>
<thead>
<tr>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>25200</td>
</tr>
</tbody>
</table>

1 row selected.

**Example 10-16 Trigger Modifies CLOB Columns**

This example creates an **UPDATE** trigger that modifies **CLOB** columns.

For information about **TO_CLOB** and other conversion functions, see *Oracle Database SQL Language Reference*.

```
DROP TABLE tab1;
CREATE TABLE tab1 (c1 CLOB);
```
CREATE OR REPLACE TRIGGER trg1
    BEFORE UPDATE ON tab1
    FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Old value of CLOB column: ''||:OLD.c1);
    DBMS_OUTPUT.PUT_LINE('Proposed new value of CLOB column: '||:NEW.c1);

    :NEW.c1 := :NEW.c1 || TO_CLOB('<hr><p>Standard footer paragraph.'));

    DBMS_OUTPUT.PUT_LINE('Final value of CLOB column: '||:NEW.c1);
END;
/

SET SERVEROUTPUT ON;
UPDATE tab1 SET c1 = '<h1>Different Document Fragment</h1><p>Different text.';
SELECT * FROM tab1;

Example 10-17  Trigger with REFERENCING Clause

This example creates a table with the same name as a correlation name, new, and then creates a trigger on that table. To avoid conflict between the table name and the correlation name, the trigger references the correlation name as Newest.

CREATE TABLE new (
    field1  NUMBER,
    field2  VARCHAR2(20)
);

CREATE OR REPLACE TRIGGER Print_salary_changes
    BEFORE UPDATE ON new
    REFERENCING new AS Newest
    FOR EACH ROW
BEGIN
    :Newest.Field2 := TO_CHAR (:newest.field1);
END;
/

10.4.1 OBJECT_VALUE Pseudocolumn

A DML trigger on an object table can reference the SQL pseudocolumn OBJECT_VALUE, which returns system-generated names for the columns of the object table. The trigger can also invoke a PL/SQL subprogram that has a formal IN parameter whose data type is OBJECT_VALUE.

See Also:

- Oracle Database SQL Language Reference for more information about OBJECT_VALUE
- Oracle Database SQL Language Reference for general information about pseudocolumns
Example 10-18 creates object table `tbl`, table `tbl_history` for logging updates to `tbl`, and trigger `Tbl_Trg`. The trigger runs for each row of `tbl` that is affected by a DML statement, causing the old and new values of the object `t` in `tbl` to be written in `tbl_history`. The old and new values are `:OLD.OBJECT_VALUE` and `:NEW.OBJECT_VALUE`.

All values of column `n` were increased by 1. The value of `m` remains 0.

**Example 10-18  Trigger References OBJECT.VALUE Pseudocolumn**

Create, populate, and show object table:

```sql
CREATE OR REPLACE TYPE t AUTHID DEFINER AS OBJECT (n NUMBER, m NUMBER) /
CREATE TABLE tbl OF t /
BEGIN
    FOR j IN 1..5 LOOP
        INSERT INTO tbl VALUES (t(j, 0));
    END LOOP;
END;
/
SELECT * FROM tbl ORDER BY n;
```

Result:

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

5 rows selected.

Create history table and trigger:

```sql
CREATE TABLE tbl_history ( d DATE, old_obj t, new_obj t) /
CREATE OR REPLACE TRIGGER Tbl_Trg
    AFTER UPDATE ON tbl
    FOR EACH ROW
BEGIN
    INSERT INTO tbl_history (d, old_obj, new_obj)
    VALUES (SYSDATE, :OLD.OBJECT_VALUE, :NEW.OBJECT_VALUE);
END Tbl_Trg;
/
```

Update object table:

```sql
UPDATE tbl SET tbl.n = tbl.n+1 /
```

Result:

5 rows updated.

Show old and new values:
BEGIN
  FOR j IN (SELECT d, old_obj, new_obj FROM tbl_history) LOOP
    DBMS_OUTPUT.PUT_LINE ({
      'j.d ||
      ' -- old: ' || j.old_obj.n || ' ' || j.old_obj.m ||
      ' -- new: ' || j.new_obj.n || ' ' || j.new_obj.m
    });
  END LOOP;
END;
/

Result:
28-APR-10 -- old: 1 0 -- new: 2 0
28-APR-10 -- old: 2 0 -- new: 3 0
28-APR-10 -- old: 3 0 -- new: 4 0
28-APR-10 -- old: 4 0 -- new: 5 0
28-APR-10 -- old: 5 0 -- new: 6 0

10.5 System Triggers

A system trigger is created on either a schema or the database.

Its triggering event is composed of either DDL statements (listed in "ddl_event") or database operation statements (listed in "database_event").

A system trigger fires at exactly one of these timing points:

- Before the triggering statement runs
  (The trigger is called a BEFORE statement trigger or statement-level BEFORE trigger.)
- After the triggering statement runs
  (The trigger is called a AFTER statement trigger or statement-level AFTER trigger.)
- Instead of the triggering CREATE statement
  (The trigger is called an INSTEAD OF CREATE trigger.)

Topics

- SCHEMA Triggers
- DATABASE Triggers
- INSTEAD OF CREATE Triggers

10.5.1 SCHEMA Triggers

A schema trigger is created on a schema and fires whenever the user who owns it is the current user and initiates the triggering event.

Suppose that both user1 and user2 own schema triggers, and user1 invokes a DR unit owned by user2. Inside the DR unit, user2 is the current user. Therefore, if the DR unit initiates the triggering event of a schema trigger that user2 owns, then that trigger fires. However, if the DR unit initiates the triggering event of a schema trigger that user1 owns, then that trigger does not fire.

Example 10-19 creates a BEFORE statement trigger on the sample schema HR. When a user connected as HR tries to drop a database object, the database fires the trigger before dropping the object.
Example 10-19  BEFORE Statement Trigger on Sample Schema HR

CREATE OR REPLACE TRIGGER drop_trigger
BEFORE DROP ON hr.SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (  
    num => -20000,  
    msg => 'Cannot drop object');
END;
/

10.5.2 DATABASE Triggers

A DATABASE trigger is created on the database and fires whenever any database user initiates the triggering event.

Example 10-20 shows the basic syntax for a trigger to log errors. This trigger fires after an unsuccessful statement execution, such as unsuccessful logon.

Note:

An AFTER SERVERERROR trigger fires only if Oracle relational database management system (RDBMS) determines that it is safe to fire error triggers. For more information about AFTER SERVERERROR triggers, see CREATE TRIGGER Statement.

The trigger in Example 10-21 runs the procedure check_user after a user logs onto the database.

Example 10-20  AFTER Statement Trigger on Database

CREATE TRIGGER log_errors
AFTER SERVERERROR ON DATABASE
BEGIN
  IF (IS_SERVERERROR (1017)) THEN
    NULL;  -- (substitute code that processes logon error)
  ELSE
    NULL;  -- (substitute code that logs error code)
  END IF;
END;
/

Example 10-21  Trigger Monitors Logons

CREATE OR REPLACE TRIGGER check_user
AFTER LOGON ON DATABASE
BEGIN
  check_user;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR
    (-20000, 'Unexpected error: '|| DBMS_Utility.Format_Error_Stack);
END;
/
10.5.3 INSTEAD OF CREATE Triggers

An INSTEAD OF CREATE trigger is a SCHEMA trigger whose triggering event is a CREATE statement. The database fires the trigger instead of executing its triggering statement.

Example 10-22 shows the basic syntax for an INSTEAD OF CREATE trigger on the current schema. This trigger fires when the owner of the current schema issues a CREATE statement in the current schema.

Example 10-22  INSTEAD OF CREATE Trigger on Schema

```sql
CREATE OR REPLACE TRIGGER t
  INSTEAD OF CREATE ON SCHEMA
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE T (n NUMBER, m NUMBER)';
END;
/
```

10.6 Subprograms Invoked by Triggers

Triggers can invoke subprograms written in PL/SQL, C, and Java. The trigger in Example 10-4 invokes a PL/SQL subprogram. The trigger in Example 10-23 invokes a Java subprogram.

A subprogram invoked by a trigger cannot run transaction control statements, because the subprogram runs in the context of the trigger body.

If a trigger invokes an invoker rights (IR) subprogram, then the user who created the trigger, not the user who ran the triggering statement, is considered to be the current user. For information about IR subprograms, see "Invoker's Rights and Definer's Rights (AUTHID Property)".

If a trigger invokes a remote subprogram, and a time stamp or signature mismatch is found during execution of the trigger, then the remote subprogram does not run and the trigger is invalidated.

Example 10-23  Trigger Invokes Java Subprogram

```sql
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS
  LANGUAGE Java
  name 'thjvTriggers.beforeDelete (oracle.jdbc.NUMBER, oracle.jdbc.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
  CALL Before_delete (:OLD.Id, :OLD.Ename)
/
```

The corresponding Java file is thjvTriggers.java:

```java
import java.sql.*
import java.io.*
import oracle.jdbc.*
import oracle.oracore.*
public class thjvTriggers
{
  public static void
  beforeDelete (NUMBER old_id, CHAR old_name)
  Throws SQLException, CoreException
```
10.7 Trigger Compilation, Invalidation, and Recompilation

The `CREATE TRIGGER` statement compiles the trigger and stores its code in the database. If a compilation error occurs, the trigger is still created, but its triggering statement fails, except in these cases:

- The trigger was created in the disabled state.
- The triggering event is `AFTER STARTUP` on `DATABASE`.
- The triggering event is either `AFTER LOGON` on `DATABASE` or `AFTER LOGON` on `SCHEMA`, and someone logs on as `SYSTEM`.

To see trigger compilation errors, either use the `SHOW ERRORS` command in SQL*Plus or Enterprise Manager, or query the static data dictionary view `*ERRORS` (described in Oracle Database Reference).

If a trigger does not compile successfully, then its exception handler cannot run. For an example, see "Remote Exception Handling".

If a trigger references another object, such as a subprogram or package, and that object is modified or dropped, then the trigger becomes invalid. The next time the triggering event occurs, the compiler tries to revalidate the trigger (for details, see Oracle Database Development Guide).

**Note:** Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.

To recompile a trigger manually, use the `ALTER TRIGGER` statement, described in "ALTER TRIGGER Statement".

10.8 Exception Handling in Triggers

In most cases, if a trigger runs a statement that raises an exception, and the exception is not handled by an exception handler, then the database rolls back the effects of both the trigger and its triggering statement.

In the following cases, the database rolls back only the effects of the trigger, not the effects of the triggering statement (and logs the error in trace files and the alert log):

- The triggering event is either `AFTER STARTUP` on `DATABASE` or `BEFORE SHUTDOWN` on `DATABASE`.
The triggering event is **AFTER LOGON ON DATABASE** and the user has the **ADMINISTER DATABASE TRIGGER** privilege.

The triggering event is **AFTER LOGON ON SCHEMA** and the user either owns the schema or has the **ALTER ANY TRIGGER** privilege.

In the case of a compound DML trigger, the database rolls back only the effects of the triggering statement, not the effects of the trigger. However, variables declared in the trigger are re-initialized, and any values computed before the triggering statement was rolled back are lost.

**Note:**
Triggers that enforce complex security authorizations or constraints typically raise user-defined exceptions, which are explained in "User-Defined Exceptions".

**See Also:**
- **PL/SQL Error Handling**, for general information about exception handling
- **Remote Exception Handling**

A trigger that accesses a remote database can do remote exception handling only if the remote database is available. If the remote database is unavailable when the local database must compile the trigger, then the local database cannot validate the statement that accesses the remote database, and the compilation fails. If the trigger cannot be compiled, then its exception handler cannot run.

The trigger in **Example 10-24** has an **INSERT** statement that accesses a remote database. The trigger also has an exception handler. However, if the remote database is unavailable when the local database tries to compile the trigger, then the compilation fails and the exception handler cannot run.

**Example 10-25** shows the workaround for the problem in **Example 10-24**: Put the remote **INSERT** statement and exception handler in a stored subprogram and have the trigger invoke the stored subprogram. The subprogram is stored in the local database in compiled form, with a validated statement for accessing the remote database. Therefore, when the remote **INSERT** statement fails because the remote database is unavailable, the exception handler in the subprogram can handle it.

**Example 10-24  Trigger Cannot Handle Exception if Remote Database is Unavailable**

```sql
CREATE OR REPLACE TRIGGER employees_tr
    AFTER INSERT ON employees
    FOR EACH ROW
BEGIN
    -- When remote database is unavailable, compilation fails here:
    INSERT INTO employees@remote (employee_id, first_name, last_name, email, hire_date, job_id)
    VALUES (]
```
99, 'Jane', 'Doe', 'jane.doe@example.com', SYSDATE, 'ST_MAN' 
);
EXCEPTION
WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (99, SYSDATE, NULL, 'Could not insert');
    RAISE;
END;
/

Example 10-25  Workaround for Example 10-24

CREATE OR REPLACE PROCEDURE insert_row_proc AUTHID CURRENT_USER AS
no_remote_db EXCEPTION;  -- declare exception
PRAGMA EXCEPTION_INIT (no_remote_db, -20000);
-- assign error code to exception
BEGIN
    INSERT INTO employees@remote (
        employee_id, first_name, last_name, email, hire_date, job_id
    )
    VALUES (99, 'Jane', 'Doe', 'jane.doe@example.com', SYSDATE, 'ST_MAN' 
    );
EXCEPTION
WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (99, SYSDATE, NULL, 'Could not insert row.');
    RAISE_APPLICATION_ERROR (-20000, 'Remote database is unavailable.');
END;
/

CREATE OR REPLACE TRIGGER employees_tr
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    insert_row_proc;
END;
/

10.9 Trigger Design Guidelines

- Use triggers to ensure that whenever a specific event occurs, any necessary actions are done (regardless of which user or application issues the triggering statement).
  
  For example, use a trigger to ensure that whenever anyone updates a table, its log file is updated.

- Do not create triggers that duplicate database features.
  
  For example, do not create a trigger to reject invalid data if you can do the same with constraints (see "How Triggers and Constraints Differ").

- Do not create triggers that depend on the order in which a SQL statement processes rows (which can vary).
  
  For example, do not assign a value to a global package variable in a row trigger if the current value of the variable depends on the row being processed by the row trigger. If a trigger updates global package variables, initialize those variables in a BEFORE statement trigger.
• Use **BEFORE** row triggers to modify the row before writing the row data to disk.

• Use **AFTER** row triggers to obtain the row ID and use it in operations.

An **AFTER** row trigger fires when the triggering statement results in ORA-02292.

**Note:**

**AFTER** row triggers are slightly more efficient than **BEFORE** row triggers. With **BEFORE** row triggers, affected data blocks are read first for the trigger and then for the triggering statement. With **AFTER** row triggers, affected data blocks are read only for the trigger.

• If the triggering statement of a **BEFORE** statement trigger is an **UPDATE** or **DELETE** statement that conflicts with an **UPDATE** statement that is running, then the database does a transparent **ROLLBACK** to **SAVEPOINT** and restarts the triggering statement. The database can do this many times before the triggering statement completes successfully. Each time the database restarts the triggering statement, the trigger fires. The **ROLLBACK** to **SAVEPOINT** does not undo changes to package variables that the trigger references. To detect this situation, include a counter variable in the package.

• Do not create recursive triggers.

  For example, do not create an **AFTER** **UPDATE** trigger that issues an **UPDATE** statement on the table on which the trigger is defined. The trigger fires recursively until it runs out of memory.

• If you create a trigger that includes a statement that accesses a remote database, then put the exception handler for that statement in a stored subprogram and invoke the subprogram from the trigger.

  For more information, see “Remote Exception Handling”.

• Use **DATABASE** triggers judiciously. They fire every time any database user initiates a triggering event.

• If a trigger runs the following statement, the statement returns the owner of the trigger, not the user who is updating the table:

  ```sql
  SELECT Username FROM USER_USERS;
  ```

• Only committed triggers fire.

  A trigger is committed, implicitly, after the **CREATE TRIGGER** statement that creates it succeeds. Therefore, the following statement cannot fire the trigger that it creates:

  ```sql
  CREATE OR REPLACE TRIGGER my_trigger
  AFTER CREATE ON DATABASE
  BEGIN
    NULL;
  END;
  /
  ```

• To allow the modular installation of applications that have triggers on the same tables, create multiple triggers of the same type, rather than a single trigger that runs a sequence of operations.

  Each trigger sees the changes made by the previously fired triggers. Each trigger can see **OLD** and **NEW** values.
10.10 Trigger Restrictions

In addition to the restrictions that apply to all PL/SQL units (see Table C-1), triggers have these restrictions:

- **Trigger Size Restriction**
- **Trigger LONG and LONG RAW Data Type Restrictions**
- **Mutating-Table Restriction**
- **Only an autonomous trigger can run TCL or DDL statements.**
  
  For information about autonomous triggers, see "Autonomous Triggers".

- **A trigger cannot invoke a subprogram that runs transaction control statements, because the subprogram runs in the context of the trigger body.**
  
  For more information about subprograms invoked by triggers, see "Subprograms Invoked by Triggers".

- **A trigger cannot access a SERIALLY_REUSABLE package.**
  
  For information about SERIALLY_REUSABLE packages, see "SERIALLY_REUSABLE Packages".

#### See Also:

"Compound DML Trigger Restrictions"

10.10.1 Trigger Size Restriction

The size of the trigger cannot exceed 32K.

If the logic for your trigger requires much more than 60 lines of PL/SQL source text, then put most of the source text in a stored subprogram and invoke the subprogram from the trigger. For information about subprograms invoked by triggers, see "Subprograms Invoked by Triggers".

10.10.2 Trigger LONG and LONG RAW Data Type Restrictions

**Note:**

Oracle supports the LONG and LONG RAW data types only for backward compatibility with existing applications.

In addition to the restrictions that apply to all PL/SQL units (see "LONG and LONG RAW Variables"), triggers have these restrictions:

- **A trigger cannot declare a variable of the LONG or LONG RAW data type.**
10.10.3 Mutating-Table Restriction

**Note:**
This topic applies only to row-level simple DML triggers.

A **mutating table** is a table that is being modified by a DML statement (possibly by the effects of a `DELETE CASCADE` constraint). (A view being modified by an `INSTEAD OF` trigger is not considered to be mutating.)

The mutating-table restriction prevents the trigger from querying or modifying the table that the triggering statement is modifying. When a row-level trigger encounters a mutating table, ORA-04091 occurs, the effects of the trigger and triggering statement are rolled back, and control returns to the user or application that issued the triggering statement, as Example 10-26 shows.

**Caution:**
Oracle Database does not enforce the mutating-table restriction for a trigger that accesses remote nodes, because the database does not support declarative referential constraints between tables on different nodes of a distributed database.

Similarly, the database does not enforce the mutating-table restriction for tables in the same database that are connected by loop-back database links. A loop-back database link makes a local table appear remote by defining an Oracle Net path back to the database that contains the link.

If you must use a trigger to update a mutating table, you can avoid the mutating-table error in either of these ways:

- Use a compound DML trigger (see "Using Compound DML Triggers to Avoid Mutating-Table Error").
- Use a temporary table.
  
  For example, instead of using one `AFTER` each row trigger that updates the mutating table, use two triggers—an `AFTER` each row trigger that updates the temporary table and an `AFTER` statement trigger that updates the mutating table with the values from the temporary table.

**Mutating-Table Restriction Relaxed**

As of Oracle Database 8g Release 1, a deletion from the parent table causes `BEFORE` and `AFTER` triggers to fire once. Therefore, you can create row-level and statement-level triggers that query and modify the parent and child tables. This allows most
foreign key constraint actions to be implemented through their after-row triggers (unless the constraint is self-referential). Update cascade, update set null, update set default, delete set default, inserting a missing parent, and maintaining a count of children can all be implemented easily—see "Triggers for Ensuring Referential Integrity".

However, cascades require care for multiple-row foreign key updates. The trigger cannot miss rows that were changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is invoked.

In Example 10-27, the triggering statement updates \( p \) correctly but causes problems when the trigger updates \( f \). First, the triggering statement changes (1) to (2) in \( p \), and the trigger updates (1) to (2) in \( f \), leaving two rows of value (2) in \( f \). Next, the triggering statement updates (2) to (3) in \( p \), and the trigger updates both rows of value (2) to (3) in \( f \). Finally, the statement updates (3) to (4) in \( p \), and the trigger updates all three rows in \( f \) from (3) to (4). The relationship between the data items in \( p \) and \( f \) is lost.

To avoid this problem, either forbid multiple-row updates to \( p \) that change the primary key and reuse existing primary key values, or track updates to foreign key values and modify the trigger to ensure that no row is updated twice.

**Example 10-26  Trigger Causes Mutating-Table Error**

```sql
-- Create log table
DROP TABLE log;
CREATE TABLE log (  
  emp_id  NUMBER(6),
  l_name  VARCHAR2(25),
  f_name  VARCHAR2(20)
);

-- Create trigger that updates log and then reads employees
CREATE OR REPLACE TRIGGER log_deletions
AFTER DELETE ON employees
FOR EACH ROW
DECLARE
  n INTEGER;
BEGIN
  INSERT INTO log VALUES (
    :OLD.employee_id,
    :OLD.last_name,
    :OLD.first_name
  );

  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.);
END;
/

-- Issue triggering statement:
DELETE FROM employees WHERE employee_id = 197;
```

Result:

DELETE FROM employees WHERE employee_id = 197
ERROR at line 1:
ORA-04091: table HR.EMPLOYEES is mutating, trigger/function might not see it
ORA-06512: at "HR.LOG_DELETIONS", line 10
ORA-04088: error during execution of trigger 'HR.LOG_DELETIONS'

Show that effect of trigger was rolled back:

SELECT count(*) FROM log;

Result:

COUNT(*)
----------
0

1 row selected.

Show that effect of triggering statement was rolled back:

SELECT employee_id, last_name FROM employees WHERE employee_id = 197;

Result:

EMPLOYEE_ID LAST_NAME
----------- -------------------------
197 Feeney

1 row selected.

Example 10-27  Update Cascade

DROP TABLE p;
CREATE TABLE p (p1 NUMBER CONSTRAINT pk_p_p1 PRIMARY KEY);
INSERT INTO p VALUES (1);
INSERT INTO p VALUES (2);
INSERT INTO p VALUES (3);

DROP TABLE f;
CREATE TABLE f (f1 NUMBER CONSTRAINT fk_f_f1 REFERENCES p);
INSERT INTO f VALUES (1);
INSERT INTO f VALUES (2);
INSERT INTO f VALUES (3);

CREATE TRIGGER pt
AFTER UPDATE ON p
FOR EACH ROW
BEGIN
  UPDATE f SET f1 = :NEW.p1 WHERE f1 = :OLD.p1;
END;
/

Query:

SELECT * FROM p ORDER BY p1;

Result:

P1
----------
1
Query:
SELECT * FROM f ORDER BY f1;

Result:

<table>
<thead>
<tr>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Issue triggering statement:
UPDATE p SET p1 = p1+1;

Query:
SELECT * FROM p ORDER BY p1;

Result:

<table>
<thead>
<tr>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Query:
SELECT * FROM f ORDER BY f1;

Result:

<table>
<thead>
<tr>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

10.11 Order in Which Triggers Fire

If two or more triggers with different timing points are defined for the same statement on the same table, then they fire in this order:

1. All BEFORE STATEMENT triggers
2. All BEFORE EACH ROW triggers
3. All AFTER EACH ROW triggers
4. All AFTER STATEMENT triggers
If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order you intend. For information about compound triggers, see "Compound DML Triggers".

If you are creating two or more triggers with the same timing point, and the order in which they fire is important, then you can control their firing order using the follows and precedes clauses (see "FOLLOWS | PRECEDES").

If multiple compound triggers are created on a table, then:

- All before statement sections run at the before statement timing point, before each row sections run at the before each row timing point, and so forth.

  If trigger execution order was specified using the follows clause, then the follows clause determines the order of execution of compound trigger sections. If follows is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the follows clause.

- All after statement sections run at the after statement timing point, after each row sections run at the after each row timing point, and so forth.

  If trigger execution order was specified using the precedes clause, then the precedes clause determines the order of execution of compound trigger sections. If precedes is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the precedes clause.

  Note: precedes applies only to reverse crossedition triggers, which are described in Oracle Database Development Guide.

The firing of compound triggers can be interleaved with the firing of simple triggers.

When one trigger causes another trigger to fire, the triggers are said to be cascading. The database allows up to 32 triggers to cascade simultaneously. To limit the number of trigger cascades, use the initialization parameter open cursors (described in Oracle Database Reference), because a cursor opens every time a trigger fires.

10.12 Trigger Enabling and Disabling

By default, the create trigger statement creates a trigger in the enabled state. To create a trigger in the disabled state, specify disable. Creating a trigger in the disabled state lets you ensure that it compiles without errors before you enable it.

Some reasons to temporarily disable a trigger are:

- The trigger refers to an unavailable object.
- You must do a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

To enable or disable a single trigger, use this statement:

```sql
ALTER TRIGGER [schema.]trigger_name ( ENABLE | DISABLE );
```
To enable or disable all triggers in all editions created on a specific table, use this statement:

```
ALTER TABLE table_name { ENABLE | DISABLE } ALL TRIGGERS;
```

In both of the preceding statements, `schema` is the name of the schema containing the trigger, and the default is your schema.

---

**See Also:**

- "ALTER TRIGGER Statement" for more information about the `ALTER TRIGGER` statement
- Oracle Database SQL Language Reference for more information about the `ALTER TABLE` statement

---

### 10.13 Trigger Changing and Debugging

To change a trigger, you must either replace or re-create it. (The `ALTER TRIGGER` statement only enables, disables, compiles, or renames a trigger.)

To replace a trigger, use the `CREATE TRIGGER` statement with the `OR REPLACE` clause.

To re-create a trigger, first drop it with the `DROP TRIGGER` statement and then create it again with the `CREATE TRIGGER` statement.

To debug a trigger, you can use the facilities available for stored subprograms. For information about these facilities, see Oracle Database Development Guide.

---

**See Also:**

- "CREATE TRIGGER Statement" for more information about the `CREATE TRIGGER` statement
- "DROP TRIGGER Statement" for more information about the `DROP TRIGGER` statement
- "ALTER TRIGGER Statement" for more information about the `ALTER TRIGGER` statement

---

### 10.14 Triggers and Oracle Database Data Transfer Utilities

The Oracle database utilities that transfer data to your database, possibly firing triggers, are:

- **SQL*Loader (`sqlldr`)**
  SQL*Loader loads data from external files into tables of an Oracle database.
  During a SQL*Loader conventional load, `INSERT` triggers fire.
  Before a SQL*Loader direct load, triggers are disabled.
• **Data Pump Import (impdp)**

Data Pump Import (impdp) reads an export dump file set created by Data Pump Export (expdp) and writes it to an Oracle database.

If a table to be imported does not exist on the target database, or if you specify **TABLE_EXISTS_ACTION=REPLACE**, then impdp creates and loads the table before creating any triggers, so no triggers fire.

If a table to be imported exists on the target database, and you specify either **TABLE_EXISTS_ACTION=APPEND** or **TABLE_EXISTS_ACTION=TRUNCATE**, then impdp loads rows into the existing table, and INSERT triggers created on the table fire.

---

• **Original Import (imp)**

Original Import (the original Import utility, imp) reads object definitions and table data from dump files created by original Export (the original Export utility, exp) and writes them to the target database.

---

**Note:**

To import files that original Export created, you must use original Import. In all other cases, Oracle recommends that you use Data Pump Import instead of original Import.

If a table to be imported does not exist on the target database, then imp creates and loads the table before creating any triggers, so no triggers fire.

If a table to be imported exists on the target database, then the Import **IGNORE** parameter determines whether triggers fire during import operations. The **IGNORE** parameter specifies whether object creation errors are ignored or not, resulting in the following behavior:

- If **IGNORE=n** (default), then imp does not change the table and no triggers fire.
- If **IGNORE=y**, then imp loads rows into the existing table, and INSERT triggers created on the table fire.
10.15 Triggers for Publishing Events

To use a trigger to publish an event, create a trigger that:

- Has the event as its triggering event
- Invokes the appropriate subprograms in the DBMS_AQ package, which provides an interface to Oracle Advanced Queuing (AQ)

For information about the DBMS_AQ package, see Oracle Database PL/SQL Packages and Types Reference.

For information about AQ, see Oracle Database Advanced Queuing User's Guide.

By enabling and disabling such triggers, you can turn event notification on and off. For information about enabling and disabling triggers, see "Trigger Enabling and Disabling".

How Triggers Publish Events

When the database detects an event, it fires all enabled triggers that are defined on that event, except:

- Any trigger that is the target of the triggering event.
  
  For example, a trigger for all DROP events does not fire when it is dropped itself.

- Any trigger that was modified, but not committed, in the same transaction as the triggering event.
  
  For example, if a recursive DDL statement in a system trigger modifies another trigger, then events in the same transaction cannot fire the modified trigger.

When a trigger fires and invokes AQ, AQ publishes the event and passes to the trigger the publication context and specified attributes. The trigger can access the attributes by invoking event attribute functions.

The attributes that a trigger can specify to AQ (by passing them to AQ as IN parameters) and then access with event attribute functions depends on the triggering event, which is either a database event or a client event.
A trigger always behaves like a definer rights (DR) unit. The trigger action of an event runs as the definer of the action (as the definer of the package or function in callouts, or as owner of the trigger in queues). Because the owner of the trigger must have EXECUTE privileges on the underlying queues, packages, or subprograms, this action is consistent. For information about DR units, see "Invoker's Rights and Definer's Rights (AUTHID Property)".

The database ignores the return status from callback functions for all events. For example, the database does nothing with the return status from a SHUTDOWN event.

Topics

- Event Attribute Functions
- Event Attribute Functions for Database Event Triggers
- Event Attribute Functions for Client Event Triggers

10.15.1 Event Attribute Functions

By invoking system-defined event attribute functions in Table 10-5, a trigger can retrieve certain attributes of the triggering event. Not all triggers can invoke all event attribute functions—for details, see "Event Attribute Functions for Database Event Triggers" and "Event Attribute Functions for Client Event Triggers".

Note:

- In earlier releases, you had to access these functions through the SYS package. Now Oracle recommends accessing them with their public synonyms (the names starting with ora_ in the first column of Table 10-5).
- The function parameter ora_name_list_t is defined in package DBMS_STANDARD as:

  ```
  TYPE ora_name_list_t IS TABLE OF VARCHAR2(2*(ORA_MAX_NAME_LEN+2)+1);
  ```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Return Type and Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ora_client_ip_address</td>
<td>VARCHAR2: IP address of client in LOGON event when underlying protocol is TCP/IP</td>
<td>DECLARE v_addr VARCHAR2(11);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF (ora_sysevent = 'LOGON') THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v_addr := ora_client_ip_address;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END IF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/*</td>
</tr>
<tr>
<td>ora_database_name</td>
<td>VARCHAR2 (50): Database name</td>
<td>DECLARE v_db_name VARCHAR2(50);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>v_db_name := ora_database_name;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/*</td>
</tr>
<tr>
<td>ora_des_encrypted_password</td>
<td>VARCHAR2: DES-encrypted password of user being created or altered</td>
<td>IF (ora_dict_obj_type = 'USER') THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INSERT INTO event_table VALUES (ora_des_encrypted_password);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END IF</td>
</tr>
<tr>
<td>ora_dict_obj_name</td>
<td>VARCHAR2(128): Name of dictionary object on which DDL operation occurred</td>
<td>INSERT INTO event_table VALUES ('Changed object is '</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/*</td>
</tr>
<tr>
<td>ora_dict_obj_name_list</td>
<td>PLS_INTEGER: Number of object names modified in event</td>
<td>DECLARE name_list ora_name_list_t;</td>
</tr>
<tr>
<td>(name_list OUT ora_name_list_t)</td>
<td></td>
<td>number_modified PLS_INTEGER;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF (ora_sysevent='ASSOCIATE STATISTICS') THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number_modified := ora_dict_obj_name_list(name_list);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END IF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END;</td>
</tr>
<tr>
<td>ora_dict_obj_owner</td>
<td>VARCHAR2(128): Owner of dictionary object on which DDL operation occurred</td>
<td>INSERT INTO event_table VALUES ('object owner is '</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/*</td>
</tr>
<tr>
<td>ora_dict_obj_owner_list</td>
<td>PLS_INTEGER: Number of owners of objects modified in event</td>
<td>DECLARE owner_list ora_name_list_t;</td>
</tr>
<tr>
<td>(owner_list OUT ora_name_list_t)</td>
<td></td>
<td>number_modified PLS_INTEGER;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF (ora_sysevent='ASSOCIATE STATISTICS') THEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number_modified := ora_dict_obj_name_list(owner_list);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END IF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>END;</td>
</tr>
</tbody>
</table>
### Table 10-5  (Cont.) System-Defined Event Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Return Type and Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ora_dict_obj_type</td>
<td>VARCHAR2(20): Type of dictionary object on which DDL operation occurred</td>
<td>INSERT INTO event_table VALUES ('This object is a '</td>
</tr>
<tr>
<td>ora_grantee (user_list OUT ora_name_list_t)</td>
<td>PLS_INTEGER: Number of grantees in grant event</td>
<td>DECLARE user_list ora_name_list_t; number_of_grantees PLS_INTEGER; BEGIN IF (ora_sysevent = 'GRANT') THEN number_of_grantees := ora_grantee(user_list); END IF; END;</td>
</tr>
<tr>
<td>ora_instance_num</td>
<td>NUMBER: Instance number</td>
<td>IF (ora_instance_num = 1) THEN INSERT INTO event_table VALUES ('1'); END IF;</td>
</tr>
<tr>
<td>ora_is_alter_column (column_name IN VARCHAR2)</td>
<td>BOOLEAN: TRUE if specified column is altered, FALSE otherwise</td>
<td>IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN alter_column := ora_is_alter_column('C'); END IF;</td>
</tr>
<tr>
<td>ora_is_creating_nested_table</td>
<td>BOOLEAN: TRUE if current event is creating nested table, FALSE otherwise</td>
<td>IF (ora_sysevent = 'CREATE' AND ora_dict_obj_type = 'TABLE') THEN INSERT INTO event_table VALUES ('A nested table is created'); END IF;</td>
</tr>
<tr>
<td>ora_is_drop_column (column_name IN VARCHAR2)</td>
<td>BOOLEAN: TRUE if specified column is dropped, FALSE otherwise</td>
<td>IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN drop_column := ora_is_drop_column('C'); END IF;</td>
</tr>
<tr>
<td>ora_is_servererror (error_number IN VARCHAR2)</td>
<td>BOOLEAN: TRUE if given error is on error stack, FALSE otherwise</td>
<td>IF ora_is_servererror(error_number) THEN INSERT INTO event_table VALUES ('Server error!!'); END IF;</td>
</tr>
<tr>
<td>ora_login_user</td>
<td>VARCHAR2(128): Login user name</td>
<td>SELECT ora_login_user FROM DUAL;</td>
</tr>
<tr>
<td>ora_partition_pos</td>
<td>PLS_INTEGER: In INSTEAD OF trigger for CREATE TABLE, position in SQL text where you can insert PARTITION clause</td>
<td>-- Retrieve ora_sql_txt into sql_text variable  v_n := ora_partition_pos;  v_new_stmt := SUBSTR(sql_text,1,v_n - 1)</td>
</tr>
</tbody>
</table>
### Table 10-5  (Cont.) System-Defined Event Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Return Type and Value</th>
<th>Example</th>
</tr>
</thead>
</table>
| ora_privilege_list (privilege_list OUT ora_name_list_t) | PLS_INTEGER: Number of privileges in grant or revoke event OUT parameter: List of privileges granted or revoked in event | DECLARE privilege_list ora_name_list_t;
number_of_privileges PLS_INTEGER;
BEGIN
  IF (ora_sysevent = 'GRANT' OR ora_sysevent = 'REVOKE') THEN
    number_of_privileges := ora_privilege_list(privilege_list);
  END IF;
END; |
| ora_revokee (user_list OUT ora_name_list_t) | PLS_INTEGER: Number of revokees in revoke event OUT parameter: List of revokees in event | DECLARE user_list ora_name_list_t;
number_of_users PLS_INTEGER;
BEGIN
  IF (ora_sysevent = 'REVOKE') THEN
    number_of_users := ora_revokee(user_list);
  END IF;
END; |
| ora_server_error (position IN PLS_INTEGER) | NUMBER: Error code at given position on error stack | INSERT INTO event_table VALUES ('top stack error ' || ora_server_error(1)); |
| ora_server_error_depth | PLS_INTEGER: Number of error messages on error stack | n := ora_server_error_depth; -- Use n with functions such as ora_server_error |
| ora_server_error_msg (position IN PLS_INTEGER) | VARCHAR2: Error message at given position on error stack | INSERT INTO event_table VALUES ('top stack error message' || ora_server_error_msg(1)); |
| ora_server_error_num_params (position IN PLS_INTEGER) | PLS_INTEGER: Number of strings substituted into error message (using format like %s) at given position on error stack | n := ora_server_error_num_params(1); |
| ora_server_error_param (position IN PLS_INTEGER, param IN PLS_INTEGER) | VARCHAR2: Matching substitution value (%s, %d, and so on) in error message at given position and parameter number | -- Second %s in "Expected %s, found %s": param := ora_server_error_param(1,2); |
### Table 10-5  (Cont.) System-Defined Event Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Return Type and Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ora_sql_txt (sql_text OUT ora_name_list_t)</code></td>
<td>PLS_INTEGER: Number of elements in PL/SQL table OUT parameter: SQL text of triggering statement (broken into multiple collection elements if statement is long)</td>
<td>CREATE TABLE event_table (col VARCHAR2(2030)); DECLARE sql_text ora_name_list_t; n PLS_INTEGER; v_stmt VARCHAR2(2000); BEGIN n := ora_sql_txt(sql_text); FOR i IN 1..n LOOP v_stmt := v_stmt</td>
</tr>
<tr>
<td><code>ora_sysevent</code></td>
<td>VARCHAR2(20): Name of triggering event, as given in syntax</td>
<td>INSERT INTO event_table VALUES (ora_sysevent);</td>
</tr>
<tr>
<td><code>ora_with_grant_option</code></td>
<td>BOOLEAN: TRUE if privileges are granted with GRANT option, FALSE otherwise</td>
<td>IF (ora_sysevent = 'GRANT' AND ora_with_grant_option = TRUE) THEN INSERT INTO event_table VALUES ('with grant option'); END IF;</td>
</tr>
<tr>
<td><code>ora_space_error_info (error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2)</code></td>
<td>BOOLEAN: TRUE if error is related to out-of-space condition, FALSE otherwise OUT parameters: Information about object that caused error</td>
<td>IF (ora_space_error_info (eno,typ,owner,ts,obj,subobj) = TRUE) THEN DBMS_OUTPUT.PUT_LINE('The object '</td>
</tr>
</tbody>
</table>

1. Position 1 is the top of the stack.

### 10.15.2 Event Attribute Functions for Database Event Triggers

Table 10-6 summarizes the database event triggers that can invoke event attribute functions. For more information about the triggering events in Table 10-6, see "database_event".
### Table 10-6 Database Event Triggers

<table>
<thead>
<tr>
<th>Triggering Event</th>
<th>When Trigger Fires</th>
<th>WHEN Conditions</th>
<th>Restrictions</th>
<th>Transaction Details</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFTER STARTUP</td>
<td>When database is opened.</td>
<td>None allowed</td>
<td>Trigger cannot do database operations.</td>
<td>Starts a separate transaction and commits it after firing the triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE SHUTDOWN</td>
<td>Just before server starts shutdown of an instance.</td>
<td>None allowed</td>
<td>Trigger cannot do database operations.</td>
<td>Starts separate transaction and commits it after firing triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name</td>
</tr>
<tr>
<td></td>
<td>This lets the cartridge shutdown completely. For abnormal instance shutdown, this trigger might not fire.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER DB_ROLE_CHANGE</td>
<td>When database is opened for first time after role change.</td>
<td>None allowed</td>
<td>None</td>
<td>Starts separate transaction and commits it after firing triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER SERVERERROR</td>
<td>With condition, whenever specified error occurs. Without condition, whenever any error occurs. Trigger does not fire for errors listed in &quot;database_event&quot;. ERRNO = eno</td>
<td>Depends on error</td>
<td></td>
<td>Starts separate transaction and commits it after firing triggers.</td>
<td>ora_sysevent, ora_login_user, ora_instance_num, ora_database_name, ora_server_error, ora_is_servererror, ora_space_error_info</td>
</tr>
</tbody>
</table>

### 10.15.3 Event Attribute Functions for Client Event Triggers

Table 10-7 summarizes the client event triggers that can invoke event attribute functions. For more information about the triggering events in Table 10-7, see "ddl_event" and "database_event".

**Note:**

If a client event trigger becomes the target of a DDL operation (such as CREATE OR REPLACE TRIGGER), then it cannot fire later during the same transaction.
### Table 10-7  Client Event Triggers

<table>
<thead>
<tr>
<th>Triggering Event</th>
<th>When Trigger Fires</th>
<th>WHEN Conditions</th>
<th>Restrictions</th>
<th>Transaction</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE ALTER</td>
<td>When catalog object is altered</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column (for ALTER TABLE events) ora_is_drop_column (for ALTER TABLE events)</td>
</tr>
<tr>
<td>AFTER ALTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE DROP</td>
<td>When catalog object is dropped</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner</td>
</tr>
<tr>
<td>AFTER DROP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE ANALYZE</td>
<td>When ANALYZE statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner</td>
</tr>
<tr>
<td>Triggering Event</td>
<td>When Trigger Fires</td>
<td>WHEN Conditions</td>
<td>Restrictions</td>
<td>Transaction</td>
<td>Attribute Functions</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>-------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>BEFORE ASSOCIATE STATISTICS</td>
<td>When ASSOCIATE STATISTICS statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list</td>
</tr>
<tr>
<td>AFTER ASSOCIATE STATISTICS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE AUDIT</td>
<td>When AUDIT or NOAUDIT statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name</td>
</tr>
<tr>
<td>AFTER NOAUDIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE NOAUDIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER NOAUDIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE COMMENT</td>
<td>When object is commented</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner</td>
</tr>
<tr>
<td>AFTER COMMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 10-7  (Cont.) Client Event Triggers

<table>
<thead>
<tr>
<th>Triggering Event</th>
<th>When Trigger Fires</th>
<th>WHEN Conditions</th>
<th>Restrictions</th>
<th>Transaction</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE CREATE</td>
<td>When catalog object is created</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_dict_obj_type</code>&lt;br&gt;<code>ora_dict_obj_name</code>&lt;br&gt;<code>ora_dict_obj_owner</code>&lt;br&gt;<code>ora_is_creating_nested_table</code>&lt;br&gt;(for CREATE TABLE events)</td>
</tr>
<tr>
<td>AFTER CREATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE DDL</td>
<td>When most SQL DDL statements are issued. Not fired for <code>ALTER DATABASE</code>, <code>CREATE CONTROLFILE</code>, <code>CREATE DATABASE</code>, and DDL issued through the PL/SQL subprogram interface, such as creating an advanced queue.</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_dict_obj_name</code>&lt;br&gt;<code>ora_dict_obj_type</code>&lt;br&gt;<code>ora_dict_obj_owner</code></td>
</tr>
<tr>
<td>AFTER DDL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE DISASSOCIATE STATISTICS</td>
<td>When DISASSOCIATE STATISTICS statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td><code>ora_sysevent</code>&lt;br&gt;<code>ora_login_user</code>&lt;br&gt;<code>ora_instance_num</code>&lt;br&gt;<code>ora_database_name</code>&lt;br&gt;<code>ora_dict_obj_name</code>&lt;br&gt;<code>ora_dict_obj_type</code>&lt;br&gt;<code>ora_dict_obj_owner</code>&lt;br&gt;<code>ora_dict_obj_name_list</code>&lt;br&gt;<code>ora_dict_obj_owner_list</code></td>
</tr>
<tr>
<td>Triggering Event</td>
<td>When Trigger Fires</td>
<td>WHEN Conditions</td>
<td>Restrictions</td>
<td>Transaction</td>
<td>Attribute Functions</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>-------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>BEFORE GRANT</td>
<td>When GRANT statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privilege_list</td>
</tr>
<tr>
<td>AFTER GRANT</td>
<td>At start of user logoff</td>
<td>Simple conditions on UID and USER</td>
<td>DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name</td>
</tr>
<tr>
<td>BEFORE LOGOFF</td>
<td>After successful user logon</td>
<td>Simple conditions on UID and USER</td>
<td>DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Starts separate transaction and commits it after firing triggers.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address</td>
</tr>
<tr>
<td>BEFORE RENAME</td>
<td>When RENAME statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type</td>
</tr>
</tbody>
</table>
## Table 10-7  (Cont.) Client Event Triggers

<table>
<thead>
<tr>
<th>Triggering Event</th>
<th>When Trigger Fires</th>
<th>WHEN Conditions</th>
<th>Restrictions</th>
<th>Transaction</th>
<th>Attribute Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE REVOKE</td>
<td>When REVOKE statement is issued</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privilege_list</td>
</tr>
<tr>
<td>AFTER REVOKE</td>
<td>After SQL statement is suspended because of out-of-space condition. (Trigger must correct condition so statement can be resumed.)</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror ora_space_error_info</td>
</tr>
<tr>
<td>BEFORE TRUNCATE</td>
<td>When object is truncated</td>
<td>Simple conditions on type and name of object, UID, and USER</td>
<td>Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.</td>
<td>Fires triggers in current transaction.</td>
<td>ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner</td>
</tr>
<tr>
<td>AFTER TRUNCATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 10.16 Views for Information About Triggers

The *_TRIGGERS* static data dictionary views reveal information about triggers. For information about these views, see *Oracle Database Reference*. 
Example 10-28  Viewing Information About Triggers

This example creates a trigger and queries the static data dictionary view USER_TRIGGERS twice—first to show its type, triggering event, and the name of the table on which it is created, and then to show its body.

CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON employees
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/

These SQL*Plus commands format the query results.

COLUMN Trigger_type FORMAT A15
COLUMN Triggering_event FORMAT A16
COLUMN Table_name FORMAT A11
COLUMN Trigger_body FORMAT A50
SET LONG 9999

Query:

SELECT Trigger_type, Triggering_event, Table_name 
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';

Result:

<table>
<thead>
<tr>
<th>TRIGGER_TYPE</th>
<th>TRIGGERING_EVENT</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFTER STATEMENT</td>
<td>DELETE</td>
<td>EMPLOYEES</td>
</tr>
</tbody>
</table>

Query:

SELECT Trigger_body 
FROM USER_TRIGGERS 
WHERE Trigger_name = 'EMP_COUNT';

Result:

<table>
<thead>
<tr>
<th>TRIGGER_BODY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE</td>
</tr>
<tr>
<td>n  INTEGER;</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>SELECT COUNT(*) INTO n FROM employees;</td>
</tr>
<tr>
<td>DBMS_OUTPUT.PUT_LINE('There are now '</td>
</tr>
<tr>
<td>END;</td>
</tr>
</tbody>
</table>
PL/SQL Packages

This chapter explains how to bundle related PL/SQL code and data into a package, whose contents are available to many applications.

Topics

- What is a Package?
- Reasons to Use Packages
- Package Specification
- Package Body
- Package Instantiation and Initialization
- Package State
- SERIALLY_REUSABLE Packages
- Package Writing Guidelines
- Package Example
- How STANDARD Package Defines the PL/SQL Environment

11.1 What is a Package?

A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents.

A package always has a specification, which declares the public items that can be referenced from outside the package.

If the public items include cursors or subprograms, then the package must also have a body. The body must define queries for public cursors and code for public subprograms. The body can also declare and define private items that cannot be referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an initialization part, whose statements initialize variables and do other one-time setup steps, and an exception-handling part. You can change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box.

In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using a call specification, which maps the external subprogram name, parameter types, and return type to their SQL counterparts.

The AUTHID clause of the package specification determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.
The **ACCESSIBLE BY clause** of the package specification lets you specify a white list of PL/SQL units that can access the package. You use this clause in situations like these:

- You implement a PL/SQL application as several packages—one package that provides the application programming interface (API) and helper packages to do the work. You want clients to have access to the API, but not to the helper packages. Therefore, you omit the **ACCESSIBLE BY** clause from the API package specification and include it in each helper package specification, where you specify that only the API package can access the helper package.

- You create a utility package to provide services to some, but not all, PL/SQL units in the same schema. To restrict use of the package to the intended units, you list them in the **ACCESSIBLE BY** clause in the package specification.

### See Also:

- "Package Specification" for more information about the package specification
- "Package Body" for more information about the package body
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"
- "Invoker's Rights and Definer's Rights (AUTHID Property)"

## 11.2 Reasons to Use Packages

Packages support the development and maintenance of reliable, reusable code with the following features:

- **Modularity**
  
  Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules. You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined. This practice aids application development.

- **Easier Application Design**
  
  When designing an application, all you need initially is the interface information in the package specifications. You can code and compile specifications without their bodies. Next, you can compile standalone subprograms that reference the packages. You need not fully define the package bodies until you are ready to complete the application.

- **Hidden Implementation Details**
  
  Packages let you share your interface information in the package specification, and hide the implementation details in the package body. Hiding the implementation details in the body has these advantages:
  
  - You can change the implementation details without affecting the application interface.
– Application users cannot develop code that depends on implementation
details that you might want to change.

• **Added Functionality**
  Package public variables and cursors can persist for the life of a session. They
  can be shared by all subprograms that run in the environment. They let you
  maintain data across transactions without storing it in the database. (For the
  situations in which package public variables and cursors do not persist for the life
  of a session, see "Package State").

• **Better Performance**
  The first time you invoke a package subprogram, Oracle Database loads the
  whole package into memory. Subsequent invocations of other subprograms in
  same the package require no disk I/O.

  Packages prevent cascading dependencies and unnecessary recompiling. For
  example, if you change the body of a package function, Oracle Database does not
  recompile other subprograms that invoke the function, because these
  subprograms depend only on the parameters and return value that are declared in
  the specification.

• **Easier to Grant Roles**
  You can grant roles on the package, instead of granting roles on each object in the
  package.

  **Note:**
  You cannot reference host variables from inside a package.

### 11.3 Package Specification

A **package specification** declares **public items**. The scope of a public item is the
schema of the package. A public item is visible everywhere in the schema. To
reference a public item that is in scope but not visible, qualify it with the package
name. (For information about scope, visibility, and qualification, see "Scope and
Visibility of Identifiers").

Each public item declaration has all information needed to use the item. For example,
suppose that a package specification declares the function **factorial** this way:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

The declaration shows that **factorial** needs one argument of type **INTEGER** and
returns a value of type **INTEGER**, which is invokers must know to invoke **factorial**.
Invokers need not know how **factorial** is implemented (for example, whether it is
iterative or recursive).
To restrict the use of your package to specified PL/SQL units, include the **ACCESSIBLE BY** clause in the package specification.

### Topics

- Appropriate Public Items
- Creating Package Specifications

#### 11.3.1 Appropriate Public Items

Appropriate public items are:

- Types, variables, constants, subprograms, cursors, and exceptions used by multiple subprograms
  
  A type defined in a package specification is either a PL/SQL user-defined subtype (described in "User-Defined PL/SQL Subtypes") or a PL/SQL composite type (described in PL/SQL Collections and Records).

- Associative array types of standalone subprogram parameters
  
  You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone subprogram, you must declare the type of that variable in a package specification. Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and to the invoking subprogram or anonymous block (which declares a variable of that type). See Example 11-2.

- Variables that must remain available between subprogram invocations in the same session

- Subprograms that read and write public variables ("get" and "set" subprograms)
  
  Provide these subprograms to discourage package users from reading and writing public variables directly.

- Subprograms that invoke each other
  
  You need not worry about compilation order for package subprograms, as you must for standalone subprograms that invoke each other.

- Overloaded subprograms
  
  Overloaded subprograms are variations of the same subprogram. That is, they have the same name but different formal parameters. For more information about them, see “Overloaded Subprograms”.

---

**Note:**

A PL/SQL composite type defined in a package specification is incompatible with an identically defined local or standalone type (see Example 6-36, Example 6-37, and Example 6-42).
11.3.2 Creating Package Specifications

To create a package specification, use the "CREATE PACKAGE Statement".

Because the package specifications in Example 11-1 and Example 11-2 do not declare cursors or subprograms, the packages trans_data and aa_pkg do not need bodies.

Example 11-1 Simple Package Specification

In this example, the specification for the package trans_data declares two public types and three public variables.

```
CREATE OR REPLACE PACKAGE trans_data AUTHID DEFINER AS
    TYPE TimeRec IS RECORD (
        minutes SMALLINT,
        hours   SMALLINT);
    TYPE TransRec IS RECORD (
        category VARCHAR2(10),
        account  INT,
        amount   REAL,
        time_of  TimeRec);
    minimum_balance     CONSTANT REAL := 10.00;
    number_processed    INT;
    insufficient_funds  EXCEPTION;
    PRAGMA EXCEPTION_INIT(insufficient_funds, -4097);
END trans_data;
/
```

Example 11-2 Passing Associative Array to Standalone Subprogram

In this example, the specification for the package aa_pkg declares an associative array type, aa_type. Then, the standalone procedure print_aa declares a formal parameter of type aa_type. Next, the anonymous block declares a variable of type aa_type, populates it, and passes it to the procedure print_aa, which prints it.

```
CREATE OR REPLACE PACKAGE aa_pkg AUTHID DEFINER IS
    TYPE aa_type IS TABLE OF INTEGER INDEX BY VARCHAR2(15);
END;
/
CREATE OR REPLACE PROCEDURE print_aa (aa aa_pkg.aa_type)
    AUTHID DEFINER IS
    i  VARCHAR2(15);
BEGIN
    i := aa.FIRST;
    WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE (aa(i) || ' ' || i);
        i := aa.NEXT(i);
    END LOOP;
```
DECLARE
    aa_var  aa_pkg.aa_type;
BEGIN
    aa_var('zero') := 0;
    aa_var('one') := 1;
    aa_var('two') := 2;
    print_aa(aa_var);
END;
/

Result:
1 one
2 two
0 zero

11.4 Package Body

If a package specification declares cursors or subprograms, then a package body is required; otherwise, it is optional. The package body and package specification must be in the same schema.

Every cursor or subprogram declaration in the package specification must have a corresponding definition in the package body. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

To create a package body, use the "CREATE PACKAGE BODY Statement".

The cursors and subprograms declared in the package specification and defined in the package body are public items that can be referenced from outside the package. The package body can also declare and define private items that cannot be referenced from outside the package, but are necessary for the internal workings of the package.

Finally, the body can have an initialization part, whose statements initialize public variables and do other one-time setup steps. The initialization part runs only the first time the package is referenced. The initialization part can include an exception handler.

You can change the package body without changing the specification or the references to the public items.

Example 11-3 Matching Package Specification and Body

In this example, the headings of the corresponding subprogram declaration and definition do not match word for word; therefore, PL/SQL raises an exception, even though employees.hire_date%TYPE is DATE.

CREATE PACKAGE emp_bonus AS
    PROCEDURE calc_bonus (date_hired employees.hire_date%TYPE);
END emp_bonus;
/
CREATE PACKAGE BODY emp_bonus AS
    -- DATE does not match employees.hire_date%TYPE
    PROCEDURE calc_bonus (date_hired DATE) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE
            ('Employees hired on ' || date_hired || ' get bonus.');
Warning: Package Body created with compilation errors.

Show errors (in SQL*Plus):
SHOW ERRORS

Errors for PACKAGE BODY EMP_BONUS:

LINE/COL ERROR
-------- -----------------------------------------------------------------
2/13     PLS-00323: subprogram or cursor 'CALC_BONUS' is declared in a
          package specification and must be defined in the package body

Correct problem:
CREATE OR REPLACE PACKAGE BODY emp_bonus AS
PROCEDURE calc_bonus
  (date_hired employees.hire_date%TYPE) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE
       ('Employees hired on ' || date_hired || ' get bonus.');
END;
END emp_bonus;
/

Result:
Package body created.

11.5 Package Instantiation and Initialization

When a session references a package item, Oracle Database instantiates the package
for that session. Every session that references a package has its own instantiation of
that package.

When Oracle Database instantiates a package, it initializes it. Initialization includes
whichever of the following are applicable:

• Assigning initial values to public constants
• Assigning initial values to public variables whose declarations specify them
• Executing the initialization part of the package body

11.6 Package State

The values of the variables, constants, and cursors that a package declares (in either
its specification or body) comprise its package state.

If a PL/SQL package declares at least one variable, constant, or cursor, then the
package is stateful; otherwise, it is stateless.
Each session that references a package item has its own instantiation of that package. If the package is stateful, the instantiation includes its state.

The package state persists for the life of a session, except in these situations:

- The package is **SERIALLY_REUSABLE**.
- The package body is recompiled.

If the body of an instantiated, stateful package is recompiled (either explicitly, with the "ALTER PACKAGE Statement", or implicitly), the next invocation of a subprogram in the package causes Oracle Database to discard the existing package state and raise the exception ORA-04068.

After PL/SQL raises the exception, a reference to the package causes Oracle Database to re-instantiate the package, which re-initializes it. Therefore, previous changes to the package state are lost.

- Any of the session's instantiated packages are invalidated and revalidated.
- All of a session's package instantiations (including package states) can be lost if any of the session's instantiated packages are invalidated and revalidated.

Oracle Database treats a package as stateless if its state is constant for the life of a session (or longer). This is the case for a package whose items are all compile-time constants.

A **compile-time constant** is a constant whose value the PL/SQL compiler can determine at compilation time. A constant whose initial value is a literal is always a compile-time constant. A constant whose initial value is not a literal, but which the optimizer reduces to a literal, is also a compile-time constant. Whether the PL/SQL optimizer can reduce a nonliteral expression to a literal depends on optimization level. Therefore, a package that is stateless when compiled at one optimization level might be stateful when compiled at a different optimization level.

---

**See Also:**

- "**SERIALLY_REUSABLE Packages**"
- "**Package Instantiation and Initialization**" for information about initialization
- *Oracle Database Development Guide* for information about invalidation and revalidation of schema objects
- "**PL/SQL Optimizer**" for information about the optimizer

---

### 11.7 SERIALLY_REUSABLE Packages

**SERIALLY_REUSABLE** packages let you design applications that manage memory better for scalability.

If a package is not **SERIALLY_REUSABLE**, its package state is stored in the user global area (UGA) for each user. Therefore, the amount of UGA memory needed increases linearly with the number of users, limiting scalability. The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days.
If a package is `SERIALLY_REUSABLE`, its package state is stored in a work area in a small pool in the system global area (SGA). The package state persists only for the life of a server call. After the server call, the work area returns to the pool. If a subsequent server call references the package, then Oracle Database reuses an instantiation from the pool. Reusing an instantiation re-initializes it; therefore, changes made to the package state in previous server calls are invisible. (For information about initialization, see "Package Instantiation and Initialization").

**Note:**

Trying to access a `SERIALLY_REUSABLE` package from a database trigger, or from a PL/SQL subprogram invoked by a SQL statement, raises an error.

**Topics**

- Creating `SERIALLY_REUSABLE` Packages
- `SERIALLY_REUSABLE` Package Work Unit
- Explicit Cursors in `SERIALLY_REUSABLE` Packages

### 11.7.1 Creating `SERIALLY_REUSABLE` Packages

To create a `SERIALLY_REUSABLE` package, include the `SERIALLY_REUSABLE` pragma in the package specification and, if it exists, the package body.

**Example 11-4** creates two very simple `SERIALLY_REUSABLE` packages, one with only a specification, and one with both a specification and a body.

**See Also:**

"`SERIALLY_REUSABLE` Pragma"

**Example 11-4   Creating `SERIALLY_REUSABLE` Packages**

```sql
-- Create bodiless `SERIALLY_REUSABLE` package:
CREATE OR REPLACE PACKAGE bodiless_pkg AUTHID DEFINER IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

-- Create `SERIALLY_REUSABLE` package with specification and body:
CREATE OR REPLACE PACKAGE pkg AUTHID DEFINER IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

CREATE OR REPLACE PACKAGE BODY pkg IS
  PRAGMA SERIALLY_REUSABLE;
```
11.7.2 SERIALLY_REUSABLE Package Work Unit

For a SERIALLY_REUSABLE package, the work unit is a server call. You must use its public variables only within the work unit.

**Note:**

If you make a mistake and depend on the value of a public variable that was set in a previous work unit, then your program can fail. PL/SQL cannot check for such cases.

After the work unit (server call) of a SERIALLY_REUSABLE package completes, Oracle Database does the following:

- Closes any open cursors.
- Frees some nonreusable memory (for example, memory for collection and long VARCHAR2 variables)
- Returns the package instantiation to the pool of reusable instantiations kept for this package.

**Example 11-5  Effect of SERIALLY_REUSABLE Pragma**

In this example, the bodiless packages `pkg` and `sr_pkg` are the same, except that `sr_pkg` is SERIALLY_REUSABLE and `pkg` is not. Each package declares public variable `n` with initial value 5. Then, an anonymous block changes the value of each variable to 10. Next, another anonymous block prints the value of each variable. The value of `pkg.n` is still 10, because the state of `pkg` persists for the life of the session. The value of `sr_pkg.n` is 5, because the state of `sr_pkg` persists only for the life of the server call.

```sql
CREATE OR REPLACE PACKAGE pkg IS
  n NUMBER := 5;
END pkg;
/

CREATE OR REPLACE PACKAGE sr_pkg IS
  n NUMBER := 5;
  PRAGMA SERIALLY_REUSABLE;
END sr_pkg;
/

BEGIN
  pkg.n := 10;
  sr_pkg.n := 10;
END;
/

BEGIN
  DBMS_OUTPUT.PUT_LINE('pkg.n: ' || pkg.n);
  DBMS_OUTPUT.PUT_LINE('sr_pkg.n: ' || sr.pkg.n);
```
11.7.3 Explicit Cursors in SERIALLY_REUSABLE Packages

An explicit cursor in a SERIALLY_REUSABLE package remains open until either you close it or its work unit (server call) ends. To re-open the cursor, you must make a new server call. A server call can be different from a subprogram invocation, as Example 11-6 shows.

In contrast, an explicit cursor in a package that is not SERIALLY_REUSABLE remains open until you either close it or disconnect from the session.

**Example 11-6  Cursor in SERIALLY_REUSABLE Package Open at Call Boundary**

```sql
DROP TABLE people;
CREATE TABLE people (name VARCHAR2(20));

INSERT INTO people (name) VALUES ('John Smith');
INSERT INTO people (name) VALUES ('Mary Jones');
INSERT INTO people (name) VALUES ('Joe Brown');
INSERT INTO people (name) VALUES ('Jane White');

CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  CURSOR c IS SELECT name FROM people;
END sr_pkg;
/

CREATE OR REPLACE PROCEDURE fetch_from_cursor IS
  v_name  people.name%TYPE;
BEGIN
  IF sr_pkg.c%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE('Cursor is open. ');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Cursor is closed; opening now. ');
    OPEN sr_pkg.c;
  END IF;

  FETCH sr_pkg.c INTO v_name;
  DBMS_OUTPUT.PUT_LINE('Fetched: ' || v_name);

  FETCH sr_pkg.c INTO v_name;
  DBMS_OUTPUT.PUT_LINE('Fetched: ' || v_name);
END fetch_from_cursor;
/

First call to server:

BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
```
END;
/

Result:

**Cursor is closed; opening now.**
Fetched: John Smith
Fetched: Mary Jones

**Cursor is open.**
Fetched: Joe Brown
Fetched: Jane White

New call to server:

BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/

Result:

**Cursor is closed; opening now.**
Fetched: John Smith
Fetched: Mary Jones

**Cursor is open.**
Fetched: Joe Brown
Fetched: Jane White

### 11.8 Package Writing Guidelines

- Become familiar with the packages that Oracle Database supplies, and avoid writing packages that duplicate their features.
  
  For more information about the packages that Oracle Database supplies, see *Oracle Database PL/SQL Packages and Types Reference*.

- Keep your packages general so that future applications can reuse them.

- Design and define the package specifications before the package bodies.

- In package specifications, declare only items that must be visible to invoking programs.

  This practice prevents other developers from building unsafe dependencies on your implementation details and reduces the need for recompilation.

  If you change the package specification, you must recompile any subprograms that invoke the public subprograms of the package. If you change only the package body, you need not recompile those subprograms.

- Declare public cursors in package specifications and define them in package bodies, as in Example 11-7.

  This practice lets you hide cursors' queries from package users and change them without changing cursor declarations.

- Assign initial values in the initialization part of the package body instead of in declarations.

  This practice has these advantages:
– The code for computing the initial values can be more complex and better documented.
– If computing an initial value raises an exception, the initialization part can handle it with its own exception handler.

• If you implement a database application as several PL/SQL packages—one package that provides the API and helper packages to do the work, then make the helper packages available only to the API package, as in Example 11-8.

In Example 11-7, the declaration and definition of the cursor c1 are in the specification and body, respectively, of the package emp_stuff. The cursor declaration specifies only the data type of the return value, not the query, which appears in the cursor definition (for complete syntax and semantics, see “Explicit Cursor Declaration and Definition”).

Example 11-8 creates an API package and a helper package. Because of the ACCESSIBLE BY clause in the helper package specification, only the API package can access the helper package.

**Example 11-7  Separating Cursor Declaration and Definition in Package**

```sql
CREATE PACKAGE emp_stuff AS
    CURSOR c1 RETURN employees%ROWTYPE;  -- Declare cursor
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
    CURSOR c1 RETURN employees%ROWTYPE IS
        SELECT * FROM employees WHERE salary > 2500;  -- Define cursor
END emp_stuff;
/
```

**Example 11-8  ACCESSIBLE BY Clause**

```sql
CREATE OR REPLACE PACKAGE helper
    AUTHID DEFINER
    ACCESSIBLE BY (api)
IS
    PROCEDURE h1;
    PROCEDURE h2;
END;
/

CREATE OR REPLACE PACKAGE BODY helper
IS
    PROCEDURE h1 IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Helper procedure h1');
    END;

    PROCEDURE h2 IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Helper procedure h2');
    END;
END;
/

CREATE OR REPLACE PACKAGE api
    AUTHID DEFINER
IS
    PROCEDURE p1;
    PROCEDURE p2;
```
CREATE OR REPLACE PACKAGE BODY api
IS
PROCEDURE p1 IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('API procedure p1');
  helper.h1;
END;

PROCEDURE p2 IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('API procedure p2');
  helper.h2;
END;
END;
/

Invoke procedures in API package:
BEGIN
  api.p1;
  api.p2;
END;
/

Result:
API procedure p1
Helper procedure h1
API procedure p2
Helper procedure h2

Invoke a procedure in helper package:
BEGIN
  helper.h1;
END;
/

Result:
SQL> BEGIN
  2    helper.h1;
  3  END;
  4  /
  5  helper.h1;
  6  *
ERROR at line 2:
ORA-06550: line 2, column 3:
  PLS-00904: insufficient privilege to access object HELPER
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
Example 11-9 creates a table, log, and a package, emp_admin, and then invokes package subprograms from an anonymous block. The package has both specification and body.

The specification declares a public type, cursor, and exception, and three public subprograms. One public subprogram is overloaded (for information about overloaded subprograms, see "Overloaded Subprograms").

The body declares a private variable, defines the public cursor and subprograms that the specification declares, declares and defines a private function, and has an initialization part.

The initialization part (which runs only the first time the anonymous block references the package) inserts one row into the table log and initializes the private variable number_hired to zero. Every time the package procedure hire_employee is invoked, it updates the private variable number_hired.

Example 11-9    Creating emp_admin Package

-- Log to track changes (not part of package):

DROP TABLE log;
CREATE TABLE log (    date_of_action DATE,
    user_id VARCHAR2(20),
    package_name VARCHAR2(30)
);

-- Package specification:

CREATE OR REPLACE PACKAGE emp_admin AUTHID DEFINER AS
    -- Declare public type, cursor, and exception:
    TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
    CURSOR desc_salary RETURN EmpRecTyp;
    invalid_salary EXCEPTION;

    -- Declare public subprograms:

    FUNCTION hire_employee (    last_name VARCHAR2,
        first_name VARCHAR2,
        email VARCHAR2,
        phone_number VARCHAR2,
        job_id VARCHAR2,
        salary NUMBER,
        commission_pct NUMBER,
        manager_id NUMBER,
        department_id NUMBER
    ) RETURN NUMBER;

    -- Overload preceding public subprogram:
    PROCEDURE fire_employee (emp_id NUMBER);
    PROCEDURE fire_employee (emp_email VARCHAR2);

    PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
    FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin;
/  
  -- Package body:

CREATE OR REPLACE PACKAGE BODY emp_admin AS
  number_hired NUMBER;  -- private variable, visible only in this package

  -- Define cursor declared in package specification:

  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

  -- Define subprograms declared in package specification:

  FUNCTION hire_employee (last_name VARCHAR2,
                           first_name VARCHAR2,
                           email VARCHAR2,
                           phone_number VARCHAR2,
                           job_id VARCHAR2,
                           salary NUMBER,
                           commission_pct NUMBER,
                           manager_id NUMBER,
                           department_id NUMBER)
  RETURN NUMBER IS
    new_emp_id NUMBER;
    BEGIN
      new_emp_id := employees_seq.NEXTVAL;
      INSERT INTO employees (employee_id,
                              last_name,
                              first_name,
                              email,
                              phone_number,
                              hire_date,
                              job_id,
                              salary,
                              commission_pct,
                              manager_id,
                              department_id)
      VALUES (new_emp_id,
               hire_employee.last_name,
               hire_employee.first_name,
               hire_employee.email,
               hire_employee.phone_number,
               SYSDATE,
               hire_employee.job_id,
               hire_employee.salary,
               hire_employee.commission_pct,
               hire_employee.manager_id,
               hire_employee.department_id);
      number_hired := number_hired + 1;
      DBMS_OUTPUT.PUT_LINE('The number of employees hired is ' || TO_CHAR(number_hired) );
      RETURN new_emp_id;
    END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
  DELETE FROM employees WHERE employee_id = emp_id;
END fire_employee;

PROCEDURE fire_employee (emp_email VARCHAR2) IS
BEGIN
  DELETE FROM employees WHERE email = emp_email;
END fire_employee;

-- Define private function, available only inside package:

FUNCTION sal_ok (jobid VARCHAR2, sal NUMBER) RETURN BOOLEAN IS
  min_sal NUMBER;
  max_sal NUMBER;
BEGIN
  SELECT MIN(salary), MAX(salary)
  INTO min_sal, max_sal
  FROM employees
  WHERE job_id = jobid;

  RETURN (sal >= min_sal) AND (sal <= max_sal);
END sal_ok;

PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
  sal NUMBER(8,2);
  jobid VARCHAR2(10);
BEGIN
  SELECT job_id, salary INTO jobid, sal
  FROM employees
  WHERE employee_id = emp_id;

  IF sal_ok(jobid, sal + amount) THEN -- Invoke private function
    UPDATE employees
    SET salary = salary + amount
    WHERE employee_id = emp_id;
  ELSE
    RAISE invalid_salary;
  END IF;
EXCEPTION
  WHEN invalid_salary THEN
    DBMS_OUTPUT.PUT_LINE ('The salary is out of the specified range.');
END raise_salary;

FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp IS
  emp_rec  EmpRecTyp;
BEGIN
  OPEN desc_salary;
  FOR i IN 1..n LOOP
11.10 How STANDARD Package Defines the PL/SQL Environment

A package named STANDARD defines the PL/SQL environment. The package specification declares public types, variables, exceptions, subprograms, which are available automatically to PL/SQL programs. For example, package STANDARD declares function ABS, which returns the absolute value of its argument, as follows:

FUNCTION ABS (n NUMBER) RETURN NUMBER;

Chapter 11
How STANDARD Package Defines the PL/SQL Environment

11-18
The contents of package STANDARD are directly visible to applications. You need not qualify references to its contents by prefixing the package name. For example, you might invoke ABS from a database trigger, stored subprogram, Oracle tool, or 3GL application, as follows:

```
abs_diff := ABS(x - y);
```

If you declare your own version of ABS, your local declaration overrides the public declaration. You can still invoke the SQL function by specifying its full name:

```
abs_diff := STANDARD.ABS(x - y);
```

Most SQL functions are overloaded. For example, package STANDARD contains these declarations:

```sql
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves an invocation of TO_CHAR by matching the number and data types of the formal and actual parameters.
This chapter explains how to handle PL/SQL compile-time warnings and PL/SQL runtime errors. The latter are called exceptions.

**Note:**

The language of warning and error messages depends on the NLS\_LANGUAGE parameter. For information about this parameter, see Oracle Database Globalization Support Guide.

**Topics**

- Compile-Time Warnings
- Overview of Exception Handling
- Internally Defined Exceptions
- Predefined Exceptions
- User-Defined Exceptions
- Redeclared Predefined Exceptions
- Raising Exceptions Explicitly
- Exception Propagation
- Unhandled Exceptions
- Retrieving Error Code and Error Message
- Continuing Execution After Handling Exceptions
- Retrying Transactions After Handling Exceptions
- Handling Errors in Distributed Queries

**See Also:**

- "Exception Handling in Triggers"
- "Handling FORALL Exceptions After FORALL Statement Completes"
### 12.1 Compile-Time Warnings

While compiling stored PL/SQL units, the PL/SQL compiler generates warnings for conditions that are not serious enough to cause errors and prevent compilation—for example, using a deprecated PL/SQL feature.

To see warnings (and errors) generated during compilation, either query the static data dictionary view `*ERRORS` or, in the SQL*Plus environment, use the command `SHOW ERRORS`.

The message code of a PL/SQL warning has the form PLW-nnnnn.

#### Table 12-1 Compile-Time Warning Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEVERE</td>
<td>Condition might cause unexpected action or wrong results.</td>
<td>Aliasing problems with parameters</td>
</tr>
<tr>
<td>PERFORMANCE</td>
<td>Condition might cause performance problems.</td>
<td>Passing a VARCHAR2 value to a NUMBER column in an INSERT statement</td>
</tr>
<tr>
<td>INFORMATIONAL</td>
<td>Condition does not affect performance or correctness, but you might want to change it to make the code more maintainable.</td>
<td>Code that can never run</td>
</tr>
</tbody>
</table>

By setting the compilation parameter `PLSQL_WARNINGS`, you can:

- Enable and disable all warnings, one or more categories of warnings, or specific warnings
- Treat specific warnings as errors (so that those conditions must be corrected before you can compile the PL/SQL unit)

You can set the value of `PLSQL_WARNINGS` for:

- Your Oracle database instance
  - Use the `ALTER SYSTEM` statement, described in *Oracle Database SQL Language Reference*.
- Your session
  - Use the `ALTER SESSION` statement, described in *Oracle Database SQL Language Reference*. 
• A stored PL/SQL unit

Use an ALTER statement from "ALTER Statements" with its compiler_parameters_clause.

In any of the preceding ALTER statements, you set the value of PLSQL_WARNINGS with this syntax:

PLSQL_WARNINGS = 'value_clause' [, 'value_clause' ] ...

For the syntax of value_clause, see Oracle Database Reference.

To display the current value of PLSQL_WARNINGS, query the static data dictionary view ALL_PLSQL_OBJECT_SETTINGS.

See Also:

• Oracle Database Reference for more information about the static data dictionary view ALL_PLSQL_OBJECT_SETTINGS
• Oracle Database Error Messages Reference for the message codes of all PL/SQL warnings
• Oracle Database Reference for more information about the static data dictionary view *_ERRORS
• “PL/SQL Units and Compilation Parameters” for more information about PL/SQL units and compiler parameters

Example 12-1 Setting Value of PLSQL_WARNINGS Compilation Parameter

This example shows several ALTER statements that set the value of PLSQL_WARNINGS.

For the session, enable all warnings—highly recommended during development:

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';

For the session, enable PERFORMANCE warnings:

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';

For the procedure loc_var, enable PERFORMANCE warnings, and reuse settings:

ALTER PROCEDURE loc_var
   COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'
   REUSE SETTINGS;

For the session, enable SEVERE warnings, disable PERFORMANCE warnings, and treat PLW-06002 warnings as errors:

ALTER SESSION
   SET PLSQL_WARNINGS='ENABLE:SEVERE', 'DISABLE:PERFORMANCE', 'ERROR:06002';

For the session, disable all warnings:

ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
12.1.1 DBMS_WARNING Package

If you are writing PL/SQL units in a development environment that compiles them (such as SQL*Plus), you can display and set the value of PLSQL_WARNINGS by invoking subprograms in the DBMS_WARNING package.

Example 12-2 uses an ALTER SESSION statement to disable all warning messages for the session and then compiles a procedure that has unreachable code. The procedure compiles without warnings. Next, the example enables all warnings for the session by invoking DBMS_WARNING.set_warning_setting_string and displays the value of PLSQL_WARNINGS by invoking DBMS_WARNING.get_warning_setting_string. Finally, the example recompiles the procedure, and the compiler generates a warning about the unreachable code.

Note:
Unreachable code could represent a mistake or be intentionally hidden by a debug flag.

DBMS_WARNING subprograms are useful when you are compiling a complex application composed of several nested SQL*Plus scripts, where different subprograms need different PLSQL_WARNINGS settings. With DBMS_WARNING subprograms, you can save the current PLSQL_WARNINGS setting, change the setting to compile a particular set of subprograms, and then restore the setting to its original value.

See Also:
Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_WARNING package

Example 12-2 Displaying and Setting PLSQL_WARNINGS with DBMS_WARNING Subprograms

Disable all warning messages for this session:
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';

With warnings disabled, this procedure compiles with no warnings:

CREATE OR REPLACE PROCEDURE unreachable_code AUTHID DEFINER AS
  x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/

Enable all warning messages for this session:
CALL DBMS_WARNING.set_warning_setting_string ('ENABLE:ALL', 'SESSION');

Check warning setting:

SELECT DBMS_WARNING.get_warning_setting_string() FROM DUAL;

Result:

DBMS_WARNING.GET_WARNING_SETTING_STRING()
-----------------------------------------
ENABLE:ALL

1 row selected.

Recompile procedure:

ALTER PROCEDURE unreachable_code COMPILE;

Result:

SP2-0805: Procedure altered with compilation warnings

Show errors:

SHOW ERRORS

Result:

Errors for PROCEDURE UNREACHABLE_CODE:

LINE/COL  ERROR
-------------------------------
7/5      PLW-06002: Unreachable code

12.2 Overview of Exception Handling

Exceptions (PL/SQL runtime errors) can arise from design faults, coding mistakes, hardware failures, and many other sources. You cannot anticipate all possible exceptions, but you can write exception handlers that let your program to continue to operate in their presence.

Any PL/SQL block can have an exception-handling part, which can have one or more exception handlers. For example, an exception-handling part could have this syntax:

```
EXCEPTION
    WHEN ex_name_1 THEN statements_1    -- Exception handler
    WHEN ex_name_2 OR ex_name_3 THEN statements_2  -- Exception handler
    WHEN OTHERS THEN statements_3        -- Exception handler
END;
```

In the preceding syntax example, `ex_name_n` is the name of an exception and `statements_n` is one or more statements. (For complete syntax and semantics, see "Exception Handler".)

When an exception is raised in the executable part of the block, the executable part stops and control transfers to the exception-handling part. If `ex_name_1` was raised, then `statements_1` run. If either `ex_name_2` or `ex_name_3` was raised, then `statements_2` run. If any other exception was raised, then `statements_3` run.
After an exception handler runs, control transfers to the next statement of the enclosing block. If there is no enclosing block, then:

- If the exception handler is in a subprogram, then control returns to the invoker, at the statement after the invocation.
- If the exception handler is in an anonymous block, then control transfers to the host environment (for example, SQL*Plus)

If an exception is raised in a block that has no exception handler for it, then the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a block has a handler for it or there is no enclosing block (for more information, see "Exception Propagation"). If there is no handler for the exception, then PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome (for more information, see "Unhandled Exceptions").

Topics

- Exception Categories
- Advantages of Exception Handlers
- Guidelines for Avoiding and Handling Exceptions

12.2.1 Exception Categories

The exception categories are:

- **Internally defined**
  The runtime system raises internally defined exceptions implicitly (automatically). Examples of internally defined exceptions are ORA-00060 (deadlock detected while waiting for resource) and ORA-27102 (out of memory).
  An internally defined exception always has an error code, but does not have a name unless PL/SQL gives it one or you give it one.
  For more information, see "Internally Defined Exceptions".

- **Predefined**
  A predefined exception is an internally defined exception that PL/SQL has given a name. For example, ORA-06500 (PL/SQL: storage error) has the predefined name STORAGE_ERROR.
  For more information, see "Predefined Exceptions".

- **User-defined**
  You can declare your own exceptions in the declarative part of any PL/SQL anonymous block, subprogram, or package. For example, you might declare an exception named insufficient_funds to flag overdrawn bank accounts.
  You must raise user-defined exceptions explicitly.
  For more information, see "User-Defined Exceptions".

Table 12-2 summarizes the exception categories.
Table 12-2  Exception Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Definer</th>
<th>Has Error Code</th>
<th>Has Name</th>
<th>Raised Implicitly</th>
<th>Raised Explicitly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internally defined</td>
<td>Runtime system</td>
<td>Always</td>
<td>Only if you assign one</td>
<td>Yes</td>
<td>Optionally[^1]</td>
</tr>
<tr>
<td>Predefined</td>
<td>Runtime system</td>
<td>Always</td>
<td>Always</td>
<td>Yes</td>
<td>Optionally[^1]</td>
</tr>
<tr>
<td>User-defined</td>
<td>User</td>
<td>Only if you assign one</td>
<td>Always</td>
<td>No</td>
<td>Always</td>
</tr>
</tbody>
</table>

[^1]: For details, see "Raising Internally Defined Exception with RAISE Statement".

For a named exception, you can write a specific exception handler, instead of handling it with an OTHERS exception handler. A specific exception handler is more efficient than an OTHERS exception handler, because the latter must invoke a function to determine which exception it is handling. For details, see "Retrieving Error Code and Error Message".

12.2.2 Advantages of Exception Handlers

Using exception handlers for error-handling makes programs easier to write and understand, and reduces the likelihood of unhandled exceptions.

Without exception handlers, you must check for every possible error, everywhere that it might occur, and then handle it. It is easy to overlook a possible error or a place where it might occur, especially if the error is not immediately detectable (for example, bad data might be undetectable until you use it in a calculation). Error-handling code is scattered throughout the program.

With exception handlers, you need not know every possible error or everywhere that it might occur. You need only include an exception-handling part in each block where errors might occur. In the exception-handling part, you can include exception handlers for both specific and unknown errors. If an error occurs anywhere in the block (including inside a sub-block), then an exception handler handles it. Error-handling code is isolated in the exception-handling parts of the blocks.

In Example 12-3, a procedure uses a single exception handler to handle the predefined exception `NO_DATA_FOUND`, which can occur in either of two `SELECT INTO` statements.

If multiple statements use the same exception handler, and you want to know which statement failed, you can use locator variables, as in Example 12-4.

You determine the precision of your error-handling code. You can have a single exception handler for all division-by-zero errors, bad array indexes, and so on.
can also check for errors in a single statement by putting that statement inside a block with its own exception handler.

**Example 12-3  Single Exception Handler for Multiple Exceptions**

```sql
CREATE OR REPLACE PROCEDURE select_item (  
t_column VARCHAR2,
  t_name   VARCHAR2  
) AUTHID DEFINER  
IS  
temp VARCHAR2(30);  
BEGIN  
temp := t_column; -- For error message if next SELECT fails  
  -- Fails if table t_name does not have column t_column:  
  SELECT COLUMN_NAME INTO temp  
  FROM USER_TAB_COLS  
  WHERE TABLE_NAME = UPPER(t_name)  
  AND COLUMN_NAME = UPPER(t_column);  
  temp := t_name; -- For error message if next SELECT fails  
  -- Fails if there is no table named t_name:  
  SELECT OBJECT_NAME INTO temp  
  FROM USER_OBJECTS  
  WHERE OBJECT_NAME = UPPER(t_name)  
  AND OBJECT_TYPE = 'TABLE';  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN    
    DBMS_OUTPUT.PUT_LINE ('No Data found for SELECT on ' || temp);  
  WHEN OTHERS THEN    
    DBMS_OUTPUT.PUT_LINE ('Unexpected error');    
    RAISE;  
END;  
/
```

Invoke procedure (there is a **DEPARTMENTS** table, but it does not have a **LAST_NAME** column):

```sql
BEGIN  
  select_item('departments', 'last_name');  
END;  
/
```

Result:

No Data found for SELECT on departments

Invoke procedure (there is no **EMP** table):

```sql
BEGIN  
  select_item('emp', 'last_name');  
END;  
/
```

Result:

No Data found for SELECT on emp
Example 12-4  Locator Variables for Statements that Share Exception Handler

CREATE OR REPLACE PROCEDURE loc_var AUTHID DEFINER IS
    stmt_no  POSITIVE;
    name_    VARCHAR2(100);
BEGIN
    stmt_no := 1;
    SELECT table_name INTO name_
        FROM user_tables
        WHERE table_name LIKE 'ABC%';

    stmt_no := 2;
    SELECT table_name INTO name_
        FROM user_tables
        WHERE table_name LIKE 'XYZ%';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('Table name not found in query ' || stmt_no);
END;
/
CALL loc_var();

Result:

Table name not found in query 1

12.2.3 Guidelines for Avoiding and Handling Exceptions

To make your programs as reliable and safe as possible:

- Use both error-checking code and exception handlers.
  Use error-checking code wherever bad input data can cause an error. Examples of bad input data are incorrect or null actual parameters and queries that return no rows or more rows than you expect. Test your code with different combinations of bad input data to see what potential errors arise.

  Sometimes you can use error-checking code to avoid raising an exception, as in Example 12-7.

- Add exception handlers wherever errors can occur.
  Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors can also arise from problems that are independent of your code—for example, disk storage or memory hardware failure—but your code still must take corrective action.

- Design your programs to work when the database is not in the state you expect.
  For example, a table you query might have columns added or deleted, or their types might have changed. You can avoid problems by declaring scalar variables with %TYPE qualifiers and record variables to hold query results with %ROWTYPE qualifiers.

- Whenever possible, write exception handlers for named exceptions instead of using OTHERS exception handlers.
  Learn the names and causes of the predefined exceptions. If you know that your database operations might raise specific internally defined exceptions that do not
have names, then give them names so that you can write exception handlers specifically for them.

- Have your exception handlers output debugging information.

If you store the debugging information in a separate table, do it with an autonomous routine, so that you can commit your debugging information even if you roll back the work that the main subprogram did. For information about autonomous routines, see "AUTONOMOUS_TRANSACTION Pragma".

- For each exception handler, carefully decide whether to have it commit the transaction, roll it back, or let it continue.

Regardless of the severity of the error, you want to leave the database in a consistent state and avoid storing bad data.

- Avoid unhandled exceptions by including an OTHERS exception handler at the top level of every PL/SQL program.

Make the last statement in the OTHERS exception handler either RAISE or an invocation of of a subroutine marked with SUPPRESSES_WARNING_6009 pragma. (If you do not follow this practice, and PL/SQL warnings are enabled, then you get PLW-06009.) For information about RAISE or an invocation of the RAISE_APPLICATION_ERROR, see "Raising Exceptions Explicitly".

12.3 Internally Defined Exceptions

**Internally defined exceptions** (ORA-n errors) are described in *Oracle Database Error Messages Reference*. The runtime system raises them implicitly (automatically).

An internally defined exception does not have a name unless either PL/SQL gives it one (see "Predefined Exceptions") or you give it one.

If you know that your database operations might raise specific internally defined exceptions that do not have names, then give them names so that you can write exception handlers specifically for them. Otherwise, you can handle them only with OTHERS exception handlers.

To give a name to an internally defined exception, do the following in the declarative part of the appropriate anonymous block, subprogram, or package. (To determine the appropriate block, see "Exception Propagation").

1. Declare the name.

   An exception name declaration has this syntax:

   ```plsql
   exception_name EXCEPTION;
   ```

   For semantic information, see "Exception Declaration".

2. Associate the name with the error code of the internally defined exception.

   The syntax is:

   ```plsql
   PRAGMA EXCEPTION_INIT (exception_name, error_code)
   ```

   For semantic information, see "EXCEPTION_INIT Pragma".
Note:

An internally defined exception with a user-declared name is still an internally defined exception, not a user-defined exception.

Example 12-5 gives the name `deadlock_detected` to the internally defined exception ORA-00060 (deadlock detected while waiting for resource) and uses the name in an exception handler.

See Also:

"Raising Internally Defined Exception with RAISE Statement"

Example 12-5  Naming Internally Defined Exception

```plsql
DECLARE
    deadlock_detected EXCEPTION;
    PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
    ...
    EXCEPTION
        WHEN deadlock_detected THEN
            ...
END;
/```

12.4 Predefined Exceptions

Predefined exceptions are internally defined exceptions that have predefined names, which PL/SQL declares globally in the package `STANDARD`. The runtime system raises predefined exceptions implicitly (automatically). Because predefined exceptions have names, you can write exception handlers specifically for them.

Table 12-3 lists the names and error codes of the predefined exceptions.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS INTO NULL</td>
<td>-6530</td>
</tr>
<tr>
<td>CASE NOT FOUND</td>
<td>-6592</td>
</tr>
<tr>
<td>COLLECTION IS NULL</td>
<td>-6531</td>
</tr>
<tr>
<td>CURSOR ALREADY OPEN</td>
<td>-6511</td>
</tr>
<tr>
<td>DUP VAL ON INDEX</td>
<td>-1</td>
</tr>
<tr>
<td>INVALID_CURSOR</td>
<td>-1001</td>
</tr>
<tr>
<td>INVALID NUMBER</td>
<td>-1722</td>
</tr>
<tr>
<td>LOGIN_DENIED</td>
<td>-1017</td>
</tr>
</tbody>
</table>
Table 12-3  (Cont.) PL/SQL Predefined Exceptions

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_DATA_FOUND</td>
<td>+100</td>
</tr>
<tr>
<td>NO_DATA_NEEDED</td>
<td>-6548</td>
</tr>
<tr>
<td>NOT_LOGGED_ON</td>
<td>-1012</td>
</tr>
<tr>
<td>PROGRAM_ERROR</td>
<td>-6501</td>
</tr>
<tr>
<td>ROWTYPE_MISMATCH</td>
<td>-6504</td>
</tr>
<tr>
<td>SELF_IS_NULL</td>
<td>-30625</td>
</tr>
<tr>
<td>STORAGE_ERROR</td>
<td>-6500</td>
</tr>
<tr>
<td>SUBSCRIPT_BEYOND_COUNT</td>
<td>-6533</td>
</tr>
<tr>
<td>SUBSCRIPT_OUTSIDE_LIMIT</td>
<td>-6532</td>
</tr>
<tr>
<td>SYS_INVALID_ROWID</td>
<td>-1410</td>
</tr>
<tr>
<td>TIMEOUT_ON_RESOURCE</td>
<td>-51</td>
</tr>
<tr>
<td>TOO_MANY_ROWS</td>
<td>-1422</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>-6502</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>-1476</td>
</tr>
</tbody>
</table>

Example 12-6 calculates a price-to-earnings ratio for a company. If the company has zero earnings, the division operation raises the predefined exception ZERO_DIVIDE and the executable part of the block transfers control to the exception-handling part.

Example 12-7 uses error-checking code to avoid the exception that Example 12-6 handles.

In Example 12-8, the procedure opens a cursor variable for either the EMPLOYEES table or the DEPARTMENTS table, depending on the value of the parameter discrim. The anonymous block invokes the procedure to open the cursor variable for the EMPLOYEES table, but fetches from the DEPARTMENTS table, which raises the predefined exception ROWTYPE_MISMATCH.

See Also:
"Raising Internally Defined Exception with RAISE Statement"

Example 12-6  Anonymous Block Handles ZERO_DIVIDE

DECLARE
    stock_price NUMBER := 9.73;
    net_earnings NUMBER := 0;
    pe_ratio NUMBER;
BEGIN
    pe_ratio := stock_price / net_earnings;  -- raises ZERO_DIVIDE exception
    DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
DBMS_OUTPUT.PUT_LINE('Company had zero earnings.');  
pe_ratio := NULL;  
END;  
/  

Result:  
Company had zero earnings.

**Example 12-7  Anonymous Block Avoids ZERO_DIVIDE**

DECLARE  
stock_price   NUMBER := 9.73;  
net_earnings  NUMBER := 0;  
pe_ratio      NUMBER;  
BEGIN  
pe_ratio :=  
    CASE net_earnings  
        WHEN 0 THEN NULL  
        ELSE stock_price / net_earnings  
    END;  
END;  
/  

**Example 12-8  Anonymous Block Handles ROWTYPE_MISMATCH**

CREATE OR REPLACE PACKAGE emp_dept_data AUTHID DEFINER AS  
TYPE cv_type IS REF CURSOR;  
PROCEDURE open_cv (  
cv IN OUT cv_type,  
discrim IN     POSITIVE  
);  
END emp_dept_data;  
/  
CREATE OR REPLACE PACKAGE BODY emp_dept_data AS  
PROCEDURE open_cv (  
cv IN OUT cv_type,  
discrim IN     POSITIVE) IS  
BEGIN  
    IF discrim = 1 THEN  
        OPEN cv FOR  
            SELECT * FROM EMPLOYEES ORDER BY employee_id;  
    ELSIF discrim = 2 THEN  
        OPEN cv FOR  
            SELECT * FROM DEPARTMENTS ORDER BY department_id;  
    END IF;  
END open_cv;  
END emp_dept_data;  
/  

Invoke procedure open_cv from anonymous block:

DECLARE  
emp_rec   EMPLOYEES%ROWTYPE;  
dep_rec   DEPARTMENTS%ROWTYPE;  
cv        Emp_dept_data.CV_TYPE;  
BEGIN  
    emp_dept_data.open_cv(cv, 1);  
        -- Open cv for EMPLOYEES fetch.  
FETCH cv INTO dept_rec;  
        -- Fetch from DEPARTMENTS.  
DBMS_OUTPUT.PUT(dept_rec.DEPARTMENT_ID);
DBMS_OUTPUT.PUT_LINE('  ' || dept_rec.LOCATION_ID);
EXCEPTION
  WHEN ROWTYPE_MISMATCH THEN
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Row type mismatch, fetching EMPLOYEES data ...');
      FETCH cv INTO emp_rec;
      DBMS_OUTPUT.PUT(emp_rec.DEPARTMENT_ID);
      DBMS_OUTPUT.PUT_LINE('  ' || emp_rec.LAST_NAME);
    END;
  END;
/

Result:
Row type mismatch, fetching EMPLOYEES data ...
  90  King

12.5 User-Defined Exceptions

You can declare your own exceptions in the declarative part of any PL/SQL anonymous block, subprogram, or package.

An exception name declaration has this syntax:

exception_name EXCEPTION;

For semantic information, see "Exception Declaration".

You must raise a user-defined exception explicitly. For details, see "Raising Exceptions Explicitly".

12.6 Redeclared Predefined Exceptions

Oracle recommends against redeclaring predefined exceptions—that is, declaring a user-defined exception name that is a predefined exception name. (For a list of predefined exception names, see Table 12-3.)

If you redeclare a predefined exception, your local declaration overrides the global declaration in package STANDARD. Exception handlers written for the globally declared exception become unable to handle it—unless you qualify its name with the package name STANDARD.

Example 12-9 shows this.

Example 12-9  Redeclared Predefined Identifier

DROP TABLE t;
CREATE TABLE t (c NUMBER);

In the following block, the INSERT statement implicitly raises the predefined exception INVALID_NUMBER, which the exception handler handles.

DECLARE
  default_number NUMBER := 0;
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/

Result:
Substituting default value for invalid number.

The following block redeclares the predefined exception INVALID_NUMBER. When the INSERT statement implicitly raises the predefined exception INVALID_NUMBER, the exception handler does not handle it.

DECLARE
    default_number NUMBER := 0;
    i NUMBER := 5;
    invalid_number EXCEPTION;    -- redeclare predefined exception
BEGIN
    INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
    WHEN INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
        INSERT INTO t VALUES(default_number);
END;
/

Result:
DECLARE
    *
    ERROR at line 1:
    ORA-01722: invalid number
    ORA-06512: at line 6

The exception handler in the preceding block handles the predefined exception INVALID_NUMBER if you qualify the exception name in the exception handler:

DECLARE
    default_number NUMBER := 0;
    i NUMBER := 5;
    invalid_number EXCEPTION;    -- redeclare predefined exception
BEGIN
    INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
    WHEN STANDARD.INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
        INSERT INTO t VALUES(default_number);
END;
/

Result:
Substituting default value for invalid number.
12.7 Raising Exceptions Explicitly

To raise an exception explicitly, use either the RAISE statement or RAISE_APPLICATION_ERROR procedure.

Topics
- RAISE Statement
- RAISE_APPLICATION_ERROR Procedure

12.7.1 RAISE Statement

The RAISE statement explicitly raises an exception. Outside an exception handler, you must specify the exception name. Inside an exception handler, if you omit the exception name, the RAISE statement reraises the current exception.

Topics
- Raising User-Defined Exception with RAISE Statement
- Raising Internally Defined Exception with RAISE Statement
- Reraising Current Exception with RAISE Statement

12.7.1.1 Raising User-Defined Exception with RAISE Statement

In Example 12-10, the procedure declares an exception named past_due, raises it explicitly with the RAISE statement, and handles it with an exception handler.

Example 12-10  Declaring, Raising, and Handling User-Defined Exception

```sql
CREATE PROCEDURE account_status (    due_date DATE,    today    DATE ) AUTHID DEFINER IS
  past_due  EXCEPTION;  -- declare exception
BEGIN
  IF due_date < today THEN
    RAISE past_due;  -- explicitly raise exception
  END IF;
EXCEPTION
  WHEN past_due THEN  -- handle exception
    DBMS_OUTPUT.PUT_LINE ('Account past due.');
  END;
/
BEGIN
  account_status (TO_DATE('01-JUL-2010', 'DD-MON-YYYY'),
                  TO_DATE('09-JUL-2010', 'DD-MON-YYYY'));
END;
/

Result:
Account past due.
```
12.7.1.2 Raising Internally Defined Exception with RAISE Statement

Although the runtime system raises internally defined exceptions implicitly, you can raise them explicitly with the RAISE statement if they have names. Table 12-3 lists the internally defined exceptions that have predefined names. "Internally Defined Exceptions" explains how to give user-declared names to internally defined exceptions.

An exception handler for a named internally defined exception handles that exception whether it is raised implicitly or explicitly.

In Example 12-11, the procedure raises the predefined exception INVALID_NUMBER either explicitly or implicitly, and the INVALID_NUMBER exception handler always handles it.

Example 12-11  Explicitly Raising Predefined Exception

```sql
DROP TABLE t;
CREATE TABLE t (c NUMBER);

CREATE PROCEDURE p (n NUMBER) AUTHID DEFINER IS
  default_number NUMBER := 0;
BEGIN
  IF n < 0 THEN
    RAISE INVALID_NUMBER;  -- raise explicitly
  ELSE
    INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));  -- raise implicitly
  END IF;
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/

BEGIN
  p(-1);
END;
/

Result:
Substituting default value for invalid number.

BEGIN
  p(1);
END;
/

Result:
Substituting default value for invalid number.
```

12.7.1.3 Reraising Current Exception with RAISE Statement

In an exception handler, you can use the RAISE statement to "reraise" the exception being handled. Reraising the exception passes it to the enclosing block, which can handle it further. (If the enclosing block cannot handle the reraised exception, then the
exception propagates—see “Exception Propagation”). When reraising the current exception, you need not specify an exception name.

In Example 12-12, the handling of the exception starts in the inner block and finishes in the outer block. The outer block declares the exception, so the exception name exists in both blocks, and each block has an exception handler specifically for that exception. The inner block raises the exception, and its exception handler does the initial handling and then reraises the exception, passing it to the outer block for further handling.

**Example 12-12  Reraising Exception**

```
DECLARE
  salary_too_high   EXCEPTION;
  current_salary    NUMBER := 20000;
  max_salary        NUMBER := 10000;
  erroneous_salary  NUMBER;
BEGIN
  BEGIN
    IF current_salary > max_salary THEN
      RAISE salary_too_high;   -- raise exception
    END IF;
  EXCEPTION
    WHEN salary_too_high THEN  -- start handling exception
      erroneous_salary := current_salary;
      DBMS_OUTPUT.PUT_LINE('Salary ' || erroneous_salary ||' is out of range.');
      DBMS_OUTPUT.PUT_LINE ('Maximum salary is ' || max_salary || '.');
      RAISE;  -- reraise current exception (exception name is optional)
  END;
  EXCEPTION
    WHEN salary_too_high THEN    -- finish handling exception
      current_salary := max_salary;
      DBMS_OUTPUT.PUT_LINE (  
        'Revising salary from ' || erroneous_salary || 
        ' to ' || current_salary || '.');
  END;
END;
/
```

**Result:**

Salary 20000 is out of range.
Maximum salary is 10000.
Revising salary from 20000 to 10000.

### 12.7.2 RAISE_APPLICATION_ERROR Procedure

You can invoke the `RAISE_APPLICATION_ERROR` procedure (defined in the `DBMS_STANDARD` package) only from a stored subprogram or method. Typically, you invoke this procedure to raise a user-defined exception and return its error code and error message to the invoker.

The `RAISE_APPLICATION_ERROR` procedure is marked with `SUPPRESSIONS_WARNING_6009` pragma.

For semantic information, see “SUPPRESSIONS_WARNING_6009 Pragma”.
To invoke RAISE_APPLICATION_ERROR, use this syntax:

RAISE_APPLICATION_ERROR (error_code, message[, [TRUE | FALSE]]);

You must have assigned error_code to the user-defined exception with the
EXCEPTION_INIT pragma. The syntax is:

PRAGMA EXCEPTION_INIT (exception_name, error_code)

The error_code is an integer in the range -20000..-20999 and the message is a
character string of at most 2048 bytes.

For semantic information, see "EXCEPTION_INIT Pragma".

The message is a character string of at most 2048 bytes.

If you specify TRUE, PL/SQL puts error_code on top of the error stack. Otherwise,
PL/SQL replaces the error stack with error_code.

In Example 12-13, an anonymous block declares an exception named past_due,
assigns the error code -20000 to it, and invokes a stored procedure. The stored
procedure invokes the RAISE_APPLICATION_ERROR procedure with the error code
-20000 and a message, whereupon control returns to the anonymous block, which
handles the exception. To retrieve the message associated with the exception, the
exception handler in the anonymous block invokes the SQLERRM function, described in
"Retrieving Error Code and Error Message".

**Example 12-13 Raising User-Defined Exception with RAISE_APPLICATION_ERROR**

```sql
CREATE PROCEDURE account_status (
    due_date DATE,
    today    DATE
) AUTHID DEFINER
IS
BEGIN
    IF due_date < today THEN -- explicitly raise exception
        RAISE_APPLICATION_ERROR(-20000, 'Account past due.');
    END IF;
END;
/

DECLARE
    past_due  EXCEPTION;                       -- declare exception
    PRAGMA EXCEPTION_INIT (past_due, -20000);  -- assign error code to exception
BEGIN
    account_status (TO_DATE('01-JUL-2010', 'DD-MON-YYYY'),
                    TO_DATE('09-JUL-2010', 'DD-MON-YYYY'));   -- invoke procedure
EXCEPTION
    WHEN past_due THEN -- handle exception
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(SQLERRM(-20000)));
END;
/
```

Result:

ORA-20000: Account past due.
12.8 Exception Propagation

If an exception is raised in a block that has no exception handler for it, then the exception **propagates**. That is, the exception reproduces itself in successive enclosing blocks until either a block has a handler for it or there is no enclosing block. If there is no handler for the exception, then PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome (for more information, see "Unhandled Exceptions").

In Figure 12-1, one block is nested inside another. The inner block raises exception A. The inner block has an exception handler for A, so A does not propagate. After the exception handler runs, control transfers to the next statement of the outer block.

Figure 12-1  Exception Does Not Propagate

In Figure 12-2, the inner block raises exception B. The inner block does not have an exception handler for exception B, so B propagates to the outer block, which does have an exception handler for it. After the exception handler runs, control transfers to the host environment.
In Figure 12-3, the inner block raises exception C. The inner block does not have an exception handler for C, so exception C propagates to the outer block. The outer block does not have an exception handler for C, so PL/SQL returns an unhandled exception error to the host environment.

A user-defined exception can propagate beyond its scope (that is, beyond the block that declares it), but its name does not exist beyond its scope. Therefore, beyond its scope, a user-defined exception can be handled only with an `OTHERS` exception handler.
In Example 12-14, the inner block declares an exception named \texttt{past\_due}, for which it has no exception handler. When the inner block raises \texttt{past\_due}, the exception propagates to the outer block, where the name \texttt{past\_due} does not exist. The outer block handles the exception with an \texttt{OTHERS} exception handler.

If the outer block does not handle the user-defined exception, then an error occurs, as in Example 12-15.

\begin{quote}
\textbf{Note:}

Exceptions cannot propagate across remote subprogram invocations. Therefore, a PL/SQL block cannot handle an exception raised by a remote subprogram.
\end{quote}

\textbf{Topics}

- Propagation of Exceptions Raised in Declarations
- Propagation of Exceptions Raised in Exception Handlers

\textbf{Example 12-14 Exception that Propagates Beyond Scope is Handled}

\begin{verbatim}
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
    BEGIN
    DECLARE
        past_due     EXCEPTION;
        PRAGMA EXCEPTION_INIT (past_due, -4910);
        due_date     DATE := trunc(SYSDATE) - 1;
        todays_date  DATE := trunc(SYSDATE);
    BEGIN
        IF due_date < todays_date THEN
            RAISE past_due;
        END IF;
    END;
    EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        RAISE;
    END;
END;
/ 
\end{verbatim}

\textbf{Example 12-15 Exception that Propagates Beyond Scope is Not Handled}

\begin{verbatim}
BEGIN
    DECLARE
        past_due     EXCEPTION;
        due_date     DATE := trunc(SYSDATE) - 1;
        todays_date  DATE := trunc(SYSDATE);
    BEGIN
        IF due_date < todays_date THEN
            RAISE past_due;
        END IF;
    END;
END;
\end{verbatim}
12.8.1 Propagation of Exceptions Raised in Declarations

An exception raised in a declaration propagates immediately to the enclosing block (or to the invoker or host environment if there is no enclosing block). Therefore, the exception handler must be in an enclosing or invoking block, not in the same block as the declaration.

In Example 12-16, the VALUE_ERROR exception handler is in the same block as the declaration that raises VALUE_ERROR. Because the exception propagates immediately to the host environment, the exception handler does not handle it.

Example 12-17 is like Example 12-16 except that an enclosing block handles the VALUE_ERROR exception that the declaration in the inner block raises.

**Example 12-16 Exception Raised in Declaration is Not Handled**

```sql
DECLARE
    credit_limit CONSTANT NUMBER(3) := 5000;  -- Maximum value is 999
BEGIN
    NULL;
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```

**Result:**

```
DECLARE
    credit_limit CONSTANT NUMBER(3) := 5000;  -- Maximum value is 999
BEGIN
    NULL;
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```

**Example 12-17 Exception Raised in Declaration is Handled by Enclosing Block**

```sql
BEGIN

    DECLARE
        credit_limit CONSTANT NUMBER(3) := 5000;
    BEGIN
        NULL;
    END;

EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```

**Result:**

```
DECLARE
    credit_limit CONSTANT NUMBER(3) := 5000;
BEGIN
    NULL;
END;

EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```
12.8.2 Propagation of Exceptions Raised in Exception Handlers

An exception raised in an exception handler propagates immediately to the enclosing block (or to the invoker or host environment if there is no enclosing block). Therefore, the exception handler must be in an enclosing or invoking block.

In Example 12-18, when n is zero, the calculation \( \frac{1}{n} \) raises the predefined exception \( \text{ZERO
divide} \), and control transfers to the \( \text{ZERO
divide} \) exception handler in the same block. When the exception handler raises \( \text{ZERO
divide} \), the exception propagates immediately to the invoker. The invoker does not handle the exception, so PL/SQL returns an unhandled exception error to the host environment.

Example 12-19 is like Example 12-18 except that when the procedure returns an unhandled exception error to the invoker, the invoker handles it.

Example 12-20 is like Example 12-18 except that an enclosing block handles the exception that the exception handler in the inner block raises.

In Example 12-21, the exception-handling part of the procedure has exception handlers for user-defined exception \( i\_\text{is}
\text{one} \) and predefined exception \( \text{ZERO
divide} \). When the \( i\_\text{is}
\text{one} \) exception handler raises \( \text{ZERO
divide} \), the exception propagates immediately to the invoker (therefore, the \( \text{ZERO
divide} \) exception handler does not handle it). The invoker does not handle the exception, so PL/SQL returns an unhandled exception error to the host environment.

Example 12-22 is like Example 12-21 except that an enclosing block handles the \( \text{ZERO
divide} \) exception that the \( i\_\text{is}
\text{one} \) exception handler raises.

**Example 12-18 Exception Raised in Exception Handler is Not Handled**

```sql
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(1/n);  -- handled
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error:');
      DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');  -- not handled
  END;
/

BEGIN  -- invoking block
  print_reciprocal(0);
END;
```

**Result:**

```
Exception raised in declaration.
```

```sql
Error:
BEGIN
  *
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "HR.PRINT_RECIPROCAL", line 7
ORA-01476: divisor is equal to zero
ORA-06512: at line 2
```
**Example 12-19 Exception Raised in Exception Handler is Handled by Invoker**

CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(1/n);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error: ');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
END;
/

BEGIN -- invoking block
  print_reciprocal(0);
EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT_LINE('Error: 1/0 is undefined.');
END;
/

Result:
Error:
1/0 is undefined.

**Example 12-20 Exception Raised in Exception Handler is Handled by Enclosing Block**

CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN
  BEGIN
    DBMS_OUTPUT.PUT_LINE(1/n);
  EXCEPTION
    WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
      DBMS_OUTPUT.PUT_LINE('Error in inner block: ');
      DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined.');
  END;

  EXCEPTION
    WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
      DBMS_OUTPUT.PUT('Error in outer block: ');
      DBMS_OUTPUT.PUT_LINE('1/0 is undefined.');
  END;
/

BEGIN
  print_reciprocal(0);
END;
/

Result:
Error in inner block:
Error in outer block: 1/0 is undefined.

**Example 12-21 Exception Raised in Exception Handler is Not Handled**

CREATE PROCEDURE descending_reciprocals (n INTEGER) AUTHID DEFINER IS
  i INTEGER;
i_is_one EXCEPTION;
BEGIN
i := n;

LOOP
  IF i = 1 THEN
    RAISE i_is_one;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || i || ' is ' || 1/i);
  END IF;
  i := i - 1;
END LOOP;

EXCEPTION
  WHEN i_is_one THEN
    DBMS_OUTPUT.PUT_LINE('1 is its own reciprocal. ');
    DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || TO_CHAR(i-1) || ' is ' || 1/(i-1));
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error: ');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
END;
/

BEGIN
  descending_reciprocals(3);
END;
/

Result:
Reciprocal of 3 is .3333333333333333333333333333333333333333
Reciprocal of 2 is .5
1 is its own reciprocal.
BEGIN
  * 
  ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "HR.DESCENDING_RECIPROCALS", line 19
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 2

Example 12-22    Exception Raised in Exception Handler is Handled by Enclosing Block

CREATE PROCEDURE descending_reciprocals (n INTEGER) AUTHID DEFINER IS
  i INTEGER;
  i_is_one EXCEPTION;
BEGIN
  BEGIN
    i := n;
    LOOP
      IF i = 1 THEN
        RAISE i_is_one;
      ELSE
        DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || i || ' is ' || 1/i);
      END IF;
      i := i - 1;
    END LOOP;
  END IF;
  DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || TO_CHAR(i-1) || ' is ' || 1/(i-1));
END;
/
BEGIN
   descending_reciprocals(3);
END;
/

Result:
Reciprocal of 3 is .33333333333333333
Reciprocal of 2 is .5
1 is its own reciprocal.
Error:
1/0 is undefined

12.9 Unhandled Exceptions

If there is no handler for a raised exception, PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome.

If a stored subprogram exits with an unhandled exception, PL/SQL does not roll back database changes made by the subprogram.

The FORALL statement runs one DML statement multiple times, with different values in the VALUES and WHERE clauses. If one set of values raises an unhandled exception, then PL/SQL rolls back all database changes made earlier in the FORALL statement. For more information, see “Handling FORALL Exceptions Immediately” and “Handling FORALL Exceptions After FORALL Statement Completes”.

Tip:

Avoid unhandled exceptions by including an OTHERS exception handler at the top level of every PL/SQL program.

12.10 Retrieving Error Code and Error Message

In an exception handler, for the exception being handled:
• You can retrieve the error code with the PL/SQL function SQLCODE, described in "SQLCODE Function”.

• You can retrieve the error message with either:
  – The PL/SQL function SQLERRM, described in "SQLERRM Function”
    This function returns a maximum of 512 bytes, which is the maximum length of an Oracle Database error message (including the error code, nested messages, and message inserts such as table and column names).
  – The package function DBMS_UTILITY.Format_Error_Stack, described in Oracle Database PL/SQL Packages and Types Reference
    This function returns the full error stack, up to 2000 bytes.

Oracle recommends using DBMS_UTILITY.Format_Error_Stack, except when using the FORALL statement with its SAVE EXCEPTIONS clause, as in Example 13-13.

A SQL statement cannot invoke SQLCODE or SQLERRM. To use their values in a SQL statement, assign them to local variables first, as in Example 12-23.

See Also:

• Oracle Database PL/SQL Packages and Types Reference for information about the DBMS_UTILITY.Format_Error_Backtrace function, which displays the call stack at the point where an exception was raised, even if the subprogram is called from an exception handler in an outer scope.

• Oracle Database PL/SQL Packages and Types Reference for information about the UTL_CALL_STACK package, whose subprograms provide information about currently executing subprograms, including subprogram names.

Example 12-23  Displaying SQLCODE and SQLERRM Values

DROP TABLE errors;
CREATE TABLE errors (  
    code NUMBER,  
    message VARCHAR2(64)  
);

CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS  
    name EMPLOYEES.LAST_NAME%TYPE;  
    v_code NUMBER;  
    v_errm VARCHAR2(64);  
BEGIN  
    SELECT last_name INTO name  
    FROM EMPLOYEES  
    WHERE EMPLOYEE_ID = -1;  
    EXCEPTION  
    WHEN OTHERS THEN  
        v_code := SQLCODE;  
        v_errm := SUBSTR(SQLERRM, 1, 64);  
        DBMS_OUTPUT.PUT_LINE  
            ("Error code ' || v_code || ': ' || v_errm);


/ * Invoke another procedure, 
  declared with PRAGMA AUTONOMOUS_TRANSACTION, 
  to insert information about errors. */

  INSERT INTO errors (code, message) 
  VALUES (v_code, v_errm);

  RAISE;
END;
/

12.11 Continuing Execution After Handling Exceptions

After an exception handler runs, control transfers to the next statement of the 
enclosing block (or to the invoker or host environment if there is no enclosing block). 
The exception handler cannot transfer control back to its own block.

For example, in Example 12-24, after the SELECT INTO statement raises ZERO_DIVIDE 
and the exception handler handles it, execution cannot continue from the INSERT 
statement that follows the SELECT INTO statement.

If you want execution to resume with the INSERT statement that follows the SELECT INTO statement, then put the SELECT INTO statement in an inner block with its own 
ZERO_DIVIDE exception handler, as in Example 12-25.

See Also:

Example 13-13, where a bulk SQL operation continues despite exceptions

Example 12-24 Exception Handler Runs and Execution Ends

DROP TABLE employees_temp;
CREATE TABLE employees_temp AS 
  SELECT employee_id, salary, commission_pct 
  FROM employees;

DECLARE 
  sal_calc NUMBER(8,2);
BEGIN 
  INSERT INTO employees_temp (employee_id, salary, commission_pct) 
  VALUES (301, 2500, 0);

  SELECT (salary / commission_pct) INTO sal_calc 
  FROM employees_temp 
  WHERE employee_id = 301;

  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
  DBMS_OUTPUT.PUT_LINE('Row inserted.');
EXCEPTION 
  WHEN ZERO_DIVIDE THEN 
    DBMS_OUTPUT.PUT_LINE('Division by zero.');
END;
/

Result:
Division by zero.

**Example 12-25 Exception Handler Runs and Execution Continues**

```sql
DECLARE
    sal_calc NUMBER(8,2);
BEGIN
    INSERT INTO employees_temp (employee_id, salary, commission_pct)
    VALUES (301, 2500, 0);
BEGIN
    SELECT (salary / commission_pct) INTO sal_calc
    FROM employees_temp
    WHERE employee_id = 301;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Substituting 2500 for undefined number.');
        sal_calc := 2500;
END;
    INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
    DBMS_OUTPUT.PUT_LINE('Enclosing block: Row inserted.');
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Enclosing block: Division by zero.');
END;
/
```

Result:

Substituting 2500 for undefined number.
Enclosing block: Row inserted.

### 12.12 Retrying Transactions After Handling Exceptions

To retry a transaction after handling an exception that it raised, use this technique:

1. Enclose the transaction in a sub-block that has an exception-handling part.
2. In the sub-block, before the transaction starts, mark a savepoint.
3. In the exception-handling part of the sub-block, put an exception handler that rolls back to the savepoint and then tries to correct the problem.
4. Put the sub-block inside a `LOOP` statement.
5. In the sub-block, after the `COMMIT` statement that ends the transaction, put an `EXIT` statement.

   If the transaction succeeds, the `COMMIT` and `EXIT` statements execute.

   If the transaction fails, control transfers to the exception-handling part of the sub-block, and after the exception handler runs, the loop repeats.

**Example 12-26 Retrying Transaction After Handling Exception**

```sql
DROP TABLE results;
CREATE TABLE results (
    res_name   VARCHAR(20),
    res_answer VARCHAR2(3)
);

CREATE UNIQUE INDEX res_name_ix ON results (res_name);
```
INSERT INTO results (res_name, res_answer) VALUES ('SMYTHE', 'YES');
INSERT INTO results (res_name, res_answer) VALUES ('JONES', 'NO');

DECLARE
  name VARCHAR2(20) := 'SMYTHE';
  answer VARCHAR2(3) := 'NO';
  suffix NUMBER := 1;
BEGIN
  FOR i IN 1..5 LOOP  -- Try transaction at most 5 times.
    DBMS_OUTPUT.PUT('Try #' || i);

    BEGIN -- sub-block begins
      SAVEPOINT start_transaction;

      -- transaction begins
      DELETE FROM results WHERE res_answer = 'NO';
      INSERT INTO results (res_name, res_answer) VALUES (name, answer);

      -- Nonunique name raises DUP_VAL_ON_INDEX.
      -- If transaction succeeded:
      COMMIT;
      DBMS_OUTPUT.PUT_LINE(' succeeded.);
      EXIT;
      
      EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE(' failed; trying again.');
        ROLLBACK TO start_transaction;  -- Undo changes.
        suffix := suffix + 1;          -- Try to fix problem.
        name := name || TO_CHAR(suffix);
    END; -- sub-block ends
  END LOOP;
END;
/

Result:
Try #1 failed; trying again.
Try #2 succeeded.

Example 12-26 uses the preceding technique to retry a transaction whose INSERT statement raises the predefined exception DUP_VAL_ON_INDEX if the value of res_name is not unique.

12.13 Handling Errors in Distributed Queries

You can use a trigger or a stored subprogram to create a distributed query. This distributed query is decomposed by the local Oracle Database instance into a corresponding number of remote queries, which are sent to the remote nodes for execution. The remote nodes run the queries and send the results back to the local node. The local node then performs any necessary post-processing and returns the results to the user or application.
If a portion of a distributed statement fails, possibly from a constraint violation, then Oracle Database returns ORA-02055. Subsequent statements, or subprogram invocations, return ORA-02067 until a rollback or a rollback to savepoint is entered.

Design your application to check for any returned error messages that indicates that a portion of the distributed update has failed. If you detect a failure, rollback the entire transaction (or rollback to a savepoint) before allowing the application to proceed.
This chapter explains how the PL/SQL compiler optimizes your code and how to write efficient PL/SQL code and improve existing PL/SQL code.

Topics

• PL/SQL Optimizer
• Candidates for Tuning
• Minimizing CPU Overhead
• Bulk SQL and Bulk Binding
• Chaining Pipelined Table Functions for Multiple Transformations
• Overview of Polymorphic Table Functions
• Updating Large Tables in Parallel
• Collecting Data About User-Defined Identifiers
• Profiling and Tracing PL/SQL Programs
• Compiling PL/SQL Units for Native Execution

See Also:

Oracle Database Development Guide for disadvantages of cursor variables

13.1 PL/SQL Optimizer

Prior to Oracle Database 10g release 1, the PL/SQL compiler translated your source text to system code without applying many changes to improve performance. Now, PL/SQL uses an optimizer that can rearrange code for better performance.

The optimizer is enabled by default. In rare cases, if the overhead of the optimizer makes compilation of very large applications too slow, you can lower the optimization by setting the compilation parameter PLSQL_OPTIMIZE_LEVEL=1 instead of its default value 2. In even rarer cases, PL/SQL might raise an exception earlier than expected or not at all. Setting PLSQL_OPTIMIZE_LEVEL=1 prevents the code from being rearranged.
13.1.1 Subprogram Inlining

One optimization that the compiler can perform is **subprogram inlining**.

Subprogram inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit). To allow subprogram inlining, either accept the default value of the **PLSQL_OPTIMIZE_LEVEL** compilation parameter (which is 2) or set it to 3.

With `PLSQL_OPTIMIZE_LEVEL=2`, you must specify each subprogram to be inlined with the **INLINE** pragma:

```
PRAGMA INLINE (subprogram, 'YES')
```

If `subprogram` is overloaded, then the preceding pragma applies to every subprogram with that name.

With `PLSQL_OPTIMIZE_LEVEL=3`, the PL/SQL compiler seeks opportunities to inline subprograms. You need not specify subprograms to be inlined. However, you can use the **INLINE** pragma (with the preceding syntax) to give a subprogram a high priority for inlining, and then the compiler inlines it unless other considerations or limits make the inlining undesirable.

If a particular subprogram is inlined, performance almost always improves. However, because the compiler inlines subprograms early in the optimization process, it is possible for subprogram inlining to preclude later, more powerful optimizations.

If subprogram inlining slows the performance of a particular PL/SQL program, then use the PL/SQL hierarchical profiler to identify subprograms for which you want to turn off inlining. To turn off inlining for a subprogram, use the **INLINE** pragma:

```
PRAGMA INLINE (subprogram, 'NO')
```

The **INLINE** pragma affects only the immediately following declaration or statement, and only some kinds of statements.

When the **INLINE** pragma immediately precedes a declaration, it affects:

- Every invocation of the specified subprogram in that declaration
- Every initialization value in that declaration except the default initialization values of records

When the **INLINE** pragma immediately precedes one of these statements, the pragma affects every invocation of the specified subprogram in that statement:
• Assignment
• CALL
• Conditional
• CASE
• CONTINUE WHEN
• EXECUTE IMMEDIATE
• EXIT WHEN
• LOOP
• RETURN

The `INLINE` pragma does not affect statements that are not in the preceding list.

Multiple pragmas can affect the same declaration or statement. Each pragma applies its own effect to the statement. If `PRAGMA INLINE(subprogram,'YES')` and `PRAGMA INLINE(identifier,'NO')` have the same subprogram, then 'NO' overrides 'YES'. One `PRAGMA INLINE(subprogram,'NO')` overrides any number of occurrences of `PRAGMA INLINE(subprogram,'YES')`, and the order of these pragmas is not important.

**See Also:**
- Oracle Database Development Guide for more information about PL/SQL hierarchical profiler
- Oracle Database Reference for information about the `PLSQL_OPTIMIZE_LEVEL` compilation parameter
- Oracle Database Reference for information about the static dictionary view `ALL_PLSQL_OBJECT_SETTINGS`

**Example 13-1  Specifying that Subprogram Is To Be Inlined**

In this example, if `PLSQL_OPTIMIZE_LEVEL=2`, the `INLINE` pragma affects the procedure invocations `p1(1)` and `p1(2)`, but not the procedure invocations `p1(3)` and `p1(4)`.

```plaintext
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
x := p1(1) + p1(2) + 17;  -- These 2 invocations to p1 are inlined
...
x := p1(3) + p1(4) + 17;  -- These 2 invocations to p1 are not inlined
...
```

**Example 13-2  Specifying that Overloaded Subprogram Is To Be Inlined**

In this example, if `PLSQL_OPTIMIZE_LEVEL=2`, the `INLINE` pragma affects both functions named `p2`.

```plaintext
FUNCTION p2 (p boolean) return PLS_INTEGER IS ...
FUNCTION p2 (x PLS_INTEGER) return PLS_INTEGER IS ...
...
PRAGMA INLINE (p2, 'YES');
x := p2(true) + p2(3); 
```
Example 13-3  Specifying that Subprogram Is Not To Be Inlined

In this example, the \texttt{INLINE} pragma affects the procedure invocations \texttt{p1(1)} and \texttt{p1(2)}, but not the procedure invocations \texttt{p1(3)} and \texttt{p1(4)}.

\begin{verbatim}
PROCEDURE p1 (x PLS_INTEGER) IS ... 
... 
PRAGMA INLINE (p1, 'NO'); 
x:= p1(1) + p1(2) + 17;  -- These 2 invocations to p1 are not inlined 
... 
x:= p1(3) + p1(4) + 17;  -- These 2 invocations to p1 might be inlined 
...
\end{verbatim}

Example 13-4  \texttt{PRAGMA INLINE} ... 'NO' Overrides \texttt{PRAGMA INLINE} ... 'YES'

In this example, the second \texttt{INLINE} pragma overrides both the first and third \texttt{INLINE} pragmas.

\begin{verbatim}
PROCEDURE p1 (x PLS_INTEGER) IS ... 
... 
PRAGMA INLINE (p1, 'YES'); 
PRAGMA INLINE (p1, 'NO'); 
PRAGMA INLINE (p1, 'YES'); 
x:= p1(1) + p1(2) + 17;  -- These 2 invocations to p1 are not inlined 
...
\end{verbatim}

13.2 Candidates for Tuning

The following kinds of PL/SQL code are very likely to benefit from tuning:

- Older code that does not take advantage of new PL/SQL language features.

\begin{quote}
\textbf{Tip:}

Before tuning older code, benchmark the current system and profile the older subprograms that your program invokes (see “Profiling and Tracing PL/SQL Programs”). With the many automatic optimizations of the PL/SQL optimizer (described in “PL/SQL Optimizer”), you might see performance improvements before doing any tuning.
\end{quote}

- Older dynamic SQL statements written with the \texttt{DBMS_SQL} package.

If you know at compile time the number and data types of the input and output variables of a dynamic SQL statement, then you can rewrite the statement in native dynamic SQL, which runs noticeably faster than equivalent code that uses the \texttt{DBMS_SQL} package (especially when it can be optimized by the compiler). For more information, see PL/SQL Dynamic SQL.

- Code that spends much time processing SQL statements.

See "Tune SQL Statements".

- Functions invoked in queries, which might run millions of times.

See "Tune Function Invocations in Queries".

- Code that spends much time looping through query results.
See “Tune Loops”.

• Code that does many numeric computations.
See “Tune Computation-Intensive PL/SQL Code”.

• Code that spends much time processing PL/SQL statements (as opposed to issuing database definition language (DDL) statements that PL/SQL passes directly to SQL).
See “Compiling PL/SQL Units for Native Execution”.

13.3 Minimizing CPU Overhead

Topics

• Tune SQL Statements
• Tune Function Invocations in Queries
• Tune Subprogram Invocations
• Tune Loops
• Tune Computation-Intensive PL/SQL Code
• Use SQL Character Functions
• Put Least Expensive Conditional Tests First

13.3.1 Tune SQL Statements

The most common cause of slowness in PL/SQL programs is slow SQL statements. To make SQL statements in a PL/SQL program as efficient as possible:

• Use appropriate indexes.
  For details, see Oracle Database Performance Tuning Guide.

• Use query hints to avoid unnecessary full-table scans.
  For details, see Oracle Database SQL Language Reference.

• Collect current statistics on all tables, using the subprograms in the DBMS_STATS package.
  For details, see Oracle Database Performance Tuning Guide.

• Analyze the execution plans and performance of the SQL statements, using:
  – EXPLAIN PLAN statement
    For details, see Oracle Database Performance Tuning Guide.
  – SQL Trace facility with TKPROF utility
    For details, see Oracle Database Performance Tuning Guide.

• Use bulk SQL, a set of PL/SQL features that minimizes the performance overhead of the communication between PL/SQL and SQL.
  For details, see "Bulk SQL and Bulk Binding".
13.3.2 Tune Function Invocations in Queries

Functions invoked in queries might run millions of times. Do not invoke a function in a query unnecessarily, and make the invocation as efficient as possible.

Create a function-based index on the table in the query. The `CREATE INDEX` statement might take a while, but the query can run much faster because the function value for each row is cached.

If the query passes a column to a function, then the query cannot use user-created indexes on that column, so the query might invoke the function for every row of the table (which might be very large). To minimize the number of function invocations, use a nested query. Have the inner query filter the result set to a small number of rows, and have the outer query invoke the function for only those rows.

See Also:

- *Oracle Database SQL Language Reference* for more information about `CREATE INDEX` statement syntax
- "PL/SQL Function Result Cache" for information about caching the results of PL/SQL functions

Example 13-5 Nested Query Improves Performance

In this example, the two queries produce the same result set, but the second query is more efficient than the first. (In the example, the times and time difference are very small, because the EMPLOYEES table is very small. For a very large table, they would be significant.)

```sql
DECLARE
    starting_time  TIMESTAMP WITH TIME ZONE;
    ending_time    TIMESTAMP WITH TIME ZONE;
BEGIN
    -- Invokes SQRT for every row of employees table:
    SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;
    FOR item IN (SELECT DISTINCT(SQRT(department_id)) col_alias
    FROM employees
    ORDER BY col_alias)
    LOOP
        DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
    END LOOP;
    SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;
    DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));

    -- Invokes SQRT for every distinct department_id of employees table:
    SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;
    SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;
    DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));
END;```

```
FOR item IN (  
    SELECT SQRT(department_id) col_alias  
    FROM (SELECT DISTINCT department_id FROM employees)  
    ORDER BY col_alias  
)  
LOOP  
    IF item.col_alias IS NOT NULL THEN  
        DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);  
    END IF;  
    END LOOP;  
SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;  
DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));  
END;  
/

Result is similar to:

Square root of dept. ID = 3.16227766016837933199889354443271853372
Square root of dept. ID = 4.4721359499957993928134733746255247088
Square root of dept. ID = 5.477225575051661134566997829800082133953
Square root of dept. ID = 6.32455320336758663997778708886543706744
Square root of dept. ID = 7.07106781186547524400844362104849039285
Square root of dept. ID = 7.745966696924183377035853079956479922167
Square root of dept. ID = 8.366600265340755479781720257851874899393
Square root of dept. ID = 8.9442719099915878563669467492510494176
Square root of dept. ID = 9.4868329805051379959968063329815560116
Square root of dept. ID = 10
Square root of dept. ID = 10.48808848170151546991453513679937598475
Time = +000000000 00:00:00.046000000
Square root of dept. ID = 3.16227766016837933199889354443271853372
Square root of dept. ID = 4.4721359499957993928134733746255247088
Square root of dept. ID = 5.477225575051661134566997829800082133953
Square root of dept. ID = 6.32455320336758663997778708886543706744
Square root of dept. ID = 7.07106781186547524400844362104849039285
Square root of dept. ID = 7.745966696924183377035853079956479922167
Square root of dept. ID = 8.366600265340755479781720257851874899393
Square root of dept. ID = 8.9442719099915878563669467492510494176
Square root of dept. ID = 9.4868329805051379959968063329815560116
Square root of dept. ID = 10
Square root of dept. ID = 10.48808848170151546991453513679937598475
Time = +000000000 00:00:00.000000000

13.3.3 Tune Subprogram Invocations

If a subprogram has OUT or IN OUT parameters, you can sometimes decrease its
invocation overhead by declaring those parameters with the nocopy hint.

When OUT or IN OUT parameters represent large data structures such as collections,
records, and instances of ADTs, copying them slows execution and increases memory
use—especially for an instance of an ADT.

For each invocation of an ADT method, PL/SQL copies every attribute of the ADT. If
the method is exited normally, then PL/SQL applies any changes that the method
made to the attributes. If the method is exited with an unhandled exception, then
PL/SQL does not change the attributes.

If your program does not require that an OUT or IN OUT parameter retain its pre-
invocation value if the subprogram ends with an unhandled exception, then include the
NOCOPY hint in the parameter declaration. The NOCOPY hint requests (but does not ensure) that the compiler pass the corresponding actual parameter by reference instead of value.

**Caution:**

Do not rely on NOCOPY (which the compiler might or might not obey for a particular invocation) to ensure that an actual parameter or ADT attribute retains its pre-invocation value if the subprogram is exited with an unhandled exception. Instead, ensure that the subprogram handle all exceptions.

**See Also:**

- "NOCOPY" for more information about NOCOPY hint
- Oracle Database Object-Relational Developer's Guide for information about using NOCOPY with member methods of ADTs

**Example 13-6  NOCOPY Subprogram Parameters**

In this example, if the compiler obeys the NOCOPY hint for the invocation of do_nothing2, then the invocation of do_nothing2 is faster than the invocation of do_nothing1.

```sql
DECLARE
    TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE;
    emp_tab EmpTabTyp := EmpTabTyp(NULL);  -- initialize
    t1 NUMBER;
    t2 NUMBER;
    t3 NUMBER;

    PROCEDURE get_time (t OUT NUMBER) IS
    BEGIN
        t := DBMS_UTILITY.get_time;
    END;

    PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
    BEGIN
        NULL;
    END;

    PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
    BEGIN
        NULL;
    END;

    BEGIN
        SELECT * INTO emp_tab(1)
        FROM employees
        WHERE employee_id = 100;

        emp_tab.EXTEND(49999, 1);  -- Copy element 1 into 2..50000
        get_time(t1);
```
13.3.4 Tune Loops

Because PL/SQL applications are often built around loops, it is important to optimize both the loops themselves and the code inside them.

If you must loop through a result set more than once, or issue other queries as you loop through a result set, you might be able to change the original query to give you exactly the results you want. Explore the SQL set operators that let you combine multiple queries, described in Oracle Database SQL Language Reference.

You can also use subqueries to do the filtering and sorting in multiple stages—see "Processing Query Result Sets with Subqueries".

See Also:
"Bulk SQL and Bulk Binding"

13.3.5 Tune Computation-Intensive PL/SQL Code

These recommendations apply especially (but not only) to computation-intensive PL/SQL code.

Topics

- Use Data Types that Use Hardware Arithmetic
- Avoid Constrained Subtypes in Performance-Critical Code
- Minimize Implicit Data Type Conversion

13.3.5.1 Use Data Types that Use Hardware Arithmetic

Avoid using data types in the NUMBER data type family (described in "NUMBER Data Type Family"). These data types are represented internally in a format designed for portability and arbitrary scale and precision, not for performance. Operations on data of these types use library arithmetic, while operations on data of the types PLS_INTEGER, BINARY_FLOAT and BINARY_DOUBLE use hardware arithmetic.

For local integer variables, use PLS_INTEGER, described in "PLS_INTEGER and BINARY_INTEGER Data Types". For variables used in performance-critical code, that can never have the value NULL, and do not need overflow checking, use SIMPLE_INTEGER, described in "SIMPLE_INTEGER Subtype of PLS_INTEGER".
For floating-point variables, use `BINARY_FLOAT` or `BINARY_DOUBLE`, described in Oracle Database SQL Language Reference. For variables used in performance-critical code, that can never have the value `NULL`, and that do not need overflow checking, use `SIMPLE_FLOAT` or `SIMPLE_DOUBLE`, explained in "Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE".

**Note:**

`BINARY_FLOAT` and `BINARY_DOUBLE` and their subtypes are less suitable for financial code where accuracy is critical, because they do not always represent fractional values precisely, and handle rounding differently than the `NUMBER` types.

Many SQL numeric functions (described in Oracle Database SQL Language Reference) are overloaded with versions that accept `BINARY_FLOAT` and `BINARY_DOUBLE` parameters. You can speed up computation-intensive code by passing variables of these data types to such functions, and by invoking the conversion functions `TO_BINARY_FLOAT` (described in Oracle Database SQL Language Reference) and `TO_BINARY_DOUBLE` (described in Oracle Database SQL Language Reference) when passing expressions to such functions.

13.3.5.2 Avoid Constrained Subtypes in Performance-Critical Code

In performance-critical code, avoid constrained subtypes (described in "Constrained Subtypes"). Each assignment to a variable or parameter of a constrained subtype requires extra checking at run time to ensure that the value to be assigned does not violate the constraint.

**See Also:**

PL/SQL Predefined Data Types includes predefined constrained subtypes

13.3.5.3 Minimize Implicit Data Type Conversion

At run time, PL/SQL converts between different data types implicitly (automatically) if necessary. For example, if you assign a `PLS_INTEGER` variable to a `NUMBER` variable, then PL/SQL converts the `PLS_INTEGER` value to a `NUMBER` value (because the internal representations of the values differ).

Whenever possible, minimize implicit conversions. For example:

- If a variable is to be either inserted into a table column or assigned a value from a table column, then give the variable the same data type as the table column.

**Tip:**

Declare the variable with the `%TYPE` attribute, described in "%TYPE Attribute".
• Make each literal the same data type as the variable to which it is assigned or the expression in which it appears.

• Convert values from SQL data types to PL/SQL data types and then use the converted values in expressions.

For example, convert NUMBER values to PLS_INTEGER values and then use the PLS_INTEGER values in expressions. PLS_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic. For more information about the PLS_INTEGER data type, see "PLS_INTEGER and BINARY_INTEGER Data Types".

• Before assigning a value of one SQL data type to a variable of another SQL data type, explicitly convert the source value to the target data type, using a SQL conversion function (for information about SQL conversion functions, see Oracle Database SQL Language Reference).

• Overload your subprograms with versions that accept parameters of different data types and optimize each version for its parameter types. For information about overloaded subprograms, see "Overloaded Subprograms".

See Also:

- Oracle Database SQL Language Reference for information about implicit conversion of SQL data types (which are also PL/SQL data types)
- "Subtypes with Base Types in Same Data Type Family"

13.3.6 Use SQL Character Functions

SQL has many highly optimized character functions, which use low-level code that is more efficient than PL/SQL code. Use these functions instead of writing PL/SQL code to do the same things.

See:

- Oracle Database SQL Language Reference for information about SQL character functions that return character values
- Oracle Database SQL Language Reference for information about SQL character functions that return NLS character values
- Oracle Database SQL Language Reference for information about SQL character functions that return number values
- Example 7-6 for an example of PL/SQL code that uses SQL character function REGEXP_LIKE

13.3.7 Put Least Expensive Conditional Tests First

PL/SQL stops evaluating a logical expression as soon as it can determine the result. Take advantage of this short-circuit evaluation by putting the conditions that are least
expensive to evaluate first in logical expressions whenever possible. For example, test
the values of PL/SQL variables before testing function return values, so that if the
variable tests fail, PL/SQL need not invoke the functions:

\[
\text{IF boolean\_variable OR } (\text{number} > 10) \text{ OR boolean\_function}\(\text{parameter}\) \text{ THEN} \ldots
\]

See Also:
"Short-Circuit Evaluation"

13.4 Bulk SQL and Bulk Binding

**Bulk SQL** minimizes the performance overhead of the communication between
PL/SQL and SQL. The PL/SQL features that comprise bulk SQL are the **FORALL**
statement and the **BULK** **COLLECT** clause. Assigning values to PL/SQL variables that
appear in SQL statements is called **binding**.

PL/SQL and SQL communicate as follows: To run a **SELECT** **INTO** or DML statement,
the PL/SQL engine sends the query or DML statement to the SQL engine. The SQL
engine runs the query or DML statement and returns the result to the PL/SQL engine.

The **FORALL** statement sends DML statements from PL/SQL to SQL in batches rather
than one at a time. The **BULK** **COLLECT** clause returns results from SQL to PL/SQL in
batches rather than one at a time. If a query or DML statement affects four or more
database rows, then bulk SQL can significantly improve performance.

**Note:**
You cannot perform bulk SQL on remote tables.

PL/SQL binding operations fall into these categories:

<table>
<thead>
<tr>
<th>Binding Category</th>
<th>When This Binding Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-bind</td>
<td>When an <strong>INSERT</strong>, <strong>UPDATE</strong>, or <strong>MERGE</strong> statement stores a PL/SQL or host variable in the database</td>
</tr>
<tr>
<td>Out-bind</td>
<td>When the <strong>RETURNING INTO</strong> clause of an <strong>INSERT</strong>, <strong>UPDATE</strong>, or <strong>DELETE</strong> statement assigns a database value to a PL/SQL or host variable</td>
</tr>
<tr>
<td>DEFINE</td>
<td>When a <strong>SELECT</strong> or <strong>FETCH</strong> statement assigns a database value to a PL/SQL or host variable</td>
</tr>
</tbody>
</table>

For in-binds and out-binds, bulk SQL uses **bulk binding**; that is, it binds an entire
collection of values at once. For a collection of \( n \) elements, bulk SQL uses a single
operation to perform the equivalent of \( n \) **SELECT** **INTO** or DML statements. A query that
uses bulk SQL can return any number of rows, without using a **FETCH** statement for
each one.
The **FORALL** statement, a feature of bulk SQL, sends DML statements from PL/SQL to SQL in batches rather than one at a time.

To understand the **FORALL** statement, first consider the **FOR LOOP** statement in Example 13-7. It sends these DML statements from PL/SQL to SQL one at a time:

```
DELETE FROM employees_temp WHERE department_id = depts(10);
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

Now consider the **FORALL** statement in Example 13-8. It sends the same three DML statements from PL/SQL to SQL as a batch.

A **FORALL** statement is usually much faster than an equivalent **FOR LOOP** statement. However, a **FOR LOOP** statement can contain multiple DML statements, while a **FORALL** statement can contain only one. The batch of DML statements that a **FORALL** statement sends to SQL differ only in their **VALUES** and **WHERE** clauses. The values in those clauses must come from existing, populated collections.

---

### Note:

The DML statement in a **FORALL** statement can reference multiple collections, but performance benefits apply only to collection references that use the **FORALL** index variable as an index.

---

**Example 13-9** inserts the same collection elements into two database tables, using a **FOR LOOP** statement for the first table and a **FORALL** statement for the second table and showing how long each statement takes. (Times vary from run to run.)

In **Example 13-10**, the **FORALL** statement applies to a subset of a collection.

---

Topics

- **FORALL Statement**
- **BULK COLLECT Clause**
- Using **FORALL** Statement and **BULK COLLECT** Clause Together
- Client Bulk-Binding of Host Arrays

---

---
• Handling FORALL Exceptions After FORALL Statement Completes
• Getting Number of Rows Affected by FORALL Statement

See Also:
• "FORALL Statement" for its complete syntax and semantics, including restrictions
• "Implicit Cursors" for information about implicit cursor attributes in general and other implicit cursor attributes that you can use with the FORALL statement

Example 13-7  DELETE Statement in FOR LOOP Statement

DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FOR i IN depts.FIRST..depts.LAST LOOP
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
  END LOOP;
END;
/

Example 13-8  DELETE Statement in FORALL Statement

DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FORALL i IN depts.FIRST..depts.LAST
  DELETE FROM employees_temp
  WHERE department_id = depts(i);
END;
/

Example 13-9  Time Difference for INSERT Statement in FOR LOOP and FORALL Statements

DROP TABLE parts1;
CREATE TABLE parts1 {
  pnum INTEGER,
  pname VARCHAR2(15)
};

DROP TABLE parts2;
CREATE TABLE parts2 {
  pnum INTEGER,
  pname VARCHAR2(15)
};
DECLARE
  TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
  pnums   NumTab;
  pnames  NameTab;
  iterations CONSTANT PLS_INTEGER := 50000;
  t1  INTEGER;
  t2  INTEGER;
  t3  INTEGER;
BEGIN
  FOR j IN 1..iterations LOOP  -- populate collections
    pnums(j) := j;
    pnames(j) := 'Part No. ' || TO_CHAR(j);
  END LOOP;
  t1 := DBMS_UTILITY.get_time;

  FOR i IN 1..iterations LOOP
    INSERT INTO parts1 (pnum, pname) VALUES (pnums(i), pnames(i));
  END LOOP;
  t2 := DBMS_UTILITY.get_time;

  FORALL i IN 1..iterations
    INSERT INTO parts2 (pnum, pname) VALUES (pnums(i), pnames(i));
  t3 := DBMS_UTILITY.get_time;
  DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
  DBMS_OUTPUT.PUT_LINE('---------------------');
  DBMS_OUTPUT.PUT_LINE('FOR LOOP: ' || TO_CHAR((t2 - t1)/100));
  DBMS_OUTPUT.PUT_LINE('FORALL:   ' || TO_CHAR((t3 - t2)/100));
  COMMIT;
END;
/

Result is similar to:

Execution Time (secs)
---------------------
FOR LOOP: 5.97
FORALL:   .07

PL/SQL procedure successfully completed.

Example 13-10    FORALL Statement for Subset of Collection

DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(10) OF NUMBER;
  depts NumList := NumList(5,10,20,30,50,55,57,60,70,75);
BEGIN
  FORALL j IN 4..7
    DELETE FROM employees_temp WHERE department_id = depts(j);
END;
/
13.4.1.1 Using FORALL Statements for Sparse Collections

If the FORALL statement bounds clause references a sparse collection, then specify only existing index values, using either the INDICES OF or VALUES OF clause.

You can use INDICES OF for any collection except an associative array indexed by string. You can use VALUES OF only for a collection of PLS_INTEGER elements indexed by PLS_INTEGER.

A collection of PLS_INTEGER elements indexed by PLS_INTEGER can be an index collection; that is, a collection of pointers to elements of another collection (the indexed collection).

Index collections are useful for processing different subsets of the same collection with different FORALL statements. Instead of copying elements of the original collection into new collections that represent the subsets (which can use significant time and memory), represent each subset with an index collection and then use each index collection in the VALUES OF clause of a different FORALL statement.

See Also:
"Sparse Collections and SQL%BULK_EXCEPTIONS"

Example 13-11  FORALL Statements for Sparse Collection and Its Subsets

This example uses a FORALL statement with the INDICES OF clause to populate a table with the elements of a sparse collection. Then it uses two FORALL statements with VALUES OF clauses to populate two tables with subsets of a collection.

```sql
DROP TABLE valid_orders;
CREATE TABLE valid_orders (  
cust_name VARCHAR2(32),
amount NUMBER(10,2)
);

DROP TABLE big_orders;
CREATE TABLE big_orders AS  
SELECT * FROM valid_orders  
WHERE 1 = 0;

DROP TABLE rejected_orders;
CREATE TABLE rejected_orders AS  
SELECT * FROM valid_orders  
WHERE 1 = 0;

DECLARE  
SUBTYPE cust_name IS valid_orders.cust_name%TYPE;  
TYPE cust_typ IS TABLE OF cust_name;  
cust_tab cust_typ;  -- Collection of customer names

SUBTYPE order_amount IS valid_orders.amount%TYPE;  
TYPE amount_typ IS TABLE OF NUMBER;  
amount_tab amount_typ;  -- Collection of order amounts

TYPE index_pointer_t IS TABLE OF PLS_INTEGER;
```
/* Collections for pointers to elements of cust_tab collection  
   (to represent two subsets of cust_tab): */

big_order_tab       index_pointer_t := index_pointer_t();
rejected_order_tab  index_pointer_t := index_pointer_t();

PROCEDURE populate_data_collections IS
BEGIN
    cust_tab := cust_typ(
                  'Company1','Company2','Company3','Company4','Company5'
                );

    amount_tab := amount_typ(5000.01, 0, 150.25, 4000.00, NULL);
END;
BEGIN
    populate_data_collections;
    DBMS_OUTPUT.PUT_LINE ('--- Original order data ---');
    FOR i IN 1..cust_tab.LAST LOOP
        DBMS_OUTPUT.PUT_LINE (  
            'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)  
        );
    END LOOP;
    -- Delete invalid orders:
    FOR i IN 1..cust_tab.LAST LOOP
        IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
            cust_tab.delete(i);
            amount_tab.delete(i);
        END IF;
    END LOOP;
    -- cust_tab is now a sparse collection.
    DBMS_OUTPUT.PUT_LINE ('--- Order data with invalid orders deleted ---');
    FOR i IN 1..cust_tab.LAST LOOP
        IF cust_tab.EXISTS(i) THEN
            DBMS_OUTPUT.PUT_LINE (  
                'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)  
            );
        END IF;
    END LOOP;
    -- Using sparse collection, populate valid_orders table:
    FORALL i IN INDICES OF cust_tab
    INSERT INTO valid_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));
    populate_data_collections;  -- Restore original order data
    -- cust_tab is a dense collection again.

/* Populate collections of pointers to elements of cust_tab collection  
   (which represent two subsets of cust_tab): */
FOR i IN cust_tab.FIRST .. cust_tab.LAST LOOP
  IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
    rejected_order_tab.EXTEND;
    rejected_order_tab(rejected_order_tab.LAST) := i;
  END IF;

  IF amount_tab(i) > 2000 THEN
    big_order_tab.EXTEND;
    big_order_tab(big_order_tab.LAST) := i;
  END IF;
END LOOP;

/* Using each subset in a different FORALL statement, populate rejected_orders and big_orders tables: */

FORALL i IN VALUES OF rejected_order_tab
  INSERT INTO rejected_orders (cust_name, amount)
  VALUES (cust_tab(i), amount_tab(i));

FORALL i IN VALUES OF big_order_tab
  INSERT INTO big_orders (cust_name, amount)
  VALUES (cust_tab(i), amount_tab(i));
END;
/

Result:

--- Original order data ---
Customer #1, Company1: $5000.01
Customer #2, Company2: $0
Customer #3, Company3: $150.25
Customer #4, Company4: $4000
Customer #5, Company5: $

--- Data with invalid orders deleted ---
Customer #1, Company1: $5000.01
Customer #3, Company3: $150.25
Customer #4, Company4: $4000

Verify that correct order details were stored:

SELECT cust_name "Customer", amount "Valid order amount"
FROM valid_orders
ORDER BY cust_name;

Result:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Valid order amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company1</td>
<td>5000.01</td>
</tr>
<tr>
<td>Company3</td>
<td>150.25</td>
</tr>
<tr>
<td>Company4</td>
<td>4000</td>
</tr>
</tbody>
</table>

3 rows selected.

Query:

SELECT cust_name "Customer", amount "Big order amount"
FROM big_orders
ORDER BY cust_name;

Result:
<table>
<thead>
<tr>
<th>Customer</th>
<th>Big order amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company1</td>
<td>5000.01</td>
</tr>
<tr>
<td>Company4</td>
<td>4000</td>
</tr>
</tbody>
</table>

2 rows selected.

Query:

```sql
SELECT cust_name "Customer", amount "Rejected order amount"
FROM rejected_orders
ORDER BY cust_name;
```

Result:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Rejected order amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company2</td>
<td></td>
</tr>
<tr>
<td>Company5</td>
<td></td>
</tr>
</tbody>
</table>

2 rows selected.

### 13.4.1.2 Unhandled Exceptions in FORALL Statements

In a **FORALL** statement without the **SAVE EXCEPTIONS** clause, if one DML statement raises an unhandled exception, then PL/SQL stops the **FORALL** statement and rolls back all changes made by previous DML statements.

For example, the **FORALL** statement in Example 13-8 executes these DML statements in this order, unless one of them raises an unhandled exception:

```sql
DELETE FROM employees_temp WHERE department_id = depts(10);
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

If the third statement raises an unhandled exception, then PL/SQL rolls back the changes that the first and second statements made. If the second statement raises an unhandled exception, then PL/SQL rolls back the changes that the first statement made and never runs the third statement.

You can handle exceptions raised in a **FORALL** statement in either of these ways:

- As each exception is raised (see "Handling **FORALL** Exceptions Immediately")
- After the **FORALL** statement completes execution, by including the **SAVE EXCEPTIONS** clause (see "Handling **FORALL** Exceptions After **FORALL** Statement Completes")

### 13.4.1.3 Handling **FORALL** Exceptions Immediately

To handle exceptions raised in a **FORALL** statement immediately, omit the **SAVE EXCEPTIONS** clause and write the appropriate exception handlers.

If one DML statement raises a handled exception, then PL/SQL rolls back the changes made by that statement, but does not roll back changes made by previous DML statements.

In Example 13-12, the **FORALL** statement is designed to run three **UPDATE** statements. However, the second one raises an exception. An exception handler handles the exception, displaying the error message and committing the change made by the first **UPDATE** statement. The third **UPDATE** statement never runs.
For information about exception handlers, see PL/SQL Error Handling.

Example 13-12 Handling FORALL Exceptions Immediately

DROP TABLE emp_temp;
CREATE TABLE emp_temp (  
deptno NUMBER(2),  
job VARCHAR2(18)  
);

CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  TYPE NumList IS TABLE OF NUMBER;
  depts          NumList := NumList(10, 20, 30);
  error_message  VARCHAR2(100);
BEGIN
  -- Populate table:
  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:
  FORALL j IN depts.FIRST..depts.LAST
    UPDATE emp_temp SET job = job || ' (Senior)'  
      WHERE deptno = depts(j);
EXCEPTION
  WHEN OTHERS THEN
    error_message := SQLERRM;
    DBMS_OUTPUT.PUT_LINE (error_message);
  COMMIT;  -- Commit results of successful updates
  RAISE;
END;
/

Result:
Procedure created.

Invoke procedure:
BEGIN
  p;
END;
/

Result:
ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19, maximum: 18)
BEGIN
  *
  ERROR at line 1:
  ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19, maximum: 18)
  ORA-06512: at "HR.P", line 27
  ORA-06512: at line 2
**Query:**

```
SELECT * FROM emp_temp;
```

**Result:**

```
DEPTNO JOB
---------- ------------------
10 Clerk (Senior)
20 Bookkeeper
30 Analyst
```

3 rows selected.

### 13.4.1.4 Handling FORALL Exceptions After FORALL Statement Completes

To allow a FORALL statement to continue even if some of its DML statements fail, include the `SAVE EXCEPTIONS` clause. When a DML statement fails, PL/SQL does not raise an exception; instead, it saves information about the failure. After the FORALL statement completes, PL/SQL raises a single exception for the FORALL statement (ORA-24381).

In the exception handler for ORA-24381, you can get information about each individual DML statement failure from the implicit cursor attribute `SQL%BULK_EXCEPTIONS`.

`SQL%BULK_EXCEPTIONS` is like an associative array of information about the DML statements that failed during the most recently run FORALL statement.

`SQL%BULK_EXCEPTIONS.COUNT` is the number of DML statements that failed. If `SQL%BULK_EXCEPTIONS.COUNT` is not zero, then for each index value `i` from 1 through `SQL%BULK_EXCEPTIONS.COUNT`:

- `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX` is the number of the DML statement that failed.
- `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` is the Oracle Database error code for the failure.

For example, if a FORALL SAVE EXCEPTIONS statement runs 100 DML statements, and the tenth and sixty-fourth ones fail with error codes ORA-12899 and ORA-19278, respectively, then:

- `SQL%BULK_EXCEPTIONS.COUNT = 2`
- `SQL%BULK_EXCEPTIONS(1).ERROR_INDEX = 10`
- `SQL%BULK_EXCEPTIONS(1).ERROR_CODE = 12899`
- `SQL%BULK_EXCEPTIONS(2).ERROR_INDEX = 64`
- `SQL%BULK_EXCEPTIONS(2).ERROR_CODE = 19278`

**Note:**

After a FORALL statement without the SAVE EXCEPTIONS clause raises an exception, `SQL%BULK_EXCEPTIONS.COUNT = 1.`
With the error code, you can get the associated error message with the SQLERRM function (described in "SQLERRM Function"):

\[
\text{SQLERRM}(-\text{SQL\%BULK_EXCEPTIONS}(i).\text{ERROR_CODE})
\]

However, the error message that SQLERRM returns excludes any substitution arguments (compare the error messages in Example 13-12 and Example 13-13).

**Example 13-13** is like **Example 13-12** except:

- The **FORALL** statement includes the **SAVE EXCEPTIONS** clause.
- The exception-handling part has an exception handler for ORA-24381, the internally defined exception that PL/SQL raises implicitly when a bulk operation raises and saves exceptions. The example gives ORA-24381 the user-defined name **dml_errors**.
- The exception handler for **dml_errors** uses **SQL\%BULK_EXCEPTIONS** and **SQLERRM** (and some local variables) to show the error message and which statement, collection item, and string caused the error.

**Example 13-13 Handling FORALL Exceptions After FORALL Statement Completes**

```sql
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  TYPE NumList IS TABLE OF NUMBER;
decs        NumList := NumList(10, 20, 30);

  error_message  VARCHAR2(100);
bad_stmt_no    PLS_INTEGER;
bad_deptno     emp_temp.deptno%TYPE;
bad_job        emp_temp.job%TYPE;

  dml_errors  EXCEPTION;
PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
  -- Populate table:
  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:
  FORALL j IN dects.FIRST..dects.LAST SAVE EXCEPTIONS
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = dects(j);
  EXCEPTION
  WHEN dml_errors THEN
    FOR i IN 1..SQL\%BULK_EXCEPTIONS.COUNT LOOP
      error_message := SQLERRM(-SQL\%BULK_EXCEPTIONS(i).ERROR_CODE));
      DBMS_OUTPUT.PUT_LINE (error_message);

      bad_stmt_no := SQL\%BULK_EXCEPTIONS(i).ERROR_INDEX;
      DBMS_OUTPUT.PUT_LINE('Bad statement #: ' || bad_stmt_no);

      bad_deptno := dects(bad_stmt_no);
      DBMS_OUTPUT.PUT_LINE('Bad department #: ' || bad_deptno);

      SELECT job INTO bad_job FROM emp_temp WHERE deptno = bad_deptno;
    END LOOP;
END;
```
DBMS_OUTPUT.PUT_LINE('Bad job: ' || bad_job);
END LOOP;

COMMIT;  -- Commit results of successful updates

WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Unrecognized error.');
  RAISE;
END;
/

Result:
Procedure created.

Invoke procedure:
BEGIN
  p;
END;
/

Result:
ORA-12899: value too large for column  (actual: , maximum: )
Bad statement #: 2
Bad department #: 20
Bad job: Bookkeeper

PL/SQL procedure successfully completed.

Query:
SELECT * FROM emp_temp;

Result:

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>JOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Clerk (Senior)</td>
</tr>
<tr>
<td>20</td>
<td>Bookkeeper</td>
</tr>
<tr>
<td>30</td>
<td>Analyst (Senior)</td>
</tr>
</tbody>
</table>

3 rows selected.

13.4.1.4.1 Sparse Collections and SQL%BULK_EXCEPTIONS

If the FORALL statement bounds clause references a sparse collection, then to find the collection element that caused a DML statement to fail, you must step through the elements one by one until you find the element whose index is SQL%BULK_EXCEPTIONS(i).ERROR_INDEX. Then, if the FORALL statement uses the VALUES OF clause to reference a collection of pointers into another collection, you must find the element of the other collection whose index is SQL%BULK_EXCEPTIONS(i).ERROR_INDEX.
### 13.4.1.5 Getting Number of Rows Affected by FORALL Statement

After a `FORALL` statement completes, you can get the number of rows that each DML statement affected from the implicit cursor attribute `SQL%BULK_ROWCOUNT`.

To get the total number of rows affected by the `FORALL` statement, use the implicit cursor attribute `SQL%ROWCOUNT`, described in "SQL%ROWCOUNT Attribute: How Many Rows Were Affected?".

`SQL%BULK_ROWCOUNT` is like an associative array whose $i$th element is the number of rows affected by the $i$th DML statement in the most recently completed `FORALL` statement. The data type of the element is `INTEGER`.

**Note:**

If a server is Oracle Database 12c or later and its client is Oracle Database 11g release 2 or earlier (or the reverse), then the maximum number that `SQL%BULK_ROWCOUNT` returns is 4,294,967,295.

Example 13-14 uses `SQL%BULK_ROWCOUNT` to show how many rows each `DELETE` statement in the `FORALL` statement deleted and `SQL%ROWCOUNT` to show the total number of rows deleted.

Example 13-15 uses `SQL%BULK_ROWCOUNT` to show how many rows each `INSERT SELECT` construct in the `FORALL` statement inserted and `SQL%ROWCOUNT` to show the total number of rows inserted.

Example 13-14  **Showing Number of Rows Affected by Each DELETE in FORALL**

```sql
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(30, 50, 60);
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp WHERE department_id = depts(j);

  FOR i IN depts.FIRST..depts.LAST LOOP
    DBMS_OUTPUT.PUT_LINE ('Statement #' || i || ' deleted ' || SQL%BULK_ROWCOUNT(i) || ' rows.');
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Total rows deleted: ' || SQL%ROWCOUNT);
END;
/

Result:

Statement #1 deleted 6 rows.
Statement #2 deleted 45 rows.
```
Statement #3 deleted 5 rows.
Total rows deleted: 56

Example 13-15  Showing Number of Rows Affected by Each INSERT SELECT in FORALL

```sql
DROP TABLE emp_by_dept;
CREATE TABLE emp_by_dept AS
  SELECT employee_id, department_id
  FROM employees
  WHERE 1 = 0;

DECLARE
  TYPE dept_tab IS TABLE OF departments.department_id%TYPE;
  deptnums  dept_tab;
BEGIN
  SELECT department_id BULK COLLECT INTO deptnums FROM departments;
  FORALL i IN 1..deptnums.COUNT
    INSERT INTO emp_by_dept (employee_id, department_id)
    SELECT employee_id, department_id
    FROM employees
    WHERE department_id = deptnums(i)
    ORDER BY department_id, employee_id;
  FOR i IN 1..deptnums.COUNT LOOP
    -- Count how many rows were inserted for each department; that is,
    -- how many employees are in each department.
    DBMS_OUTPUT.PUT_LINE ('Dept '||deptnums(i)||': inserted '"||SQL%BULK_ROWCOUNT(i)||'" records'');
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total records inserted: '||SQL%ROWCOUNT);
END;
/
```

Result:

Dept 10: inserted 1 records
Dept 20: inserted 2 records
Dept 30: inserted 6 records
Dept 40: inserted 1 records
Dept 50: inserted 45 records
Dept 60: inserted 5 records
Dept 70: inserted 1 records
Dept 80: inserted 34 records
Dept 90: inserted 3 records
Dept 100: inserted 6 records
Dept 110: inserted 2 records
Dept 120: inserted 0 records
Dept 130: inserted 0 records
Dept 140: inserted 0 records
Dept 150: inserted 0 records
Dept 160: inserted 0 records
Dept 170: inserted 0 records
Dept 180: inserted 0 records
Dept 190: inserted 0 records
Dept 200: inserted 0 records
Dept 210: inserted 0 records
Dept 220: inserted 0 records
13.4.2 BULK COLLECT Clause

The BULK COLLECT clause, a feature of bulk SQL, returns results from SQL to PL/SQL in batches rather than one at a time.

The BULK COLLECT clause can appear in:

- SELECT INTO statement
- FETCH statement
- RETURNING INTO clause of:
  - DELETE statement
  - INSERT statement
  - UPDATE statement
  - EXECUTE IMMEDIATE statement

With the BULK COLLECT clause, each of the preceding statements retrieves an entire result set and stores it in one or more collection variables in a single operation (which is more efficient than using a loop statement to retrieve one result row at a time).

Note:

PL/SQL processes the BULK COLLECT clause similar to the way it processes a FETCH statement inside a LOOP statement. PL/SQL does not raise an exception when a statement with a BULK COLLECT clause returns no rows. You must check the target collections for emptiness, as in Example 13-22.

Topics

- SELECT INTO Statement with BULK COLLECT Clause
- FETCH Statement with BULK COLLECT Clause
- RETURNING INTO Clause with BULK COLLECT Clause

13.4.2.1 SELECT INTO Statement with BULK COLLECT Clause

The SELECT INTO statement with the BULK COLLECT clause (also called the SELECT BULK COLLECT INTO statement) selects an entire result set into one or more collection variables.

For more information, see "SELECT INTO Statement".
Caution:

The `SELECT BULK COLLECT INTO` statement is vulnerable to aliasing, which can cause unexpected results. For details, see "SELECT BULK COLLECT INTO Statements and Aliasing".

Example 13-16 uses a `SELECT BULK COLLECT INTO` statement to select two database columns into two collections (nested tables).

Example 13-17 uses a `SELECT BULK COLLECT INTO` statement to select a result set into a nested table of records.

Topics

- SELECT BULK COLLECT INTO Statements and Aliasing
- Row Limits for SELECT BULK COLLECT INTO Statements
- Guidelines for Looping Through Collections

Example 13-16  Bulk-Selecting Two Database Columns into Two Nested Tables

DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;
  enums NumTab;
  names NameTab;
  
  PROCEDURE print_first_n (n POSITIVE) IS
    BEGIN
      IF enums.COUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE ('Collections are empty.');
      ELSE
        DBMS_OUTPUT.PUT_LINE ('First ' || n || ' employees:');

        FOR i IN 1 .. n LOOP
          DBMS_OUTPUT.PUT_LINE ('Employee #' || enums(i) || ': ' || names(i));
        END LOOP;
      END IF;
    END;

BEGIN
  SELECT employee_id, last_name
  BULK COLLECT INTO enums, names
  FROM employees
  ORDER BY employee_id;

  print_first_n(3);
  print_first_n(6);
END;
/

Result:

First 3 employees:
Employee #100: King
Employee #101: Kochhar
Employee #102: De Haan
First 6 employees:
Employee #100: King
Employee #101: Kochhar
Employee #102: De Haan
Employee #103: Hunold
Employee #104: Ernst
Employee #105: Austin

Example 13-17   Bulk-Selecting into Nested Table of Records

DECLARE
   CURSOR c1 IS
      SELECT first_name, last_name, hire_date
         FROM employees;

   TYPE NameSet IS TABLE OF c1%ROWTYPE;

   stock_managers  NameSet;  -- nested table of records

BEGIN
   -- Assign values to nested table of records:
   SELECT first_name, last_name, hire_date
      BULK COLLECT INTO stock_managers
         FROM employees
      WHERE job_id = 'ST_MAN'
      ORDER BY hire_date;

   -- Print nested table of records:
   FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (stock_managers(i).hire_date || ' ' || stock_managers(i).last_name  || ', ' || stock_managers(i).first_name);
   END LOOP;END;

    Result:
01-MAY-03 Kaufling, Payam
18-JUL-04 Weiss, Matthew
10-APR-05 Fripp, Adam
10-OCT-05 Vollman, Shanta
16-NOV-07 Mourgos, Kevin

13.4.2.1.1 SELECT BULK COLLECT INTO Statements and Aliasing

In a statement of the form

SELECT column BULK COLLECT INTO collection FROM table ...

column and collection are analogous to IN NOCOPY and OUT NOCOPY subprogram parameters, respectively, and PL/SQL passes them by reference. As with subprogram parameters that are passed by reference, aliasing can cause unexpected results.
In Example 13-18, the intention is to select specific values from a collection, \texttt{numbers1}, and then store them in the same collection. The unexpected result is that all elements of \texttt{numbers1} are deleted. For workarounds, see Example 13-19 and Example 13-20.

Example 13-19 uses a cursor to achieve the result intended by Example 13-18.

Example 13-20 selects specific values from a collection, \texttt{numbers1}, and then stores them in a different collection, \texttt{numbers2}. Example 13-20 runs faster than Example 13-19.

Example 13-18  SELECT BULK COLLECT INTO Statement with Unexpected Results

```sql
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;
  --Self-selecting BULK COLLECT INTO clause:
  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers1
  FROM TABLE(numbers1) a
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;
  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
END p;
/

Invoke p:
BEGIN
  p(2);
END;
/

Result:
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
```
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0

PL/SQL procedure successfully completed.

Invoke p:

BEGIN
  p(10);
END;
/

Result:

Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0

Example 13-19  Cursor Workaround for Example 13-18

CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);

  CURSOR c IS
    SELECT a.COLUMN_VALUE
    FROM TABLE(numbers1) a
    WHERE a.COLUMN_VALUE > p.i
    ORDER BY a.COLUMN_VALUE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;
  OPEN c;
  FETCH c BULK COLLECT INTO numbers1;
  CLOSE c;
  DBMS_OUTPUT.PUT_LINE('After FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;
END p;
/
Invoke p:
BEGIN
  p(2);
END;
/

Result:

Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5

After FETCH statement
numbers1.COUNT() = 3
numbers1(1) = 3
numbers1(2) = 4
numbers1(3) = 5

Invoke p:
BEGIN
  p(10);
END;
/

Result:

Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5

After FETCH statement
numbers1.COUNT() = 0

Example 13-20 Second Collection Workaround for Example 13-18

CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1 numbers_type := numbers_type(1,2,3,4,5);
  numbers2 numbers_type := numbers_type(0,0,0,0,0);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());
  FOR j IN 1..numbers2.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
  END LOOP;
END;
```
DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ' || ' || numbers2(j));
END LOOP;

SELECT a.COLUMN_VALUE
BULK COLLECT INTO numbers2      -- numbers2 appears here
FROM TABLE(numbers1) a        -- numbers1 appears here
WHERE a.COLUMN_VALUE > p.i
ORDER BY a.COLUMN_VALUE;

DBMS_OUTPUT.PUT_LINE('After SELECT statement');
DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

IF numbers1.COUNT() > 0 THEN
  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;
END IF;

DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

IF numbers2.COUNT() > 0 THEN
  FOR j IN 1..numbers2.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
  END LOOP;
END IF;
END p;
/

Invoke p:

BEGIN
  p(2);
END;
/

Result:

Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0

After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 3
numbers2(1) = 3
numbers2(2) = 4
numbers2(3) = 5
```
PL/SQL procedure successfully completed.

Invoke p:
BEGIN
  p(10);
END;
/

Result:

Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0

After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 0

13.4.2.1.2 Row Limits for SELECT BULK COLLECT INTO Statements

A SELECT BULK COLLECT INTO statement that returns a large number of rows produces a large collection. To limit the number of rows and the collection size, use one of these:

- ROWNUM pseudocolumn (described in Oracle Database SQL Language Reference)
- SAMPLE clause (described in Oracle Database SQL Language Reference)
- FETCH FIRST clause (described in Oracle Database SQL Language Reference)

Example 13-21 shows several ways to limit the number of rows that a SELECT BULK COLLECT INTO statement returns.

Example 13-21 Limiting Bulk Selection with ROWNUM, SAMPLE, and FETCH FIRST

DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  sals SalList;
BEGIN
  SELECT salary BULK COLLECT INTO sals FROM employees
    WHERE ROWNUM <= 50;

  SELECT salary BULK COLLECT INTO sals FROM employees
    SAMPLE (10);

  SELECT salary BULK COLLECT INTO sals FROM employees
13.4.2.1.3 Guidelines for Looping Through Collections

When a result set is stored in a collection, it is easy to loop through the rows and refer to different columns. This technique can be very fast, but also very memory-intensive. If you use it often:

- To loop once through the result set, use a cursor FOR LOOP (see "Processing Query Result Sets With Cursor FOR LOOP Statements"). This technique avoids the memory overhead of storing a copy of the result set.
- Instead of looping through the result set to search for certain values or filter the results into a smaller set, do the searching or filtering in the query of the `SELECT INTO` statement.
  
  For example, in simple queries, use `WHERE` clauses; in queries that compare multiple result sets, use set operators such as `INTERSECT` and `MINUS`. For information about set operators, see *Oracle Database SQL Language Reference*.

- Instead of looping through the result set and running another query for each result row, use a subquery in the query of the `SELECT INTO` statement (see "Processing Query Result Sets with Subqueries").

- Instead of looping through the result set and running another DML statement for each result row, use the `FORALL` statement (see "FORALL Statement").

13.4.2.2 FETCH Statement with BULK COLLECT Clause

The `FETCH` statement with the `BULK COLLECT` clause (also called the `FETCH BULK COLLECT` statement) fetches an entire result set into one or more collection variables.

For more information, see "FETCH Statement".

**Example 13-22** uses a `FETCH BULK COLLECT` statement to fetch an entire result set into two collections (nested tables).

**Example 13-23** uses a `FETCH BULK COLLECT` statement to fetch a result set into a collection (nested table) of records.

**Example 13-22  Bulk-Fetching into Two Nested Tables**

```sql
DECLARE
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  TYPE SalList IS TABLE OF employees.salary%TYPE;

  CURSOR c1 IS
    SELECT last_name, salary
    FROM employees
    WHERE salary > 10000
    ORDER BY last_name;

  names  NameList;
  sals   SalList;

  TYPE RecList IS TABLE OF c1%ROWTYPE;
  recs  RecList;
```

Chapter 13
Bulk SQL and Bulk Binding

13-34
v_limit PLS_INTEGER := 10;

PROCEDURE print_results IS
BEGIN
  -- Check if collections are empty:
  IF names IS NULL OR names.COUNT = 0 THEN
    DBMS_OUTPUT.PUT_LINE('No results!');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Result: ');
    FOR i IN names.FIRST .. names.LAST LOOP
      DBMS_OUTPUT.PUT_LINE('  Employee ' || names(i) || ': $' || sals(i));
    END LOOP;
  END IF;
END;

BEGIN
  DBMS_OUTPUT.PUT_LINE ('--- Processing all results simultaneously ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO names, sals;
  CLOSE c1;
  print_results();
  DBMS_OUTPUT.PUT_LINE ('--- Processing ' || v_limit || ' rows at a time ---');
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO names, sals LIMIT v_limit;
    EXIT WHEN names.COUNT = 0;
    print_results();
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('--- Fetching records rather than columns ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO recs;
  FOR i IN recs.FIRST .. recs.LAST LOOP
    -- Now all columns from result set come from one record
    DBMS_OUTPUT.PUT_LINE ('  Employee ' || recs(i).last_name || ': $' || recs(i).salary);
  END LOOP;
END;
/

Result:
--- Processing all results simultaneously ---
Result:
Employee Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $10418.1
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $10036.95
Employee King: $24000
Employee Kochhar: $17000
Employee Ozer: $11500
Employee Partners: $13500
Example 13-23  Bulk-Fetching into Nested Table of Records

DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;

  TYPE NameSet IS TABLE OF c1%ROWTYPE;
  stock_managers  NameSet;  -- nested table of records

  TYPE cursor_var_type is REF CURSOR;
  cv cursor_var_type;

BEGIN
  -- Assign values to nested table of records:
OPEN cv FOR
  SELECT first_name, last_name, hire_date
  FROM employees
  WHERE job_id = 'ST_MAN'
  ORDER BY hire_date;

FETCH cv BULK COLLECT INTO stock_managers;
CLOSE cv;

-- Print nested table of records:

FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
  DBMS_OUTPUT.PUT_LINE (
    stock_managers(i).hire_date || ' ' ||
    stock_managers(i).last_name  || ', ' ||
    stock_managers(i).first_name
  );
END LOOP;
/

Result:

01-MAY-03 Kaufling, Payam
18-JUL-04 Weiss, Matthew
10-APR-05 Fripp, Adam
10-OCT-05 Vollman, Shanta
16-NOV-07 Mourgos, Kevin

13.4.2.2.1 Row Limits for FETCH BULK COLLECT Statements

A FETCH BULK COLLECT statement that returns a large number of rows produces a large collection. To limit the number of rows and the collection size, use the LIMIT clause.

In Example 13-24, with each iteration of the LOOP statement, the FETCH statement fetches ten rows (or fewer) into associative array empids (overwriting the previous values). Note the exit condition for the LOOP statement.

Example 13-24    Limiting Bulk FETCH with LIMIT

DECLARE
  TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  CURSOR c1 IS
    SELECT employee_id
    FROM employees
    WHERE department_id = 80
    ORDER BY employee_id;

  empids  numtab;
BEGIN
  OPEN c1;
  LOOP  -- Fetch 10 rows or fewer in each iteration
    FETCH c1 BULK COLLECT INTO empids LIMIT 10;
    DBMS_OUTPUT.PUT_LINE ('------- Results from One Bulk Fetch --------');
    FOR i IN 1..empids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE ('Employee Id: ' || empids(i));
    END LOOP;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;

Result:

-------- Results from One Bulk Fetch --------
Employee Id: 145
Employee Id: 146
Employee Id: 147
Employee Id: 148
Employee Id: 149
Employee Id: 150
Employee Id: 151
Employee Id: 152
Employee Id: 153
Employee Id: 154
-------- Results from One Bulk Fetch --------
Employee Id: 155
Employee Id: 156
Employee Id: 157
Employee Id: 158
Employee Id: 159
Employee Id: 160
Employee Id: 161
Employee Id: 162
Employee Id: 163
Employee Id: 164
-------- Results from One Bulk Fetch --------
Employee Id: 165
Employee Id: 166
Employee Id: 167
Employee Id: 168
Employee Id: 169
Employee Id: 170
Employee Id: 171
Employee Id: 172
Employee Id: 173
Employee Id: 174
-------- Results from One Bulk Fetch --------
Employee Id: 175
Employee Id: 176
Employee Id: 177
Employee Id: 179

13.4.2.3 RETURNING INTO Clause with BULK COLLECT Clause

The RETURNING INTO clause with the BULK COLLECT clause (also called the RETURNING BULK COLLECT INTO clause) can appear in an INSERT, UPDATE, DELETE, or EXECUTE IMMEDIATE statement. With the RETURNING BULK COLLECT INTO clause, the statement stores its result set in one or more collections.

For more information, see "RETURNING INTO Clause".

Example 13-25 uses a DELETE statement with the RETURNING BULK COLLECT INTO clause to delete rows from a table and return them in two collections (nested tables).

Example 13-25    Returning Deleted Rows in Two Nested Tables

DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id;

DECLARE
    TYPE NumList IS TABLE OF employees.employee_id%TYPE;
    enums  NumList;
    TYPE NameList IS TABLE OF employees.last_name%TYPE;
    names  NameList;
BEGIN
    DELETE FROM emp_temp
    WHERE department_id = 30
    RETURNING employee_id, last_name
    BULK COLLECT INTO enums, names;

    DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
    FOR i IN enums.FIRST .. enums.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE ('Employee #' || enums(i) || ': ' || names(i));
    END LOOP;
END;
/

Result:

Deleted 6 rows:
Employee #114: Raphaely
Employee #115: Khoo
Employee #116: Baida
Employee #117: Tobias
Employee #118: Himuro
Employee #119: Colmenares

13.4.3 Using FORALL Statement and BULK COLLECT Clause Together

In a FORALL statement, the DML statement can have a RETURNING BULK COLLECT INTO clause. For each iteration of the FORALL statement, the DML statement stores the specified values in the specified collections—without overwriting the previous values, as the same DML statement would do in a FOR LOOP statement.

In Example 13-26, the FORALL statement runs a DELETE statement that has a RETURNING BULK COLLECT INTO clause. For each iteration of the FORALL statement, the DELETE statement stores the employee_id and department_id values of the deleted row in the collections e_ids and d_ids, respectively.

Example 13-27 is like Example 13-26 except that it uses a FOR LOOP statement instead of a FORALL statement.

**Example 13-26  DELETE with RETURN BULK COLLECT INTO in FORALL Statement**

DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id, department_id;

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts  NumList := NumList(10,20,30);
TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
e_ids  enum_t;

TYPE dept_t IS TABLE OF employees.department_id%TYPE;
d_ids  dept_t;

BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');

  FOR i IN e_ids.FIRST .. e_ids.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Employee #' || e_ids(i) || ' from dept #' || d_ids(i));
  END LOOP;
END;
/

Result:
Deleted 9 rows:
Employee #200 from dept #10
Employee #201 from dept #20
Employee #202 from dept #20
Employee #114 from dept #30
Employee #115 from dept #30
Employee #116 from dept #30
Employee #117 from dept #30
Employee #118 from dept #30
Employee #119 from dept #30

Example 13-27    DELETE with RETURN BULK COLLECT INTO in FOR LOOP Statement

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts  NumList := NumList(10,20,30);

  TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
  e_ids  enum_t;

  TYPE dept_t IS TABLE OF employees.department_id%TYPE;
  d_ids  dept_t;

BEGIN
  FOR j IN depts.FIRST..depts.LAST LOOP
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');

  FOR i IN e_ids.FIRST .. e_ids.LAST
    DBMS_OUTPUT.PUT_LINE ('Employee #' || e_ids(i) || ' from dept #' || d_ids(i));
  END LOOP;
END;
/
13.4.4 Client Bulk-Binding of Host Arrays

Client programs (such as OCI and Pro*C programs) can use PL/SQL anonymous blocks to bulk-bind input and output host arrays. This is the most efficient way to pass collections to and from the database server.

In the client program, declare and assign values to the host variables to be referenced in the anonymous block. In the anonymous block, prefix each host variable name with a colon (:) to distinguish it from a PL/SQL collection variable name. When the client program runs, the database server runs the PL/SQL anonymous block.

In Example 13-28, the anonymous block uses a FORALL statement to bulk-bind a host input array. In the FORALL statement, the DELETE statement refers to four host variables: scalars lower, upper, and emp_id and array depts.

Example 13-28  Anonymous Block Bulk-Binds Input Host Array

BEGIN
   FORALL i IN :lower..:upper
      DELETE FROM employees
         WHERE department_id = :depts(i);
   END;
/

13.5 Chaining Pipelined Table Functions for Multiple Transformations

Chaining pipelined table functions is an efficient way to perform multiple transformations on data.
### 13.5.1 Overview of Table Functions

A **table function** is a user-defined PL/SQL function that returns a collection of rows (an associative array, nested table or varray).

You can select from this collection as if it were a database table by invoking the table function inside the `TABLE` clause in a `SELECT` statement. The `TABLE` operator is optional.

For example:

```sql
SELECT * FROM TABLE(table_function_name(parameter_list))
```

Alternatively, the same query can be written without the `TABLE` operator as follow:

```sql
SELECT * FROM table_function_name(parameter_list)
```

A table function can take a collection of rows as input (that is, it can have an input parameter that is a nested table, varray, or cursor variable). Therefore, output from table function `tf1` can be input to table function `tf2`, and output from `tf2` can be input to table function `tf3`, and so on.

To improve the performance of a table function, you can:

- Enable the function for parallel execution, with the `PARALLEL_ENABLE` option. Functions enabled for parallel execution can run concurrently.
- Stream the function results directly to the next process, with Oracle Streams. Streaming eliminates intermediate staging between processes.
▪ Pipeline the function results, with the `PIPELINED` option.

A **pipelined table function** returns a row to its invoker immediately after processing that row and continues to process rows. Response time improves because the entire collection need not be constructed and returned to the server before the query can return a single result row. (Also, the function needs less memory, because the object cache need not materialize the entire collection.)

⚠️ **Caution:**

A pipelined table function always references the current state of the data. If the data in the collection changes after the cursor opens for the collection, then the cursor reflects the changes. PL/SQL variables are private to a session and are not transactional. Therefore, read consistency, well known for its applicability to table data, does not apply to PL/SQL collection variables.

🔍 **See Also:**

▪ [Chaining Pipelined Table Functions](#)

▪ *Oracle Database SQL Language Reference* for more information about the TABLE clause of the SELECT statement

▪ *Oracle Database Data Cartridge Developer’s Guide* for information about using pipelined and parallel table functions

### 13.5.2 Creating Pipelined Table Functions

A pipelined table function must be either a standalone function or a package function.

**PIPELINED Option (Required)**

For a standalone function, specify the `PIPELINED` option in the `CREATE FUNCTION` statement (for syntax, see "CREATE FUNCTION Statement"). For a package function, specify the `PIPELINED` option in both the function declaration and function definition (for syntax, see "Function Declaration and Definition").

**PARALLEL_ENABLE Option (Recommended)**

To improve its performance, enable the pipelined table function for parallel execution by specifying the `PARALLEL_ENABLE` option.

**AUTONOMOUS_TRANSACTION Pragma**

If the pipelined table function runs DML statements, then make it autonomous, with the `AUTONOMOUS_TRANSACTION` pragma (described in "AUTONOMOUS_TRANSACTION Pragma"). Then, during parallel execution, each instance of the function creates an independent transaction.
DETERMINISTIC Option (Recommended)

Multiple invocations of a pipelined table function, in either the same query or separate queries, cause multiple executions of the underlying implementation. If the function is deterministic, specify the `DETERMINISTIC` option, described in "DETERMINISTIC Clause".

Parameters

Typically, a pipelined table function has one or more cursor variable parameters. For information about cursor variables as function parameters, see “Cursor Variables as Subprogram Parameters”.

See Also:

- “Cursor Variables” for general information about cursor variables
- “Subprogram Parameters” for general information about subprogram parameters

RETURN Data Type

The data type of the value that a pipelined table function returns must be a collection type defined either at schema level or inside a package (therefore, it cannot be an associative array type). The elements of the collection type must be SQL data types, not data types supported only by PL/SQL (such as `PLS_INTEGER` and `BOOLEAN`). For information about collection types, see "Collection Types". For information about SQL data types, see Oracle Database SQL Language Reference.

You can use SQL data types `ANYTYPE`, `ANYDATA`, and `ANYDATASET` to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these types to create unnamed types, including anonymous collection types. For information about these types, see Oracle Database PL/SQL Packages and Types Reference.

PIPE ROW Statement

Inside a pipelined table function, use the `PIPE ROW` statement to return a collection element to the invoker without returning control to the invoker. See “PIPE ROW Statement” for its syntax and semantics.

RETURN Statement

As in every function, every execution path in a pipelined table function must lead to a `RETURN` statement, which returns control to the invoker. However, in a pipelined table function, a `RETURN` statement need not return a value to the invoker. See "RETURN Statement" for its syntax and semantics.

Example

Example 13-29 Creating and Invoking Pipelined Table Function

This example creates a package that includes a pipelined table function, `f1`, and then selects from the collection of rows that `f1` returns.
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/

Create a pipelined table function f1 that returns a collection of elements (1,2,3,... x).

CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
    BEGIN
      FOR i IN 1..x LOOP
        PIPE ROW(i);
      END LOOP;
    RETURN;
    END f1;
END pkg1;

SELECT * FROM TABLE(pkg1.f1(5));

Result:

<table>
<thead>
<tr>
<th>COLUMN_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

5 rows selected.

SELECT * FROM pkg1.f1(2);

Result:

<table>
<thead>
<tr>
<th>COLUMN_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

13.5.3 Pipelined Table Functions as Transformation Functions

A pipelined table function with a cursor variable parameter can serve as a transformation function. Using the cursor variable, the function fetches an input row. Using the PIPE ROW statement, the function pipes the transformed row or rows to the invoker. If the FETCH and PIPE ROW statements are inside a LOOP statement, the function can transform multiple input rows.

In Example 13-30, the pipelined table function transforms each selected row of the employees table to two nested table rows, which it pipes to the SELECT statement that invokes it. The actual parameter that corresponds to the formal cursor variable parameter is a CURSOR expression; for information about these, see "Passing CURSOR Expressions to Pipelined Table Functions".

Example 13-30    Pipelined Table Function Transforms Each Row to Two Rows

CREATE OR REPLACE PACKAGE refcur_pkg AUTHID DEFINER IS
  TYPE refcur_t IS REF CURSOR RETURN employees%ROWTYPE;
  /
TYPE outrec_typ IS RECORD (  
  var_num NUMBER(6),  
  var_char1 VARCHAR2(30),  
  var_char2 VARCHAR2(30)  
);  
TYPE outrecset IS TABLE OF outrec_typ;  
FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED;  
END refcur_pkg;  
/

CREATE OR REPLACE PACKAGE BODY refcur_pkg IS  
FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED IS  
  out_rec outrec_typ;  
  in_rec p%ROWTYPE;  
BEGIN  
  LOOP  
    FETCH p INTO in_rec;  -- input row  
    EXIT WHEN p%NOTFOUND;  
    out_rec.var_num := in_rec.employee_id;  
    out_rec.var_char1 := in_rec.first_name;  
    out_rec.var_char2 := in_rec.last_name;  
    PIPE ROW(out_rec);  -- first transformed output row  
    out_rec.var_char1 := in_rec.email;  
    out_rec.var_char2 := in_rec.phone_number;  
    PIPE ROW(out_rec);  -- second transformed output row  
  END LOOP;  
  CLOSE p;  
  RETURN;  
END f_trans;  
END refcur_pkg;  
/

SELECT * FROM  
  TABLE (    
    refcur_pkg.f_trans (    
      CURSOR (SELECT * FROM employees WHERE department_id = 60)    
    )  
  );

Result:

<table>
<thead>
<tr>
<th>VAR_NUM</th>
<th>VAR_CHAR1</th>
<th>VAR_CHAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
</tr>
<tr>
<td>103</td>
<td>AHUNOLD</td>
<td>590.423.4567</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
</tr>
<tr>
<td>104</td>
<td>BERNST</td>
<td>590.423.4568</td>
</tr>
<tr>
<td>105</td>
<td>David</td>
<td>Austin</td>
</tr>
<tr>
<td>105</td>
<td>DAUSTIN</td>
<td>590.423.4569</td>
</tr>
<tr>
<td>106</td>
<td>Valli</td>
<td>Pataballa</td>
</tr>
<tr>
<td>106</td>
<td>VPATABAL</td>
<td>590.423.4560</td>
</tr>
<tr>
<td>107</td>
<td>Diana</td>
<td>Lorentz</td>
</tr>
<tr>
<td>107</td>
<td>DLORENTZ</td>
<td>590.423.5567</td>
</tr>
</tbody>
</table>

10 rows selected.
13.5.4 Chaining Pipelined Table Functions

To chain pipelined table functions \( tf_1 \) and \( tf_2 \) is to make the output of \( tf_1 \) the input of \( tf_2 \). For example:

\[
\text{SELECT * FROM TABLE}(\text{tf}_2(\text{CURSOR} (\text{SELECT * FROM \text{TABLE}(\text{tf}_1())))));
\]

The rows that \( tf_1 \) pipes out must be compatible actual parameters for the formal input parameters of \( tf_2 \).

If chained pipelined table functions are enabled for parallel execution, then each function runs in a different process (or set of processes).

See Also:
"Passing CURSOR Expressions to Pipelined Table Functions"

13.5.5 Fetching from Results of Pipelined Table Functions

You can associate a named cursor with a query that invokes a pipelined table function. Such a cursor has no special fetch semantics, and such a cursor variable has no special assignment semantics.

However, the SQL optimizer does not optimize across PL/SQL statements. Therefore, in Example 13-31, the first PL/SQL statement is slower than the second—despite the overhead of running two SQL statements in the second PL/SQL statement, and even if function results are piped between the two SQL statements in the first PL/SQL statement.

In Example 13-31, assume that \( f \) and \( g \) are pipelined table functions, and that each function accepts a cursor variable parameter. The first PL/SQL statement associates cursor variable \( r \) with a query that invokes \( f \), and then passes \( r \) to \( g \). The second PL/SQL statement passes CURSOR expressions to both \( f \) and \( g \).

See Also:
"Cursor Variables as Subprogram Parameters"

Example 13-31    Fetching from Results of Pipelined Table Functions

DECLARE
    r SYS_REFCURSOR;
...
    -- First PL/SQL statement (slower):
BEGIN
    OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM \text{tab})));
    SELECT * BULK COLLECT INTO rec\_tab FROM TABLE(g(r));

    -- NOTE: When \( g \) completes, it closes \( r \).
END;
13.5.6 Passing CURSOR Expressions to Pipelined Table Functions

As Example 13-31 shows, the actual parameter for the cursor variable parameter of a pipelined table function can be either a cursor variable or a CURSOR expression, and the latter is more efficient.

**Note:**

When a SQL SELECT statement passes a CURSOR expression to a function, the referenced cursor opens when the function begins to run and closes when the function completes.

**See Also:**

"CURSOR Expressions" for general information about CURSOR expressions

Example 13-32 creates a package that includes a pipelined table function with two cursor variable parameters and then invokes the function in a SELECT statement, using CURSOR expressions for actual parameters.

Example 13-33 uses a pipelined table function as an aggregate function, which takes a set of input rows and returns a single result. The SELECT statement selects the function result. (For information about the pseudocolumn COLUMN_VALUE, see Oracle Database SQL Language Reference.)

Example 13-32  Pipelined Table Function with Two Cursor Variable Parameters

```sql
CREATE OR REPLACE PACKAGE refcur_pkg AUTHID DEFINER IS
  TYPE refcur_t1 IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE refcur_t2 IS REF CURSOR RETURN departments%ROWTYPE;
  TYPE outrec_typ IS RECORD (  
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION g_trans (p1 refcur_t1, p2 refcur_t2) RETURN outrecset PIPELINED;
END refcur_pkg;
/

CREATE PACKAGE BODY refcur_pkg IS
  FUNCTION g_trans (  
    p1 refcur_t1,
    p2 refcur_t2
  ) RETURN outrecset PIPELINED
```
IS
out_rec outrec_typ;
in_rec1 p1%ROWTYPE;
in_rec2 p2%ROWTYPE;
BEGIN
LOOP
FETCH p2 INTO in_rec2;
EXIT WHEN p2%NOTFOUND;
END LOOP;
CLOSE p2;
LOOP
FETCH p1 INTO in_rec1;
EXIT WHEN p1%NOTFOUND;
-- first row
out_rec.var_num := in_rec1.employee_id;
out_rec.var_char1 := in_rec1.first_name;
out_rec.var_char2 := in_rec1.last_name;
PIPE ROW(out_rec);
-- second row
out_rec.var_num := in_rec2.department_id;
out_rec.var_char1 := in_rec2.department_name;
out_rec.var_char2 := TO_CHAR(in_rec2.location_id);
PIPE ROW(out_rec);
END LOOP;
CLOSE p1;
RETURN;
END g_trans;
END refcur_pkg;
/

SELECT * FROM TABLE (refcur_pkg.g_trans (CURSOR (SELECT * FROM employees WHERE department_id = 60), CURSOR (SELECT * FROM departments WHERE department_id = 60)));

Result:

<table>
<thead>
<tr>
<th>VAR_NUM</th>
<th>VAR_CHAR1</th>
<th>VAR_CHAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>105</td>
<td>David</td>
<td>Austin</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>106</td>
<td>Valli</td>
<td>Pataballa</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>107</td>
<td>Diana</td>
<td>Lorentz</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
</tbody>
</table>

10 rows selected.

Example 13-33   Pipelined Table Function as Aggregate Function

DROP TABLE gradereport;
CREATE TABLE gradereport (student VARCHAR2(30), subject VARCHAR2(30),
weight NUMBER,
grade NUMBER
);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Physics', 4, 4);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Chemistry', 4, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Maths', 3, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Economics', 3, 4);

CREATE PACKAGE pkg_gpa AUTHID DEFINER IS
  TYPE gpa IS TABLE OF NUMBER;
  FUNCTION weighted_average(input_values SYS_REFCURSOR)
  RETURN gpa PIPELINED;
END pkg_gpa;

CREATE PACKAGE BODY pkg_gpa IS
  FUNCTION weighted_average (input_values SYS_REFCURSOR)
  RETURN gpa PIPELINED
  IS
    grade         NUMBER;
    total         NUMBER := 0;
    total_weight  NUMBER := 0;
    weight        NUMBER := 0;
    BEGIN
      LOOP
        FETCH input_values INTO weight, grade;
        EXIT WHEN input_values%NOTFOUND;
        total_weight := total_weight + weight;  -- Accumulate weighted average
        total := total + grade*weight;
      END LOOP;
      PIPE ROW (total / total_weight);
      RETURN; -- returns single result
    END weighted_average;
END pkg_gpa;

This query shows how the table function can be invoked without the optional TABLE
operator.

SELECT w.column_value "weighted result"
FROM pkg_gpa.weighted_average (CURSOR (SELECT weight, grade FROM gradereport)) w;

Result:

weighted result
--------------
  3.5

1 row selected.
13.5.7 DML Statements on Pipelined Table Function Results

The “table” that a pipelined table function returns cannot be the target table of a DELETE, INSERT, UPDATE, or MERGE statement. However, you can create a view of such a table and create INSTEAD OF triggers on the view. For information about INSTEAD OF triggers, see "INSTEAD OF DML Triggers".

See Also:
Oracle Database SQL Language Reference for information about the CREATE VIEW statement

13.5.8 NO_DATA_NEEDED Exception

You must understand the predefined exception NO_DATA_NEEDED in two cases:

- You include an OTHERS exception handler in a block that includes a PIPE ROW statement
- Your code that feeds a PIPE ROW statement must be followed by a clean-up procedure

Typically, the clean-up procedure releases resources that the code no longer needs.

When the invoker of a pipelined table function needs no more rows from the function, the PIPE ROW statement raises NO_DATA_NEEDED. If the pipelined table function does not handle NO_DATA_NEEDED, as in Example 13-34, then the function invocation terminates but the invoking statement does not terminate. If the pipelined table function handles NO_DATA_NEEDED, its exception handler can release the resources that it no longer needs, as in Example 13-35.

In Example 13-34, the pipelined table function pipe_rows does not handle the NO_DATA_NEEDED exception. The SELECT statement that invokes pipe_rows needs only four rows. Therefore, during the fifth invocation of pipe_rows, the PIPE ROW statement raises the exception NO_DATA_NEEDED. The fifth invocation of pipe_rows terminates, but the SELECT statement does not terminate.

If the exception-handling part of a block that includes a PIPE ROW statement includes an OTHERS exception handler to handle unexpected exceptions, then it must also include an exception handler for the expected NO_DATA_NEEDED exception. Otherwise, the OTHERS exception handler handles the NO_DATA_NEEDED exception, treating it as an unexpected error. The following exception handler reraises the NO_DATA_NEEDED exception, instead of treating it as a irrecoverable error:

```
EXCEPTION
  WHEN NO_DATA_NEEDED THEN
    RAISE;
  WHEN OTHERS THEN
    -- (Put error-logging code here)
    RAISE_APPLICATION_ERROR(-20000, 'Fatal error.');
END;
```
In Example 13-35, assume that the package External_Source contains these public items:

- **Procedure** Init, which allocates and initializes the resources that Next_Row needs
- **Function** Next_Row, which returns some data from a specific external source and raises the user-defined exception Done (which is also a public item in the package) when the external source has no more data
- **Procedure** Clean_Up, which releases the resources that Init allocated

The pipelined table function get_external_source_data pipes rows from the external source by invoking External_Source.Next_Row until either:

- The external source has no more rows.
  - In this case, the External_Source.Next_Row function raises the user-defined exception External_Source.Done.
- get_external_source_data needs no more rows.
  - In this case, the PIPE ROW statement in get_external_source_data raises the NO_DATA_NEEDED exception.

In either case, an exception handler in block b in get_external_source_data invokes External_Source.Clean_Up, which releases the resources that Next_Row was using.

**Example 13-34  Pipelined Table Function Does Not Handle NO_DATA_NEEDED**

```
CREATE TYPE t IS TABLE OF NUMBER
/
CREATE OR REPLACE FUNCTION pipe_rows RETURN t PIPELINED AUTHID DEFINER IS
  n NUMBER := 0;
BEGIN
  LOOP
    n := n + 1;
    PIPE ROW (n);
  END LOOP;
END pipe_rows;
/
SELECT COLUMN_VALUE
FROM TABLE(pipe_rows())
WHERE ROWNUM < 5
/
```

Result:

```
COLUMN_VALUE
----------
    1
    2
    3
    4
```

4 rows selected.

**Example 13-35  Pipelined Table Function Handles NO_DATA_NEEDED**

```
CREATE OR REPLACE FUNCTION get_external_source_data
RETURN t PIPELINED AUTHID DEFINER IS
BEGIN
  External_Source.Init();           -- Initialize.
  <<b>> BEGIN
```
13.6 Overview of Polymorphic Table Functions

Polymorphic table functions (PTF) are table functions whose operands can have more than one type. The return type is determined by the PTF invocation arguments list. The actual arguments to the table type usually determines the row output shape, but not always.

Introduction to Polymorphic Table Functions

Polymorphic Table Functions (PTF) are user-defined functions that can be invoked in the FROM clause of a SQL query block. They are capable of processing tables whose row type is not declared at definition time and producing a result table whose row type may or may not be declared at definition time. Polymorphic table functions leverage dynamic SQL capabilities to create powerful and complex custom functions. This is useful for applications demanding an interface with generic extensions which work for arbitrary input tables or queries.

A PTF author creates an interface to a procedural mechanism that defines a table. The PTF author defines, documents, and implements the PTF.

The query author can only describe the published interface and invoke the PTF function in queries.

The database is the PTF conductor. It manages the compilation and execution states of the PTF. The database and the PTF author can see a family of related SQL invoked procedures, called the PTF component procedures, and possibly additional private data (such as variables and cursors).

Types of Polymorphic Table Functions

The polymorphic table function type is specified based on their formal arguments list semantics:

- If an input TABLE argument has Row Semantics, the input is a single row.
- If an input TABLE argument has Table Semantics, the input is a set of rows. When a Table Semantics PTF is called from a query, the table argument can optionally be extended with either a PARTITION BY clause or an ORDER BY clause or both.

13.6.1 Polymorphic Table Function Definition

The PTF author defines, documents, and implements the Polymorphic Table Function (PTF).

A PTF has two parts:
1. The PL/SQL package which contains the client interface for the PTF implementation.

2. The standalone or package function naming the PTF and its associated implementation package.

13.6.2 Polymorphic Table Function Implementation

The Polymorphic Table Function (PTF) implementation client interface is a set of subprograms with fixed names that every PTF must provide.

Steps to Implement a Polymorphic Table Function

1. Create the implementation package containing the DESCRIBE function (required) and the OPEN, FETCH_ROWS, and CLOSE procedures (optional).

2. Create the function specification naming the PTF. The function can be created at the top-level after the package has been created, or as a package function in the implementation package (the package created in the first step). Polymorphic table functions do not have a function definition (a FUNCTION BODY), the definition is encapsulated in the associated implementation package.

The function definition specifies:

• The Polymorphic Table Function (PTF) name
• Exactly one formal argument of type TABLE and any number of non TABLE arguments
• The return type of the PTF as TABLE
• The type of PTF function (row or table semantics)
• The PTF implementation package name

See Also:

- Oracle Database PL/SQL Packages and Types Reference for information about a DESCRIBE Only polymorphic table function
- Oracle Database PL/SQL Packages and Types Reference for more information about how to specify the PTF implementation package and use the DBMS_TF utilities
- PIPELINED Clause for the standalone or package polymorphic table function creation syntax and semantic

13.6.3 Polymorphic Table Function Invocation

A polymorphic table function is invoked by specifying its name followed by the argument list in the FROM clause of a SQL query block.

The PTF arguments can be the standard scalar arguments that can be passed to a regular table function, but PTF's can additionally take a table argument. A table argument is either a WITH clause query or a schema-level object that is allowed in a FROM clause (such as tables, views, or table functions).
Syntax

table_argument ::= table [ PARTITION BY column_list ] [ORDER BY order_column_list]

column_list ::= identifier | ( identifier[, identifier…] )

order_column_list ::= order_column_name | (order_column_name [, order_column_name…] )

order_column_name ::= identifier [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

Semantics

Each identifier is a column in the corresponding table.

The PTF has Table Semantics.

Query can optionally partition and order Table Semantics PTF input. This is disallowed for Row Semantics PTF input.

A polymorphic table function (PTF) cannot be the target of a DML statement. Any table argument of a PTF is passed in by name.

For example, the noop PTF can be used in a query such as:

SELECT *
FROM noop(emp);

or

WITH e AS
  (SELECT * FROM emp NATURAL JOIN dept)
SELECT t.* FROM noop(e) t;

The input table argument must be a basic table name.

The name resolution rules of the table identifier are (in priority order) as follows:

1. Identifier is resolved as a column name (such as a correlated column from an outer query block).
2. Identifier is resolved as a Common Table Expression (CTE) name in the current or some outer query-block. CTE is commonly known as the WITH clause.
3. Identifier is resolved as a schema-level table, view, or table-function (regular or polymorphic, and defined either at the schema-level or inside a package).

Many types of table expressions otherwise allowed in the FROM clause cannot be directly used as a table argument for a PTF (such as ANSI Joins, bind-variables, in-line views, CURSOR operators, TABLE operators). To use such table expressions as a PTF argument, these table expressions must be passed indirectly into a PTF by wrapping them in a CTE and then passing the CTE name into the PTF.

A PTF can be used as a table reference in the FROM clause and thus can be part of the ANSI Join and LATERAL syntax. Additionally, a PTF can be the source table for PIVOT/UNPIVOT and MATCH_RECOGNIZE. Some table modification clauses that are meant for tables and views (such as SAMPLING, PARTITION, CONTAINERS) are disallowed for PTF.
Direct function composition of PTF is allowed (such as nested PTF cursor expression invocation or PTF(TF()) nesting). However, nested PTF is disallowed (such as PTF(PTF()) nesting).

The scalar arguments of a PTF can be any SQL scalar expression. While the constant scalar values are passed as-is to the DESCRIBE function, all other values are passed as NULLs. This is usually not a problem for the PTF implementation if these values are not row shape determining, but otherwise the DESCRIBE function can raise an error; typically the documentation accompanying the PTF will state which scalar parameters, if any, are shape defining and thus must have constant non-null values. Note, that during query execution (during OPEN, FETCH_ROWS, CLOSE) the expressions are evaluated and their actual values are passed to these PTF execution procedures. The return type is determined by the PTF invocation arguments list.

Query arguments are passed to PTF using a WITH clause.

The TABLE operator is optional when the table function arguments list or empty list () appears.

13.6.3.1 Variadic Pseudo-Operators

A variadic pseudo-operator operates with a variable number of operands.

Starting with Oracle Database Release 18c, we introduce the concept of variadic pseudo-operator into the SQL expression language to support Polymorphic Table Functions (PTF). A pseudo-operator can be used to pass list of identifiers (such as column name) to a PTF. A pseudo-operator can only appear as arguments to PTFs, and are parsed by the SQL compiler like other SQL operators or PL/SQL function invocation. A pseudo-operator has a variable number of arguments but must have at least one. The pseudo-operator does not have any execution function associated with it, and they are completely removed from the SQL cursor after the PTF compilation is finished. During SQL compilation, the pseudo-operators are converted to corresponding DBMS_TF types and then passed to the DESCRIBE method. There is no output type associated with these operators. It is not possible to embed a pseudo-operator inside a general SQL expression.

13.6.3.2 COLUMNS Pseudo-Operator

You can use the COLUMNS pseudo-operator to specify arguments to a Polymorphic Table Function (PTF) invocation in the FROM clause of a SQL query block.

The COLUMNS pseudo-operator arguments specify the list of column names, or the list of column names with associated types.

Syntax

\[
\text{column_operator ::= COLUMNS ( column_list )}
\]

\[
\text{column_list ::= column_name_list | column_type_list}
\]

\[
\text{column_name_list ::= identifier [ , identifier ... ]}
\]

\[
\text{column_type_list ::= identifier column_type [ , identifier column_type ... ]}
\]

Semantics

The COLUMNS pseudo-operator can only appear as an argument to a PTF. It cannot appear in any other SQL expression than the PTF expression itself.
The column_type must be a scalar type.

13.6.4 Polymorphic Table Function Compilation and Execution

The database fulfills the Polymorphic Table Functions (PTF) conductor role. As such, it is responsible for the PTF compilation, execution and its related states.

The database manages:

- The compilation state: This is the immutable state that is generated by `DESCRIBE` which is needed before execution.
- The execution state: This is the state used by the execution procedures of a Table semantics PTF.

See Also:

- Oracle Database PL/SQL Packages and Types Reference for more information about how the database manages the compilation and execution states of the PTFs

13.6.5 Polymorphic Table Function Optimization

A polymorphic table function (PTF) provides an efficient and scalable mechanism to extend the analytical capabilities of the database.

The key benefits are:

- Minimal data-movement: Only columns of interest are passed to PTF
- Predicates/Projections/Partitioning are/is pushed into underlying table/query (where semantically possible)
- Bulk data transfer into and out of PTF
- Parallelism is based on type of PTF and query specified partitioning (if any)

13.6.6 Skip_col Polymorphic Table Function Example

This PTF example demonstrates Row Semantics, Describe Only, package table function, and overloading features.

See Also:

Oracle Database PL/SQL Packages and Types Reference for more Polymorphic Table Function (PTF) examples
Example 13-36  Skip_col Polymorphic Table Function Example

The skip_col Polymorphic Table Function (PTF) returns all the columns in a table except the columns specified in the PTF input argument. The skip_col PTF skips columns based on column names (overload 1) or columns data type (overload 2).

Live SQL:

You can view and run this example on Oracle Live SQL at 18c Skip_col Polymorphic Table Function

Create the implementation package named skip_col_pkg containing the DESCRIBE function for the skip_col polymorphic table function (PTF). The DESCRIBE function is invoked to determine the row shape produced by the PTF. It returns a DBMS_TF.DESCRIBE_T table. It is overloaded. The FETCH_ROWS procedure is not required because it does need to produce associated new column values for a given subset of rows.

CREATE PACKAGE skip_col_pkg AS

-- OVERLOAD 1: Skip by name --
FUNCTION skip_col(tab TABLE, 
                   col COLUMNS) 
    RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;

FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T, 
                   col        DBMS_TF.COLUMNS_T) 
    RETURN DBMS_TF.DESCRIBE_T;

-- OVERLOAD 2: Skip by type --
FUNCTION skip_col(tab       TABLE, 
                   type_name VARCHAR2, 
                   flip      VARCHAR2 DEFAULT 'False') 
    RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;

FUNCTION describe(tab       IN OUT DBMS_TF.TABLE_T, 
                   type_name  VARCHAR2, 
                   flip       VARCHAR2 DEFAULT 'False') 
    RETURN DBMS_TF.DESCRIBE_T;

END skip_col_pkg;

Create the implementation package body which contains the polymorphic table function definition.

CREATE PACKAGE BODY skip_col_pkg AS

/* OVERLOAD 1: Skip by name
 Package PTF name:  skip_col_pkg.skip_col
 Standalone PTF name: skip_col_by_name
 */
* PARAMETERS:
* tab – The input table
* col – The name of the columns to drop from the output

* DESCRIPTION:
* This PTF removes all the input columns listed in col from the output
* of the PTF.

/*
FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T,
    col DBMS_TF.COLUMNS_T)
RETURN DBMS_TF.DESCRIBE_T
AS
    new_cols DBMS_TF.COLUMNS_NEW_T;
    col_id   PLS_INTEGER := 1;
BEGIN
    FOR i IN 1 .. tab.column.count() LOOP
        FOR j IN 1 .. col.count() LOOP
            tab.column(i).PASS_THROUGH := tab.column(i).DESCRIPTION.NAME !=
                col(j);
            EXIT WHEN NOT tab.column(i).PASS_THROUGH;
        END LOOP;
    END LOOP;
RETURN NULL;
END;

/* OVERLOAD 2: Skip by type
* Package PTF name:  skip_col_pkg.skip_col
* Standalone PTF name: skip_col_by_type
* PARAMETERS:
*   tab       - Input table
*   type_name - A string representing the type of columns to skip
*   flip      - 'False' [default] => Match columns with given type_name
*               otherwise         => Ignore columns with given type_name
* DESCRIPTION:
*   This PTF removes the given type of columns from the given table.
*/
FUNCTION describe(tab       IN OUT DBMS_TF.TABLE_T,
    type_name        VARCHAR2,
    flip             VARCHAR2 DEFAULT 'False')
RETURN DBMS_TF.DESCRIBE_T
AS
    typ CONSTANT VARCHAR2(1024) := UPPER(TRIM(type_name));
BEGIN
    FOR i IN 1 .. tab.column.count() LOOP
        tab.column(i).PASS_THROUGH :=
            CASE UPPER(SUBSTR(flip,1,1))
            WHEN 'F' THEN
                DBMS_TF.column_type_name(tab.column(i).DESCRIPTION)!=typ
            ELSE
                DBMS_TF.column_type_name(tab.column(i).DESCRIPTION) =typ
            END /* case */;
    END LOOP;
CREATE FUNCTION skip_col_by_name(tab TABLE, col COLUMNS)
  RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;

CREATE FUNCTION skip_col_by_type(tab TABLE, type_name VARCHAR2, flip VARCHAR2 DEFAULT 'False')
  RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;

Invoke the package skip_col PTF (overload 1) to report from the SCOTT.DEPT table only columns whose type is not NUMBER.

SELECT * FROM skip_col_pkg.skip_col(scott.dept, 'number');

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON

The same result can be achieved by invoking the standalone skip_col_by_type PTF to report from the SCOTT.DEPT table only columns whose type is not NUMBER.

SELECT * FROM skip_col_by_type(scott.dept, 'number');

DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON

RETURN NULL;
END;

END skip_col_pkg;
Invoke the package skip_col PTF (overload 2) to report from the SCOTT.DEPT table only columns whose type is NUMBER.

```
SELECT * FROM skip_col_pkg.skip_col(scott.dept, 'number', flip => 'True');
```

<table>
<thead>
<tr>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>40</td>
</tr>
</tbody>
</table>

The same result can be achieved by invoking the standalone skip_col_by_type PTF to report from the SCOTT.DEPT table only columns whose type is NUMBER.

```
SELECT * FROM skip_col_by_type(scott.dept, 'number', flip => 'True');
```

<table>
<thead>
<tr>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>40</td>
</tr>
</tbody>
</table>

Invoke the package skip_col PTF to report all employees in department 20 from the SCOTT.EMP table all columns except COMM, HIREDATE and MGR.

```
SELECT *
FROM skip_col_pkg.skip_col(scott.emp, COLUMNS(comm, hiredate, mgr))
WHERE deptno = 20;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>SAL</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800</td>
<td>20</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975</td>
<td>20</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000</td>
<td>20</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100</td>
<td>20</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000</td>
<td>20</td>
</tr>
</tbody>
</table>

**13.6.7 To_doc Polymorphic Table Function Example**

The to_doc PTF example combines a list of specified columns into a single document column.

**Example 13-37  To_doc Polymorphic Table Function Example**

The to_doc PTF combines a list of columns into a document column constructed like a JSON object.

**Live SQL:**

You can view and run this example on Oracle Live SQL at 18c To_doc Polymorphic Table Function
Create the implementation package to_doc_p containing the **DESCRIBE** function and **FETCH_ROWS** procedure for the to_doc polymorphic table function (PTF).

The PTF parameters are:

- **tab**: The input table (The tab parameter is of type `DBMS_TF.TABLE_T`, a table descriptor record type)
- **cols** (optional): The list of columns to convert to document. (The cols parameter is type `DBMS_TF.COLUMNS_T`, a column descriptor record type)

```sql
CREATE PACKAGE to_doc_p AS
  FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T,
                     cols IN     DBMS_TF.COLUMNS_T DEFAULT NULL) RETURN DBMS_TF.DESCRIBE_T;

  PROCEDURE fetch_rows;
END to_doc_p;
END;
```

Create the package containing the **DESCRIBE** function and **FETCH_ROWS** procedure. The **FETCH_ROWS** procedure is required to produce a new column named DOCUMENT in the output rowset. The **DESCRIBE** function indicates the read columns by annotating them in the input table descriptor, `TABLE_T`. Only the indicated read columns will be fetched and thus available for processing during **FETCH_ROWS**. The PTF invocation in a query can use the `COLUMNS` pseudo-operator to indicate which columns the query wants the PTF to read, and this information is passed to the **DESCRIBE** function which then in turn sets the `COLUMN_T.FOR_READ` boolean flag. Only scalar SQL data types are allowed for the read columns. The `COLUMN_T.PASS_THROUGH` boolean flag indicates columns that are passed from the input table of the PTF to the output, without any modifications.

```sql
CREATE PACKAGE BODY to_doc_p AS
  FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T,
                    cols IN     DBMS_TF.COLUMNS_T DEFAULT NULL) RETURN DBMS_TF.DESCRIBE_T AS
    BEGIN
      FOR i IN 1 .. tab.column.count LOOP
        CONTINUE WHEN NOT DBMS_TF.SUPPORTED_TYPE(tab.column(i).DESCRIPTION.TYPE);
        IF cols IS NULL THEN
          tab.column(i).FOR_READ     := TRUE;
          tab.column(i).PASS_THROUGH := FALSE;
          CONTINUE;
        END IF;
        FOR j IN 1 .. cols.count LOOP
          IF (tab.column(i).DESCRIPTION.NAME = cols(j)) THEN
            tab.column(i).FOR_READ     := TRUE;
            tab.column(i).PASS_THROUGH := FALSE;
          END IF;
        END LOOP;
      END LOOP;
    END LOOP;
```

Chapter 13
Overview of Polymorphic Table Functions
create function to_doc(
tab table,
cols columns default null)
return table
pipelined row polymorphic using to_doc_p;

invoke the to_doc PTF to display all columns of table scott.dept as one combined DOCUMENT column.

SELECT * FROM to_doc(scott.dept);

DOCUMENT
----------------------------------
{"DEPTNO":10, "DNAME":"ACCOUNTING", "LOC":"NEW YORK"}
{"DEPTNO":20, "DNAME":"RESEARCH", "LOC":"DALLAS"}
{"DEPTNO":30, "DNAME":"SALES", "LOC":"CHICAGO"}
{"DEPTNO":40, "DNAME":"OPERATIONS", "LOC":"BOSTON"}

for all employees in departments 10 and 30, display the DEPTNO, ENAME and DOCUMENT columns ordered by DEPTNO and ENAME. invoke the to_doc PTF with the columns pseudo-operator to select columns EMPNO, JOB, MGR, HIREDATE, SAL and COMM of table scott.emp . The PTF combines these columns into the DOCUMENT column.

SELECT deptno, ename, document
FROM   to_doc(scott.emp, columns(empno,job,mgr,hiredate,sal,comm))
WHERE  deptno in (10, 30)
ORDER BY 1, 2;
### Overview of Polymorphic Table Functions

#### Chapter 13

**With the subquery named E, display the DOC_ID and DOCUMENT columns. Report all clerk employees, their salary, department and department location. Use the to_doc PTF to combine the NAME, SAL, DEPTNO and LOC columns into the DOCUMENT column.**

```sql
WITH e AS (
    SELECT ename name, sal, deptno, loc
    FROM scott.emp NATURAL JOIN scott.dept
    WHERE job = 'CLERK'
)
SELECT ROWNUM doc_id, t.*
FROM to_doc(e) t;
```

**With the subquery block to display c1, c2, c3 column values converted into the DOCUMENT column.**

```sql
WITH t(c1,c2,c3) AS (
    SELECT NULL, NULL, NULL FROM dual
    UNION ALL
    SELECT 1, NULL, NULL FROM dual
    UNION ALL
    SELECT NULL, 2, NULL FROM dual
    UNION ALL
    SELECT 0, NULL, 3 FROM dual
)
SELECT *
FROM to_doc(t);
```
For all employees in department 30, display the values of the member with property names ENAME and COMM. The PTF invocation reporting from the SCOTT.EMP table produces the DOCUMENT column which can be used as input to the JSON_VALUE function. This function selects a scalar value from some JSON data.

```sql
SELECT JSON_VALUE(document, '$.ENAME') ename,
       JSON_VALUE(document, '$.COMM') comm
FROM   to_doc(scott.emp)
WHERE  JSON_VALUE(document, '$.DEPTNO') = 30;
```

<table>
<thead>
<tr>
<th>ENAME</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLEN</td>
<td>300</td>
</tr>
<tr>
<td>WARD</td>
<td>500</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1400</td>
</tr>
<tr>
<td>BLAKE</td>
<td></td>
</tr>
<tr>
<td>TURNER</td>
<td>0</td>
</tr>
<tr>
<td>JAMES</td>
<td></td>
</tr>
</tbody>
</table>

### 13.6.8 Implicit_echo Polymorphic Table Function Example

The implicit_echo PTF example demonstrates that the USING clause is optional when the Polymorphic Table Function and the DESCRIBE function are defined in the same package.

**Example 13-38 Implicit_echo Polymorphic Table Function Example**

The implicit_echo PTF, takes in a table and a column and produces a new column with the same value.

This PTF returns the column in the input table tab, and adds to it the column listed in cols but with the column names prefixed with "ECHO_".

Create the implementation package `implicit_echo_package` containing the DESCRIBE function, `implicit_echo` polymorphic table function (PTF) and `FETCH_ROWS` procedure.

```sql
CREATE PACKAGE implicit_echo_package AS
    prefix   DBMS_ID := '"ECHO_';

    FUNCTION DESCRIBE(tab   IN OUT DBMS_TF.TABLE_T,
                       cols  IN     DBMS_TF.COLUMNS_T)
               RETURN DBMS_TF.DESCRIBE_T;

    PROCEDURE FETCH_ROWS;

    -- PTF FUNCTION: WITHOUT USING CLAUSE --
    FUNCTION implicit_echo(tab TABLE, cols COLUMNS)
        RETURN TABLE PIPELINED ROW POLYMORPHIC;

END implicit_echo_package;
```
Create the package containing the DESCRIBE function containing the input table parameter and the column parameter to be read. This function is invoked to determine the type of rows produced by the Polymorphic Table Function. The function returns a table DBMS_TF.DESCRIBE_T. The FETCH_ROWS procedure is required to produce the indicated read column along with a new column prefixed with "ECHO_" in the output rowset. The implicit_echo is the PTF function and contains two arguments, tab and cols, whose values are obtained from the query and this information is passed to the DESCRIBE function. The Row semantics specifies a PTF type but without the USING clause. This function is invoked from the SQL query.

Create the implementation package body implicit_echo_package which contains the PTF definition.

```
CREATE PACKAGE BODY implicit_echo_package AS

FUNCTION DESCRIBE(tab IN OUT DBMS_TF.TABLE_T,
                    cols IN      DBMS_TF.COLUMNS_T)
    RETURN DBMS_TF.DESCRIBE_T
AS
  new_cols DBMS_TF.COLUMNS_NEW_T;
  col_id   PLS_INTEGER := 1;
BEGIN
  FOR i in 1 .. tab.column.COUNT LOOP
    FOR j in 1 .. cols.COUNT LOOP
      IF (tab.column(i).description.name = cols(j)) THEN
        IF (NOT DBMS_TF.SUPPORTED_TYPE(tab.column(i).description.type)) THEN
          RAISE_APPLICATION_ERROR(-20102, 'Unsupported column type['||
            tab.column(i).description.type||']');
        END IF;
        tab.column(i).for_read := TRUE;
        new_cols(col_id)       := tab.column(i).description;
        new_cols(col_id).name  := prefix||
          REGEXP_REPLACE(tab.column(i).description.name,
            '^"|"$');
        col_id                 := col_id + 1;
        EXIT;
      END IF;
    END LOOP;
  END LOOP;

  /* VERIFY ALL COLUMNS WERE FOUND */
  IF (col_id - 1 != cols.COUNT) then
    RAISE_APPLICATION_ERROR(-20101,'Column mismatch['||col_id-1||',
      ['||cols.COUNT||']]');
  END IF;
END IF;
```
RETURN DBMS_TF.DESCRIBE_T(new_columns => new_cols);
END;

PROCEDURE FETCH_ROWS AS
    rowset DBMS_TF.ROW_SET_T;
BEGIN
    DBMS_TF.GET_ROW_SET(rowset);
    DBMS_TF.PUT_ROW_SET(rowset);
END;
END implicit_echo_package;

Invoke the PTF to display ENAME column of table SCOTT.EMP and display it along with another column ECHO_ENAME having the same value.

SELECT ENAME, ECHO_ENAME
FROM implicit_echo_package.implicit_echo(SCOTT.EMP, COLUMNS(SCOTT.ENAME));

<table>
<thead>
<tr>
<th>ENAME</th>
<th>ECHO_ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>SMITH</td>
</tr>
<tr>
<td>ALLEN</td>
<td>ALLEN</td>
</tr>
<tr>
<td>WARD</td>
<td>WARD</td>
</tr>
<tr>
<td>JONES</td>
<td>JONES</td>
</tr>
<tr>
<td>MARTIN</td>
<td>MARTIN</td>
</tr>
<tr>
<td>BLAKE</td>
<td>BLAKE</td>
</tr>
<tr>
<td>CLARK</td>
<td>CLARK</td>
</tr>
<tr>
<td>SCOTT</td>
<td>SCOTT</td>
</tr>
<tr>
<td>KING</td>
<td>KING</td>
</tr>
<tr>
<td>TURNER</td>
<td>TURNER</td>
</tr>
<tr>
<td>ADAMS</td>
<td>ADAMS</td>
</tr>
<tr>
<td>JAMES</td>
<td>JAMES</td>
</tr>
<tr>
<td>FORD</td>
<td>FORD</td>
</tr>
<tr>
<td>MILLER</td>
<td>MILLER</td>
</tr>
</tbody>
</table>

### 13.7 Updating Large Tables in Parallel

The DBMS_PARALLEL_EXECUTE package lets you incrementally update the data in a large table in parallel, in two high-level steps:

1. Group sets of rows in the table into smaller chunks.
2. Apply the desired UPDATE statement to the chunks in parallel, committing each time you have finished processing a chunk.

This technique is recommended whenever you are updating a lot of data. Its advantages are:

- You lock only one set of rows at a time, for a relatively short time, instead of locking the entire table.
- You do not lose work that has been done if something fails before the entire operation finishes.
- You reduce rollback space consumption.
• You improve performance.

See Also:
Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_PARALLEL_EXECUTE package

13.8 Collecting Data About User-Defined Identifiers
PL/Scope extracts, organizes, and stores data about PL/SQL and SQL identifiers and SQL statements from PL/SQL source text. You can retrieve the identifiers and statements data with the static data dictionary views *_IDENTIFIERS and *_STATEMENTS.

See Also:
• PL/SQL Units and Compilation Parameters for more information about PLSQL_SETTINGS parameter
• Oracle Database Development Guide for more information about using PL/Scope

13.9 Profiling and Tracing PL/SQL Programs
To help you isolate performance problems in large PL/SQL programs, PL/SQL provides these tools, implemented as PL/SQL packages.

Table 13-1 Profiling and Tracing Tools Summary

<table>
<thead>
<tr>
<th>Tool</th>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiler interface</td>
<td>DBMS_PROFILER</td>
<td>Computes the time that your PL/SQL program spends at each line and in each subprogram. You must have CREATE privileges on the units to be profiled. Saves runtime statistics in database tables, which you can query.</td>
</tr>
<tr>
<td>Trace interface</td>
<td>DBMS_TRACE</td>
<td>Traces the order in which subprograms run. You can specify the subprograms to trace and the tracing level. Saves runtime statistics in database tables, which you can query.</td>
</tr>
</tbody>
</table>
## (Cont.) Profiling and Tracing Tools Summary

<table>
<thead>
<tr>
<th>Tool</th>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/SQL hierarchical profiler</td>
<td>DBMS_HPROF</td>
<td>Reports the dynamic execution program profile of your PL/SQL program, organized by subprogram invocations. Accounts for SQL and PL/SQL execution times separately. Requires no special source or compile-time preparation. Generates reports in HTML. Provides the option of storing profiler data and results in relational format in database tables for custom report generation (such as third-party tools offer).</td>
</tr>
<tr>
<td>SQL trace</td>
<td>DBMS_APPLICATION_INFO</td>
<td>Uses the DBMS_APPLICATION_INFO package with Oracle Trace and the SQL trace facility to record names of executing modules or transactions in the database for later use when tracking the performance of various modules and debugging.</td>
</tr>
<tr>
<td>PL/SQL Basic Block Coverage</td>
<td>DBMS_PLSQL_CODE_COVERAGE</td>
<td>Collects and analyzes basic block coverage data.</td>
</tr>
<tr>
<td>Call Stack Utilities</td>
<td>UTL_CALL_STACK</td>
<td>Provides information about currently executing subprograms (such as subprogram names, unit names, owner names, edition names, and error stack information) that you can use to create more revealing error logs and application execution traces.</td>
</tr>
</tbody>
</table>

### Related Topics
- Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_APPLICATION_INFO package
- Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_HPROF package
- Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_PLSQL_CODE_COVERAGE package
- Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_PROFILER package
- Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_TRACE package
- Oracle Database PL/SQL Packages and Types Reference for more information about the UTL_CALL_STACK package
- COVERAGE Pragma for the syntax and semantics of COVERAGE PRAGMA
- Oracle Database Development Guide for more information about using PL/SQL basic block coverage
- Oracle Database Development Guide for a detailed description of PL/SQL hierarchical profiler
You can usually speed up PL/SQL units by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace.

You can natively compile any PL/SQL unit of any type, including those that Oracle Database supplies.

Natively compiled program units work in all server environments, including shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

On most platforms, PL/SQL native compilation requires no special set-up or maintenance. On some platforms, the DBA might want to do some optional configuration.

---

**13.10 Compiling PL/SQL Units for Native Execution**

You can test to see how much performance gain you can get by enabling PL/SQL native compilation.

If you have determined that PL/SQL native compilation will provide significant performance gains in database operations, Oracle recommends compiling the entire database for native mode, which requires DBA privileges. This speeds up both your own code and calls to the PL/SQL packages that Oracle Database supplies.

**Topics**

- Determining Whether to Use PL/SQL Native Compilation
- How PL/SQL Native Compilation Works
- Dependencies, Invalidation, and Revalidation
- Setting Up a New Database for PL/SQL Native Compilation*
- Compiling the Entire Database for PL/SQL Native or Interpreted Compilation*

* Requires DBA privileges.

**13.10.1 Determining Whether to Use PL/SQL Native Compilation**

Whether to compile a PL/SQL unit for native or interpreted mode depends on where you are in the development cycle and on what the program unit does.

While you are debugging program units and recompiling them frequently, interpreted mode has these advantages:
• You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).

• Compiling for interpreted mode is faster than compiling for native mode.

After the debugging phase of development, in determining whether to compile a PL/SQL unit for native mode, consider:

• PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations. Examples are data warehouse applications and applications with extensive server-side transformations of data for display.

• PL/SQL native compilation provides the least performance gains for PL/SQL subprograms that spend most of their time running SQL.

• When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

13.10.2 How PL/SQL Native Compilation Works

Without native compilation, the PL/SQL statements in a PL/SQL unit are compiled into an intermediate form, system code, which is stored in the catalog and interpreted at run time.

With PL/SQL native compilation, the PL/SQL statements in a PL/SQL unit are compiled into native code and stored in the catalog. The native code need not be interpreted at run time, so it runs faster.

Because native compilation applies only to PL/SQL statements, a PL/SQL unit that uses only SQL statements might not run faster when natively compiled, but it does run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is the same.

The first time a natively compiled PL/SQL unit runs, it is fetched from the SYSTEM tablespace into shared memory. Regardless of how many sessions invoke the program unit, shared memory has only one copy it. If a program unit is not being used, the shared memory it is using might be freed, to reduce memory load.

Natively compiled subprograms and interpreted subprograms can invoke each other.

PL/SQL native compilation works transparently in an Oracle Real Application Clusters (Oracle RAC) environment.

The PLSQL_CODE_TYPE compilation parameter determines whether PL/SQL code is natively compiled or interpreted. For information about this compilation parameters, see "PL/SQL Units and Compilation Parameters".

13.10.3 Dependencies, Invalidation, and Revalidation

Recompilation is automatic with invalidated PL/SQL modules. For example, if an object on which a natively compiled PL/SQL subprogram depends changes, the subprogram is invalidated. The next time the same subprogram is called, the database recompiles the subprogram automatically. Because the PLSQL_CODE_TYPE setting is stored inside the library unit for each subprogram, the automatic recompilation uses this stored setting for code type.
Explicit recompilation does not necessarily use the stored `PLSQL_CODE_TYPE` setting. For the conditions under which explicit recompilation uses stored settings, see "PL/SQL Units and Compilation Parameters".

### 13.10.4 Setting Up a New Database for PL/SQL Native Compilation

If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the compilation parameter `PLSQL_CODE_TYPE` to `NATIVE`. The performance benefits apply to the PL/SQL packages that Oracle Database supplies, which are used for many database operations.

**Note:**

If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

### 13.10.5 Compiling the Entire Database for PL/SQL Native or Interpreted Compilation

If you have DBA privileges, you can recompile all PL/SQL modules in an existing database to `NATIVE` or `INTERPRETED`, using the `dbmsupgnv.sql` and `dbmsupgin.sql` scripts respectively during the process explained in this section. Before making the conversion, review "Determining Whether to Use PL/SQL Native Compilation".

**Note:**

- If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.
- If Database Vault is enabled, then you can run `dbmsupgnv.sql` only if the Database Vault administrator has granted you the `DV_PATCH_ADMIN` role.

During the conversion to native compilation, `TYPE` specifications are not recompiled by `dbmsupgnv.sql` to `NATIVE` because these specifications do not contain executable code.

Package specifications seldom contain executable code so the runtime benefits of compiling to `NATIVE` are not measurable. You can use the `TRUE` command-line parameter with the `dbmsupgnv.sql` script to exclude package specs from recompilation to `NATIVE`, saving time in the conversion process.

When converting to interpreted compilation, the `dbmsupgin.sql` script does not accept any parameters and does not exclude any PL/SQL units.
Note:

The following procedure describes the conversion to native compilation. If you must recompile all PL/SQL modules to interpreted compilation, make these changes in the steps.

- Skip the first step.
- Set the `PLSQL_CODE_TYPE` compilation parameter to `INTERPRETED` rather than `NATIVE`.
- Substitute `dbmsupgin.sql` for the `dbmsupgnv.sql` script.

1. Ensure that a test PL/SQL unit can be compiled. For example:
   ```sql
   ALTER PROCEDURE my_proc COMPILE PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS;
   ```

2. Shut down application services, the listener, and the database.
   - Shut down all of the Application services including the Forms Processes, Web Servers, Reports Servers, and Concurrent Manager Servers. After shutting down all of the Application services, ensure that all of the connections to the database were terminated.
   - Shut down the TNS listener of the database to ensure that no new connections are made.
   - Shut down the database in normal or immediate mode as the user `SYS`. See *Oracle Database Administrator's Guide*.

3. Set `PLSQL_CODE_TYPE` to `NATIVE` in the compilation parameter file. If the database is using a server parameter file, then set this after the database has started.

   The value of `PLSQL_CODE_TYPE` does not affect the conversion of the PL/SQL units in these steps. However, it does affect all subsequently compiled units, so explicitly set it to the desired compilation type.

4. Start up the database in upgrade mode, using the `UPGRADE` option. For information about SQL*Plus `STARTUP`, see *SQL*Plus *User's Guide and Reference*.

5. Run this code to list the invalid PL/SQL units. You can save the output of the query for future reference with the SQL `SPOOL` statement:
   ```sql
   -- To save the output of the query to a file:
   SPOOL pre_update_invalid.log
   SELECT o.OWNER, o.OBJECT_NAME, o.OBJECT_TYPE
   FROM DBA_OBJECTS o, DBA_PLSQL_OBJECT_SETTINGS s
   WHERE o.OBJECT_NAME = s.NAME AND o.STATUS='INVALID';
   -- To stop spooling the output: SPOOL OFF
   ```

   If any Oracle supplied units are invalid, try to validate them by recompiling them. For example:
   ```sql
   ALTER PACKAGE SYS.DBMS_OUTPUT COMPILE BODY REUSE SETTINGS;
   ```

   If the units cannot be validated, save the spooled log for future resolution and continue.

6. Run this query to determine how many objects are compiled `NATIVE` and `INTERPRETED` (to save the output, use the SQL `SPOOL` statement):
SELECT TYPE, PLSQL_CODE_TYPE, COUNT(*)
FROM DBA_PLSQL_OBJECT_SETTINGS
WHERE PLSQL_CODE_TYPE IS NOT NULL
GROUP BY TYPE, PLSQL_CODE_TYPE
ORDER BY TYPE, PLSQL_CODE_TYPE;

Any objects with a NULL plsql_code_type are special internal objects and can be ignored.

7. Run the $ORACLE_HOME/rdbms/admin/dbmsupgnv.sql script as the user SYS to update the plsql_code_type setting to NATIVE in the dictionary tables for all PL/SQL units. This process also invalidates the units. Use TRUE with the script to exclude package specifications; FALSE to include the package specifications.

This update must be done when the database is in UPGRADE mode. The script is guaranteed to complete successfully or rollback all the changes.

8. Shut down the database and restart in NORMAL mode.

9. Before you run the utlrp.sql script, Oracle recommends that no other sessions are connected to avoid possible problems. You can ensure this with this statement:

   ALTER SYSTEM ENABLE RESTRICTED SESSION;

10. Run the $ORACLE_HOME/rdbms/admin/utlrp.sql script as the user SYS. This script recompiles all the PL/SQL modules using a default degree of parallelism. See the comments in the script for information about setting the degree explicitly.

   If for any reason the script is abnormally terminated, rerun the utlrp.sql script to recompile any remaining invalid PL/SQL modules.

11. After the compilation completes successfully, verify that there are no invalid PL/SQL units using the query in step 5. You can spool the output of the query to the post_upgrade_invalid.log file and compare the contents with the pre_upgrade_invalid.log file, if it was created previously.

12. Re-execute the query in step 6. If recompiling with dbmsupgnv.sql, confirm that all PL/SQL units, except TYPE specifications and package specifications if excluded, are NATIVE. If recompiling with dbmsupgin.sql, confirm that all PL/SQL units are INTERPRETED.

13. Disable the restricted session mode for the database, then start the services that you previously shut down. To disable restricted session mode, use this statement:

   ALTER SYSTEM DISABLE RESTRICTED SESSION;
PL/SQL Language Elements

Summarizes the syntax and semantics of PL/SQL language elements and provides links to examples and related topics.

For instructions for reading the syntax diagrams, see Oracle Database SQL Language Reference.

Topics

- ACCESSIBLE BY Clause
- AGGREGATE Clause
- Assignment Statement
- AUTONOMOUS_TRANSACTION Pragma
- Basic LOOP Statement
- Block
- Call Specification
- CASE Statement
- CLOSE Statement
- Collection Method Invocation
- Collection Variable Declaration
- Comment
- COMPILE Clause
- Constant Declaration
- CONTINUE Statement
- COVERAGE Pragma
- Cursor FOR LOOP Statement
- Cursor Variable Declaration
- Datatype Attribute
- DEFAULT COLLATION Clause
- DELETE Statement Extension
- DEPRECATe Pragma
- DETERMINISTIC Clause
- Element Specification
- EXCEPTION_INIT Pragma
- Exception Declaration
- Exception Handler
• EXECUTE IMMEDIATE Statement
• EXIT Statement
• Explicit Cursor Declaration and Definition
• Expression
• FETCH Statement
• FOR LOOP Statement
• FORALL Statement
• Formal Parameter Declaration
• Function Declaration and Definition
• GOTO Statement
• IF Statement
• Implicit Cursor Attribute
• INLINE Pragma
• Invoker's Rights and Definer's Rights Clause
• INSERT Statement Extension
• Iterator
• Named Cursor Attribute
• NULL Statement
• OPEN Statement
• OPEN FOR Statement
• PARALLEL_ENABLE Clause
• PIPE ROW Statement
• PIPELINED Clause
• Procedure Declaration and Definition
• Qualified Expression
• RAISE Statement
• Record Variable Declaration
• RESTRICT_REFERENCES Pragma (deprecated)
• RETURN Statement
• RETURNING INTO Clause
• RESULT_CACHE Clause
• %ROWTYPE Attribute
• Scalar Variable Declaration
• SELECT INTO Statement
• SERIALLY_REUSABLE Pragma
• SHARING Clause
• SQL_MACRO Clause
14.43 Iterator

The iterator specifies an iterand and the iteration controls.

An iterator can appear in the following statements:
- FOR LOOP Statement
- Qualified Expression

Syntax

```plaintext
iterator ::= 

(iterand_decl ::= , iteration_ctl_seq ::= )

iterand_decl ::= 

iteration_ctl_seq ::= 
```

See Also:
- PL/SQL Language Fundamentals
\( \text{qual\_iteration\_ctl ::=} \)

\[
\text{REVERSE} \rightarrow \text{iteration\_control} \rightarrow \text{pred\_clause\_seq}
\]

\( (\text{iteration\_control ::=, pred\_clause\_seq ::=}) \)

\( \text{iteration\_control ::=} \)

\[
\text{stepped\_control} \rightarrow \text{single\_expression\_control} \rightarrow \text{values\_of\_control} \rightarrow \text{indices\_of\_control} \rightarrow \text{pairs\_of\_control} \rightarrow \text{cursor\_iteration\_control}
\]

\( (\text{stepped\_control ::=, single\_expression\_control ::=, values\_of\_control ::=, indices\_of\_control ::=, pairs\_of\_control ::=, cursor\_iteration\_control ::=}) \)

\( \text{pred\_clause\_seq ::=} \)

\[
\text{WHILE} \rightarrow \text{boolean\_expression} \rightarrow \text{WHEN} \rightarrow \text{boolean\_expression}
\]

\( \text{stepped\_control ::=} \)

\[
\text{lower\_bound} \rightarrow \text{upper\_bound} \rightarrow \text{BY} \rightarrow \text{step}
\]

\( \text{single\_expression\_control ::=} \)

\[
\text{REPEAT} \rightarrow \text{expr}
\]

\( \text{values\_of\_control ::=} \)

\[
\text{VALUES} \rightarrow \text{OF} \rightarrow \text{expr} \rightarrow \text{cursor\_variable} \rightarrow \text{cursor\_object} \rightarrow \text{dynamic\_sql} \rightarrow \text{sql\_statement}
\]
(dynamic_sql ::=)

indices_of_control ::= 

PAIRS OF expr cursor_variable (cursor_object dynamic_sql sql_statement)

( dynamic_sql ::= )

pairs_of_control ::= 

cursor_iteration_control ::= 

EXECUTE IMMEDIATE dynamic_sql_stmt USING IN bind_argument

( dynamic_sql ::= )

dynamic_sql ::=
Semantics

iterator
The iterator specifies an iterand and the iteration controls.

Statements outside the loop cannot reference iterator. Statements inside the loop can reference iterator, but cannot change its value. After the FOR LOOP statement runs, iterator is undefined.

iterand_decl
An iterand type can be implicitly or explicitly declared. You cannot explicitly initialize an iterand.

An iterand type is implicitly declared when no type declaration follows the iterand in the loop header. The implicit type is determined by the first iteration control.

Table 14-1  Iterand Implicit Type Defaults

<table>
<thead>
<tr>
<th>Iteration Control</th>
<th>Implicit Iterand Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>stepped control</td>
<td>PLS_INTEGER</td>
</tr>
<tr>
<td>single expression</td>
<td>PLS_INTEGER</td>
</tr>
<tr>
<td>cursor control</td>
<td>CURSOR%ROWTYPE</td>
</tr>
<tr>
<td>VALUES OF control</td>
<td>collection element type</td>
</tr>
<tr>
<td>INDICES OF control</td>
<td>collection index type</td>
</tr>
<tr>
<td>PAIRS OF control</td>
<td>The first iterand denotes the index type of collection and the second iterand denotes the element type of collection</td>
</tr>
</tbody>
</table>

pls_identifier
The iterand name for the implicitly declared variable that is local to the FOR LOOP statement.

[MUTABLE | IMMUTABLE]

The mutability property of an iterand determines whether or not it can be assigned in the loop body. If all iteration controls specified in an iterator are cursor controls, the iterand is mutable by default. Otherwise, the iterand is immutable. The default mutability property of an iterand can be changed in the iterand declaration by specifying the MUTABLE or IMMUTABLE keyword after the iterand variable. The mutability property keywords are not reserved and could be used as type names. Such usage would be ambiguous. Therefore, you must explicitly specify the mutability property of an iterand in the iterand declaration if its type is named mutable or immutable. Iterand for INDICES OF iteration control and the index iterand for PAIRS OF iteration control cannot be made mutable.

constrained_type
An iterand is explicitly declared when the iterand type is specified in the loop header. Any constraint defined for a type is considered when assigning values to the iterand. The values generated by the iteration controls must be assignment compatible with the iterand type. Usual conversion rules apply. Exceptions are raised for all constraint violations.
iteration_ctl_seq

Multiple iteration controls may be chained together by separating them with commas.

Restriction on iteration_ctl_seq:

Because two iterands are required for the pairs of iterand, pairs of iteration controls may not be mixed with other kinds of iteration controls.

qual_iteration_ctl

The qualified iteration control specifies the REVERSE option and the optional stopping and skipping predicates clauses.

[ REVERSE ]

When the optional keyword REVERSE is specified, the order of values in the sequence is reversed.

You can use this option with a collection vector value expression. In that case, specifying REVERSE generates values from LAST to FIRST rather than from FIRST to LAST.

Restrictions on REVERSE:

• You cannot use this option when a pipelined function is specified in the iteration control.

• You cannot use this option with single expression iteration control since it generates a single value and therefore the keyword does not have any sensible meaning for this control.

• You cannot use this option when the iteration control specifies a SQL statement. This creates a sequence of records returned by the query. You can specify an ORDER BY clause on the SQL statement to sort the rows in the appropriate order.

• You cannot use this option when the collection is a cursor, cursor variable, dynamic SQL, or is an expression that calls a pipelined table function.

iteration_control

An iteration control provides a sequence of values to the iterand.

pred_clause_seq

An iteration control may be modified with an optional stopping predicate clause followed by an optional skipping predicate clause. The expressions in the predicates must have a BOOLEAN type.

[ WHILE boolean_expression ]

A stopping predicate clause can cause the iteration control to be exhausted. The boolean_expression is evaluated at the beginning of each iteration of the loop. If it fails to evaluate to TRUE, the iteration control is exhausted.

[ WHEN boolean_expression ]

A skipping predicate clause can cause the loop body to be skipped for some values. The boolean_expression is evaluated. If it fails to evaluate to TRUE, the iteration control skips to the next value.
stepped_control

lower_bound .. upper_bound [ BY step ]

Without REVERSE, the value of iterand starts at lower_bound and increases by step with each iteration of the loop until it reaches upper_bound.

With REVERSE, the value of iterand starts at upper_bound and decreases by step with each iteration of the loop until it reaches lower_bound. If upper_bound is less than lower_bound, then the statements never run.

The default value for step is one if this optional BY clause is not specified.

lower_bound and upper_bound must evaluate to numbers (either numeric literals, numeric variables, or numeric expressions). If a bound does not have a numeric value, then PL/SQL raises the predefined exception VALUE_ERROR. PL/SQL evaluates lower_bound and upper_bound once, when the FOR LOOP statement is entered, and stores them as temporary PLS_INTEGER values, rounding them to the nearest integer if necessary.

If lower_bound equals upper_bound, the statements run only once.

The step value must be greater than zero.

single_expression_control

A single expression iteration control generates a single value. If REPEAT is specified, the expression will be evaluated repeatedly generating a sequence of values until a stopping clause causes the iteration control to be exhausted.

Restrictions on single_expression_control:

REVERSE is not allowed for a single expression iteration control.

values_of_control

The element type of a collection must be assignment compatible with the iterand.

indices_of_control

The index type of a collection must be assignment compatible with the iterand.

The iterand used with an INDICES OF iteration control cannot be mutable.

pairs_of_control

The PAIRS OF iteration control requires two iterands. You cannot mix the PAIRS OF iteration control with other kinds of controls. The first iterand is the index iterand and the second is the value iterand. Each iterand may be followed by an explicit type.

The element type of the collection must be assignment compatible with the value iterand. The index type of the collection must be assignment compatible with the index iterand.

The index iterand used with a PAIRS OF iteration control cannot be mutable.
**cursor_iteration_control**

Cursor iteration controls generate the sequence of records returned by an explicit or implicit cursor. The cursor definition is the controlling expression.

**Restrictions on cursor_iteration_control:**

You cannot use **REVERSE** with a cursor iteration control.

**cursor_object**

A **cursor_object** is an explicit PL/SQL cursor object.

**sql_statement**

A **sql_statement** is an implicit PL/SQL cursor object created for a SQL statement specified directly in the iteration control.

**cursor_variable**

Name of a previously declared variable of a **REF CURSOR** object.

**dynamic_sql**

**EXECUTE IMMEDIATE** `dynamic_sql_stmt [ USING [ IN ] (bind_argument [,] )+]`

You can use a dynamic query in place of an implicit cursor definition in a cursor or collection iteration control. Such a construct cannot provide a default type; if it is used as the first iteration control, an explicit type must be specified for the iterand, or for the value iterand for a pairs of control.

The optional **USING** clause is the only clause allowed with the dynamic SQL. It can only possibly have **IN** one or more bind variable, each separated by a comma.

**dynamic_sql_stmt**

String literal, string variable, or string expression that represents a SQL statement. Its type must be either **CHAR**, **VARCHAR2**, or **CLOB**.

⚠️ **Caution:**

When using dynamic SQL, beware of SQL injection, a security risk. For more information about SQL injection, see "SQL Injection".

**Examples**

- Example 5-25, "Using Dynamic SQL as an Iteration Control"
- Example 5-17, "Stepped Range Iteration Controls"
- Example 5-18, "STEP Clause in FOR LOOP Statement"
- Example 5-24, "Cursor Iteration Controls"
- Example 5-21, "VALUES OF Iteration Control"
- Example 5-22, "INDICES OF Iteration Control"
14.52 Qualified Expression

Using qualified expressions, you can declare and define a complex value in a compact form where the value is needed.

Qualified expressions appear in:
- Collection Variable Declaration
- Expression

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\[
\text{qualified_expression ::=}
\]

\[
\text{(typemark ::=, aggregate ::=)}
\]

\[
\text{typemark ::=}
\]

\[
\text{type_name ::=}
\]

\[
\text{aggregate ::=}
\]
Chapter 14
Qualified Expression

(positional_choice_list ::=, explicit_choice_list ::=)

positional_choice_list ::= 

sequence_iterator_choice ::= 

FOR iterator SEQUENCE => expr

explicit_choice_list ::= 

named_choice_list
indexed_choice_list
basic_iterator_choice
index_iterator_choice

(named_choice_list ::=, indexed_choice_list ::=, basic_iterator_choice ::=,
index_iterator_choice ::=)

named_choice_list ::= 

identifier
|
=> expr

indexed_choice_list ::= 

expr
|
=> expr

basic_iterator_choice ::= 

FOR iterator => expr

(iterator ::=)

Oracle
index_iterator_choice ::= 

```
FOR iterator INDEX expr => expr
```

(iterator ::=)

Semantics

qualified_expression

Qualified expressions for RECORD types are allowed in any context where an expression of RECORD type is allowed.

Qualified expressions for associative array types are allowed in any context where an expression of associative array type is allowed.

**typemark ( aggregate )**

Specifies explicitly the type of the aggregate (qualified items).

**typemark**

Qualified expressions use an explicit type indication to provide the type of the qualified item. This explicit indication is known as a typemark.

**type_name**

**identifier [ . ]**

Indicates the type of the qualified item.

**aggregate**

A qualified expression combines expression elements to create values of a RECORD type, or associative array type.

**positional_choice_list**

**expr [ , ]**

Positional association is allowed for qualified expressions of RECORD type.

A positional association may not follow a named association in the same construct (and vice versa).

**sequence_iterator_choice**

FOR iterator SEQUENCE => expr

The sequence iterator choice association is a positional argument and may be intermixed freely with other positional arguments. All positional arguments must precede any non-positional arguments.

**explicit_choice_list**

`named_choice_list | indexed_choice_list | basic_iterator_choice | index_iterator_choice`
Named choices must use names of fields from the qualifying structure type. Index key values must be compatible with the index type for the qualifying vector type.

\[ \text{name}_\text{ed}_\text{_choice}_\text{_list} \]

A named choice applies only to structured types

\[ \text{identifier} => \text{expr} [ , ] \]

Named association is allowed for qualified expressions of \text{RECORD} type.

\[ \text{indexed}_\text{_choice}_\text{_list} \]

An index choice applies only to vector types.

\[ \text{expr} => \text{expr} [ , ] \]

Indexed choices (key-value pairs) is allowed for qualified expressions of associative array types. Both the key and the value may be expressions.

Using \text{NULL} as an index key value is not permitted with associative array type constructs.

\[ \text{basic}_\text{_iterator}_\text{_choice} \]

\[ \text{FOR} \ \text{iterator} => \text{expr} \]

The basic iterator choice association uses the iterand as an index.

\[ \text{Restrictions:} \]

The \text{PAIRS} of iteration control may not be used with the basic iterator choice association.

\[ \text{index}_\text{_iterator}_\text{_choice} \]

\[ \text{FOR} \ \text{iterator} \ \text{INDEX} \ \text{expr} => \text{expr} \]

The index iterator choice association provides an index expression along with the value expression.

\[ \text{Examples} \]

- Example 6-11, "Assigning Values to \text{RECORD} Type Variables Using Qualified Expressions"
- Example 6-12, "Assigning Values to Associative Array Type Variables Using Qualified Expressions"
- Example 6-8, "Basic Iterator Choice Association in Qualified Expressions"
- Example 6-9, "Index Iterator Choice Association in Qualified Expressions"
- Example 6-10, "Sequence Iterator Choice Association in Qualified Expressions"
- Example 5-26, "Using Dynamic SQL As An Iteration In A Qualified Expression"

\[ \text{Related Topics} \]

- "Qualified Expressions Overview" for more information and examples
- "Expressions"
- Assigning Values to Collection Variables
14.64 SQL_MACRO Clause

The SQL_MACRO clause marks a function as a SQL macro which can be used as either a scalar expression or a table expression.

A TABLE macro is a function annotated as a SQL_MACRO and defined as a TABLE type.

A SCALAR macro is a function annotated as a SQL_MACRO and defined as a SCALAR type.

A SQL macro referenced in a view is always executed with the view owner's privileges.

The AUTHID property cannot be specified. When a SQL macro is invoked, they behave like invoker's right function. The SQL macro owner must grant inherit privileges to the invoking function.

When a macro annotated function is used in PL/SQL, it works like a regular function returning character or CLOB type with no macro expansion.

The SQL_MACRO annotation can appear in the following SQL statement:

- CREATE FUNCTION Statement

SQL Macro Usage Restrictions:

- A TABLE macro can only appear in FROM clause of a query table expression.
- A SCALAR macro cannot appear in FROM clause of a query table expression. It can appear in the select list or the WHERE clause.
- A scalar macro cannot have table arguments.
- A SQL macro cannot appear in a virtual column expression, functional index, editioning view or materialized view.
- Type methods cannot be annotated with SQL_MACRO.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\[
\text{sql_macro_clause ::=}
\]
Semantics

\textit{sql\_macro\_clause}

The \textit{sql\_macro\_clause} can appear only once in the function. To make a SQL macro function, include the \textit{sql\_macro\_clause} in the function definition. If you declare the SQL macro function before defining it, you must specify the \textit{sql\_macro\_clause} in the function declaration.

If \texttt{SCALAR} or \texttt{TABLE} is not specified, \texttt{TABLE} is the default.

\textbf{SCALAR}

Specify \texttt{SCALAR} if the macro function can be used in scalar expressions.

\textbf{TABLE (Default)}

Specify \texttt{TABLE} if the macro function can be used in table expressions.

Restrictions on \textit{sql\_macro\_clause}

The \texttt{SQL\_MACRO} annotation is disallowed with \texttt{RESULT\_CACHE}, \texttt{PARALLEL\_ENABLE}, and \texttt{PIEPLINED}. Although the \texttt{DETERMINISTIC} property cannot be specified, a SQL macro is always implicitly deterministic.

The SQL macro function must have a return type of \texttt{VARCHAR2}, \texttt{CHAR}, or \texttt{CLOB}.

Examples

\textbf{Example 14-1} \hspace{1em} \textit{Emp\_doc: Using a Scalar Macro to Convert Columns into a JSON or XML Document}

The emp\_doc SQL macro converts employee fields into a document string (JSON or XML).

The macro is implemented as a tree of nested macros with the following call graph structure.

\begin{verbatim}
emp_doc()
  ==> emp_json()
    ==> name_string()
    ==> email_string()
      ==> name_string()
    ==> date_string()
  ==> emp_xml
    ==> name_string()
    ==> email_string()
      ==> name_string()
    ==> date_string()
\end{verbatim}

The \texttt{name\_string} function converts a date in a string formatted as a four digits year, month (01-12) and day of the month (1-31).

\begin{verbatim}
CREATE FUNCTION date_string(dat DATE)
    RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
    RETURN q'
        TO_CHAR(dat, 'YYYY-MM-DD')
    ';
END;
/
\end{verbatim}
The name_string function sets the first letter of each words in the first_name and last_name in uppercase and all other letters in lowercase. It concatenates the formatted first name with a space and the formatted last name, and removes leading and trailing spaces from the resulting string.

CREATE FUNCTION name_string(first_name VARCHAR2, last_name VARCHAR2) RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
    RETURN q'\n        TRIM(INITCAP(first_name) \| \ ' \| INITCAP(last_name))
    ';
END;
/

The email_string sets the email address using the name_string function with the first_name and last_name and replacing all spaces with a period, and appending a default domain name of example.com.

CREATE FUNCTION email_string(first_name VARCHAR2, last_name VARCHAR2, host_name VARCHAR2 DEFAULT 'example.com') RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
    RETURN q'\n        REPLACE(LOWER(name_string(first_name, last_name)),' ','.') \| '@' \| host_name
    ';
END;
/

The emp_json SQL macro returns a JSON document string.

CREATE FUNCTION emp_json(first_name VARCHAR2 DEFAULT NULL, last_name VARCHAR2 DEFAULT NULL, hire_date DATE DEFAULT NULL, phone_num VARCHAR2 DEFAULT NULL) RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
    RETURN q'\n        JSON_OBJECT( 
            'name' : name_string(first_name, last_name), 
            'email' : email_string(first_name, last_name), 
            'phone' : phone_num, 
            'hire_date' : date_string(hire_date) 
            ABSENT ON NULL)
    ';
END;
/

The emp_xml SQL macro returns an XML document string.

CREATE FUNCTION emp_xml(first_name VARCHAR2 DEFAULT NULL, last_name VARCHAR2 DEFAULT NULL, hire_date DATE DEFAULT NULL, phone_num VARCHAR2 DEFAULT NULL) RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
    RETURN q'\n        XMLELEMENT("xml", 
            CASE WHEN first_name \| last_name IS NOT NULL THEN 
                XMLELEMENT("name", name_string(first_name, last_name))
The emp_doc SQL macro returns employee fields into a JSON (default) or XML document string.

```sql
CREATE FUNCTION emp_doc(first_name VARCHAR2 DEFAULT NULL,
                        last_name VARCHAR2 DEFAULT NULL,
                        hire_date DATE DEFAULT NULL,
                        phone_num VARCHAR2 DEFAULT NULL,
                        doc_type VARCHAR2 DEFAULT 'json')
RETURN VARCHAR2 SQL_MACRO(SCALAR) IS
BEGIN
    RETURN q'
        ))
    END;
/

This query shows the emp_doc SQL macro used in a scalar expression to list all employees in a JSON document string in department 30.

```sql
SELECT department_id,
       emp_doc(first_name => e.first_name, hire_date => e.hire_date) doc
FROM hr.employees e
WHERE department_id = 30
ORDER BY last_name;
```
20 <xml><name>Adams</name><email>adams@example.com</email><hire_date>1987-05-23</hire_date></xml>
30 <xml><name>Allen</name><email>allen@example.com</email><hire_date>1981-02-20</hire_date></xml>
30 <xml><name>Blake</name><email>blake@example.com</email><hire_date>1981-05-01</hire_date></xml>
10 <xml><name>Clark</name><email>clark@example.com</email><hire_date>1981-06-09</hire_date></xml>
20 <xml><name>Ford</name><email>ford@example.com</email><hire_date>1981-12-03</hire_date></xml>
...
30 <xml><name>Ward</name><email>ward@example.com</email><hire_date>1981-02-22</hire_date></xml>

VARIABLE surname VARCHAR2(100)
EXEC :surname := 'ellison'
WITH e AS (SELECT emp.*, :surname lname FROM emp WHERE deptno IN (10,20))
SELECT deptno,
    emp_doc(first_name => ename, last_name => lname, hire_date => hiredate) doc
FROM e
ORDER BY ename;

Result:

10 {"name":"Clark Ellison","email":"clark.ellison@example.com","hire_date":"1981-06-09"}
20 {"name":"Ford Ellison","email":"ford.ellison@example.com","hire_date":"1981-12-03"}
20 {"name":"Jones Ellison","email":"jones.ellison@example.com","hire_date":"1981-04-02"}
10 {"name":"King Ellison","email":"king.ellison@example.com","hire_date":"1981-11-17"}
10 {"name":"Miller Ellison","email":"miller.ellison@example.com","hire_date":"1982-01-23"}
20 {"name":"Scott Ellison","email":"scott.ellison@example.com","hire_date":"1987-04-19"}
20 {"name":"Smith Ellison","email":"smith.ellison@example.com","hire_date":"1980-12-17"}

Example 14-2  Env: Using a Scalar Macro in a Scalar Expression

The env SQL macro provides a wrapper for the value of the parameter associated with the context namespace USERENV which describes the current session.

CREATE PACKAGE env AS
    FUNCTION current_user RETURN VARCHAR2 SQL_MACRO(SCALAR);
    FUNCTION current_edition_name RETURN VARCHAR2 SQL_MACRO(SCALAR);
    FUNCTION module RETURN VARCHAR2 SQL_MACRO(SCALAR);
    FUNCTION action RETURN VARCHAR2 SQL_MACRO(SCALAR);
END;
/
CREATE PACKAGE BODY env AS
    FUNCTION current_user RETURN VARCHAR2 SQL_MACRO(SCALAR) IS 
        BEGIN 
            RETURN q'{SYS_CONTEXT('userenv','SESSION_USER')}';
        END;
    FUNCTION current_edition_name RETURN VARCHAR2 SQL_MACRO(SCALAR) IS 
        BEGIN 
            RETURN q'{SYS_CONTEXT('userenv','CURRENT_EDITION_NAME')}';
        END;
FUNCTION module RETURN VARCHAR2 SQL_MACRO(SCALAR) IS BEGIN
    RETURN q'(SYS_CONTEXT('userenv','MODULE'))';
END;

FUNCTION action RETURN VARCHAR2 SQL_MACRO(SCALAR) IS BEGIN
    RETURN q'(SYS_CONTEXT('userenv','ACTION'))';
END;

Select the current user info.
SELECT env.current_user, env.module, env.action FROM DUAL;

Result:
SCOTT SQL*PLUS

Example 14-3  Budget : Using a Table Macro in a Table Expression

This example shows the SQL macro named budget used in a table expression to return the total salary in each department for employees for a given job title.

CREATE FUNCTION budget(job VARCHAR2) RETURN VARCHAR2 SQL_MACRO IS BEGIN
    RETURN q'(SELECT deptno, SUM(sal) budget
    FROM scott.emp
    WHERE job = budget.job
    GROUP BY deptno)';
END;

This query shows the SQL macro budget used in a table expression.
SELECT * FROM budget('MANAGER');

Result:

DEPTNO     BUDGET
---------- ----------
20       2975
30       2850
10       2450

Example 14-4  Take: Using a Table Macro with a Polymorphic View

This example creates a table macro named take which returns the first n rows from table t.

CREATE FUNCTION take (n NUMBER, t DBMS_TF.table_t)
RETURN VARCHAR2 SQL_MACRO IS BEGIN
    RETURN 'SELECT * FROM t FETCH FIRST take.n ROWS ONLY';
END;

The query returns the first two rows from table dept.
SELECT * FROM take(2, dept);
**Example 14-5  Range : Using a Table Macro in a Table Expression**

This example creates a SQL macro that generates an arithmetic progression of rows in the range \([\text{first}, \text{stop}]\). The first row start with the value \(\text{first}\), and each subsequent row's value will be \(\text{step}\) more than the previous row's value.

The following combination of arguments will produce zero rows:

- \(\text{step} < 0\) and \(\text{first} < \text{stop}\)
- \(\text{step} = 0\)
- \(\text{step} > 0\) and \(\text{first} > \text{stop}\)

```sql
CREATE PACKAGE gen IS
FUNCTION range(stop NUMBER) RETURN VARCHAR2 SQL_MACRO(TABLE);
FUNCTION range(first NUMBER DEFAULT 0, stop NUMBER, step NUMBER DEFAULT 1) RETURN VARCHAR2 SQL_MACRO(TABLE);
FUNCTION tab(tab TABLE, replication_factor NATURAL) RETURN TABLE PIPELINED ROW POLYMORPHIC USING gen;
FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T, replication_factor NATURAL) RETURN DBMS_TF.DESCRIBE_T;
PROCEDURE fetch_rows(replication_factor NATURAL);
END gen;
/
CREATE PACKAGE BODY gen IS
FUNCTION range(stop NUMBER) RETURN VARCHAR2 SQL_MACRO(TABLE);
```
The `gen.get_range` SQL macro is used in table expressions.

This query returns a sequence of 5 rows starting at zero.

```
SELECT * FROM gen.range(5);
```

Result:

```
0
1
2
3
4
```

This query returns a sequence starting at 5, stopping at 10 (not included).

```
SELECT * FROM gen.range(5, 10);
```

Result:

```
5
6
7
8
9
```

This query returns a sequence starting at 0, stopping at 1, by increment of 0.1.

```
SELECT * FROM gen.range(0, 1, step => 0.1);
```

Result:
This query returns a sequence starting at 5, stopping at -6 (not included) by decrement of 2.

```sql
SELECT * FROM gen.range(+5,-6,-2);
```

Result:

```
5
3
1
-1
-3
-5
```

### Related Topics

- Overview of Polymorphic Table Functions
- Oracle Database PL/SQL Packages and Types Reference for more information about how to specify the PTF implementation package and use the DBMS_TF utilities

## 14.1 ACCESSIBLE BY Clause

The **ACCESSIBLE BY** clause restricts access to units and subprograms.

The **ACCESSIBLE BY** clause limits access to a unit or subprogram by other units. The **accessor list**, also known as the white list, explicitly lists those units which may have access. The accessor list can be defined on individual subprograms in a package. This list is checked in addition to the accessor list defined on the package itself (if any). This list may only restrict access to the subprogram – it cannot expand access. This code management feature is useful to prevent inadvertent use of internal subprograms. For example, it may not be convenient or feasible to reorganize a package into two packages: one for a small number of procedures requiring restricted access, and another one for the remaining units requiring public access.

The **ACCESSIBLE BY** clause may appear in the declarations of object types, object type bodies, packages, and subprograms.

The **ACCESSIBLE BY** clause can appear in the following SQL statements:

- **ALTER TYPE Statement**
- **CREATE FUNCTION Statement**
- **CREATE PROCEDURE Statement**
CREATE PACKAGE Statement
CREATE TYPE Statement
CREATE TYPE BODY Statement

Topics
- Syntax
- Semantics
- Usage Notes
- Examples
- Related Topics

Syntax

\[
\text{accessible_by_clause ::=}
\]

\[
\begin{align*}
\text{ACCESSIBLE} & \quad \text{BY} \\
\quad & \quad \text{accessor}
\end{align*}
\]

\[
\text{accessor ::=}
\]

\[
\begin{align*}
\text{unit_kind} & \quad \text{schema} \\
\quad & \quad . \\
\quad & \quad \text{unit_name}
\end{align*}
\]

\[
\text{unit_kind ::=}
\]

\[
\begin{align*}
\text{FUNCTION} & \\
\text{PROCEDURE} & \\
\text{PACKAGE} & \\
\text{TRIGGER} & \\
\text{TYPE}
\end{align*}
\]

Semantics

\[
\text{accessible_by_clause}
\]

\[
\text{accessor}
\]

\[
\text{[schema.]unit_name}
\]

Specifies a stored PL/SQL unit that can invoke the entity.

Each accessor specifies another PL/SQL entity that may access the entity which includes the ACCESSIBLE BY clause.
When an `ACCESSIBLE BY` clause appears, only entities named in the clause may access the entity in which the clause appears.

An accessor may appear more than once in the `ACCESSIBLE BY` clause.

The `ACCESSIBLE BY` clause can appear only once in the unit declaration.

An entity named in an accessor is not required to exist.

When an entity with an `ACCESSIBLE BY` clause is invoked, it imposes an additional access check after all other checks have been performed. These checks are:

- The invoked unit must include an accessor with the same `unit_name` and `unit_kind` as the invoking unit.
- If the accessor includes a `schema`, the invoking unit must be in that `schema`.
- If the accessor does not include a `schema`, the invoker must be from the same schema as the invoked entity.

`unit_kind` specifies if the unit is a `FUNCTION`, `PACKAGE`, `PROCEDURE`, `TRIGGER`, or `TYPE`.

Usage Notes

The `unit_kind` is optional, but it is recommended to specify it to avoid ambiguity when units have the same name. For example, it is possible to define a trigger with the same name as a function.

The `ACCESSIBLE BY` clause allows access only when the call is direct. The check will fail if the access is through static SQL, `DBMS_SQL`, or dynamic SQL.

Any call to the initialization procedure of a package specification or package body will be checked against the accessor list of the package specification.

A unit can always access itself. An item in a unit can reference another item in the same unit.

RPC calls to a protected subprogram will always fail, since there is no context available to check the validity of the call, at either compile-time or run-time.

Calls to a protected subprogram from a conditional compilation directive will fail.

Examples

**Example 14-6  Restricting Access to Top-Level Procedures in the Same Schema**

This example shows that the top-level procedure `top_protected_proc` can only be called by procedure `top_trusted_proc` in the current schema. The user cannot call `top_protected_proc` directly.
Example 14-7  Restricting Access to a Unit Name of Any Kind

This example shows that if the PL/SQL unit_kind is not specified in the ACCESSIBLE BY clause, then a call from any unit kind is allowed if the unit name matches. There is no compilation error if the unit_kind specified in the ACCESSIBLE BY clause does not match any existing objects. It is possible to define a trigger with the same name as a function. It is recommended to specify the unit_kind.
FUNCTION top_protected_f RETURN NUMBER
ACCESSIBLE BY (TRIGGER top_trusted_f ) AS
BEGIN
    RETURN 0.5;
END top_protected_f;

FUNCTION top_trusted_f RETURN NUMBER AUTHID DEFINER IS
    FUNCTION g RETURN NUMBER DETERMINISTIC IS
    BEGIN
        RETURN 0.5;
    END g;
BEGIN
    protected_proc2;
    RETURN g() - DBMS_RANDOM.VALUE();
END top_trusted_f;

SELECT top_trusted_f FROM DUAL;
    .381773176
1 row selected.

Executed protected_proc2.

Example 14-8  Restricting Access to a Stored Procedure

This example shows a package procedure that can only be called by top_trusted_proc
procedure. The ACCESSIBLE BY clause of a subprogram specification and body must
match. A compilation error is raised if a call is made to an existing procedure with an
ACCESSIBLE BY clause that does not include this procedure in its accessor list.

Live SQL:

You can view and run this example on Oracle Live SQL at Restricting Access
to a Stored Procedure

CREATE OR REPLACE PACKAGE protected_pkg
AS
    PROCEDURE public_proc;
    PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc);
END;

CREATE OR REPLACE PACKAGE BODY protected_pkg
AS
    PROCEDURE public_proc AS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Executed protected_pkg.public_proc');
    END;
    PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc) AS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Executed protected_pkg.private_proc');
    END;
CREATE OR REPLACE PROCEDURE top_trusted_proc
AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls protected_pkg.private_proc ');
    protected_pkg.private_proc;
  END;

Procedure created.

EXEC top_trusted_proc;
top_trusted_proc calls protected_pkg.private_proc
Executed protected_pkg.private_proc

EXEC protected_pkg.private_proc
PLS-00904: insufficient privilege to access object PRIVATE_PROC

Related Topics
In this chapter:
  • Function Declaration and Definition
  • Procedure Declaration and Definition
In other chapters:
  • Nested, Package, and Standalone Subprograms
  • Subprogram Properties
  • Package Writing Guidelines

14.2 AGGREGATE Clause

Identifies the function as an aggregate function, or one that evaluates a group of rows and returns a single row.

You can specify aggregate functions in the select list, HAVING clause, and ORDER BY clause.

When you specify a user-defined aggregate function in a query, you can treat it as an analytic function (one that operates on a query result set). To do so, use the OVER analytic_clause syntax available for SQL analytic functions.

The AGGREGATE clause can appear in the CREATE FUNCTION Statement.

Topics
  • Syntax
  • Semantics
  • Examples
  • Related Topics
Syntax

aggregate_clause ::= 

AGGREGATE USING

Semantics

aggregate_clause

AGGREGATE USING

Specify the name of the implementation type of the function.

[schema.] implementation_type

The implementation type must be an ADT containing the implementation of the 
ODCIAggregate subprograms. If you do not specify schema, then the database 
assumes that the implementation type is in your schema.

Restriction on AGGREGATE USING

You cannot specify the aggregate_clause for a nested function.

If you specify this clause, then you can specify only one input argument for the 
function.

Examples

• Example 13-33, "Pipelined Table Function as Aggregate Function"

Related Topics

In this chapter:

• Function Declaration and Definition

In other books:

• Oracle Database SQL Language Reference for syntax and semantics of analytic 
functions
• Oracle Database Data Cartridge Developer's Guide for more information about 
user-defined aggregate functions
• Oracle Database Data Cartridge Developer's Guide for information about ODCI 
subprograms

14.3 Assignment Statement

The assignment statement sets the value of a data item to a valid value.

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

assignment_statement ::= 

( expression ::= )

assignment_statement_target ::= 

placeholder ::= 

Semantics

assignment_statement

eexpression

Expression whose value is to be assigned to assignment_statement_target.

eexpression and assignment_statement_target must have compatible data types.
**Note:**
Collections with elements of the same type might not have the same data type. For the syntax of collection type definitions, see "Collection Variable Declaration".

**assignment_statement_target**
Data item to which the value of *expression* is to be assigned.

**collection_variable**
Name of a collection variable.

**index**
Index of an element of *collection_variable*. Without *index*, the entire collection variable is the assignment statement target.

*index* must be a numeric expression whose data type either is *PLS_INTEGER* or can be implicitly converted to *PLS_INTEGER* (for information about the latter, see "Predefined *PLS_INTEGER* Subtypes").

**cursor_variable**
Name of a cursor variable.

`:host_cursor_variable`
Name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and *host_cursor_variable*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

**object**
Name of an instance of an abstract data type (ADT).

**attribute**
Name of an attribute of *object*. Without *attribute*, the entire ADT is the assignment statement target.

**out_parameter**
Name of a formal *OUT* or *IN OUT* parameter of the subprogram in which the assignment statement appears.

**record_variable**
Name of a record variable.

**field**
Name of a field of *record_variable*. Without *field*, the entire record variable is the assignment statement target.

**scalar_variable**
Name of a PL/SQL scalar variable.

placeholder

:host_variable

Name of a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and host_variable.

:indicator_variable

Name of an indicator variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. (An indicator variable indicates the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, an indicator variable can detect null or truncated value in an output host variable.) Do not put space between host_variable and the colon (:) or between the colon and indicator_variable. This is correct:

:host_variable:indicator_variable

Examples

• Example 3-24, "Assigning Values to Variables with Assignment Statement"
• Example 3-27, "Assigning Value to BOOLEAN Variable"
• Example 6-13, "Data Type Compatibility for Collection Assignment"

Related Topics

In this chapter:

• "Expression"
• "FETCH Statement"
• "SELECT INTO Statement"

In other chapters:

• "Assigning Values to Variables"
• "Assigning Values to Collection Variables"
• "Assigning Values to Record Variables"

### 14.4 AUTONOMOUS_TRANSACTION Pragma

The AUTONOMOUS_TRANSACTION pragma marks a routine as autonomous; that is, independent of the main transaction.

In this context, a routine is one of these:

• Schema-level (not nested) anonymous PL/SQL block
• Standalone, package, or nested subprogram
• Method of an ADT
• Noncompound trigger
Topics
- Syntax
- Examples
- Related Topics

Syntax

\textit{autonomous\_trans\_pragma} ::= \\
\hspace{1cm}\text{PRAGMA}\text{ AUTONOMOUS\_TRANSACTION \;}

Examples
- Example 7-43, "Declaring Autonomous Function in Package"
- Example 7-44, "Declaring Autonomous Standalone Procedure"
- Example 7-45, "Declaring Autonomous PL/SQL Block"
- Example 7-46, "Autonomous Trigger Logs INSERT Statements"
- Example 7-47, "Autonomous Trigger Uses Native Dynamic SQL for DDL"
- Example 7-48, "Invoking Autonomous Function"

Related Topics
- Pragmas
- Autonomous Transactions

14.5 Basic LOOP Statement

With each iteration of the \textit{basic\_loop\_statement}, its statements run and control returns to the top of the loop. The \texttt{LOOP} statement ends when a statement inside the loop transfers control outside the loop or raises an exception.

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\textit{basic\_loop\_statement} ::= \\
\hspace{1cm}\texttt{LOOP} \texttt{statement} \texttt{END LOOP} \texttt{label} \;
(\texttt{statement ::=})

Semantics

\textit{basic\_loop\_statement}

\texttt{statement}

To prevent an infinite loop, at least one statement must transfer control outside the loop. The statements that can transfer control outside the loop are:

- "CONTINUE Statement" (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement"
- "GOTO Statement"
- "RAISE Statement"

\textit{label}

A label that identifies \textit{basic\_loop\_statement} (see "\texttt{statement ::=}" and "\texttt{label}"). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label in the END LOOP statement matches a label at the beginning of the same LOOP statement (the compiler does not check).

Examples

\textbf{Example 14-9} Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements

In this example, one basic LOOP statement is nested inside the other, and both have labels. The inner loop has two EXIT WHEN statements; one that exits the inner loop and one that exits the outer loop.

\begin{verbatim}
DECLARE
  s  PLS_INTEGER := 0;
  i  PLS_INTEGER := 0;
  j  PLS_INTEGER;
BEGIN
  <<outer_loop>>
  LOOP
    i := i + 1;
    j := 0;
  <<inner_loop>>
  LOOP
    j := j + 1;
    s := s + i * j; -- Sum several products
    EXIT inner_loop WHEN (j > 5);
    EXIT outer_loop WHEN ((i * j) > 15);
  END LOOP inner_loop;
  END LOOP outer_loop;
DBMS_OUTPUT.PUT_LINE ('The sum of products equals: ' || TO_CHAR(s));
END;
/
\end{verbatim}

Result:
The sum of products equals: 166

**Example 14-10  Nested, Unlabeled Basic LOOP Statements with EXIT WHEN Statements**

An **EXIT WHEN** statement in an inner loop can transfer control to an outer loop only if the outer loop is labeled.

In this example, the outer loop is not labeled; therefore, the inner loop cannot transfer control to it.

```plsql
DECLARE
    i PLS_INTEGER := 0;
    j PLS_INTEGER := 0;
BEGIN
    LOOP
        i := i + 1;
        DBMS_OUTPUT.PUT_LINE ('i = ' || i);

        LOOP
            j := j + 1;
            DBMS_OUTPUT.PUT_LINE ('j = ' || j);
            EXIT WHEN (j > 3);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE ('Exited inner loop');
        EXIT WHEN (i > 2);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('Exited outer loop');
END;
/
```

**Result:**

```
i = 1
j = 1
j = 2
j = 3
j = 4
Exited inner loop
i = 2
j = 5
Exited inner loop
i = 3
j = 6
Exited inner loop
Exited outer loop
```

PL/SQL procedure successfully completed.

**Related Topics**

- "Cursor FOR LOOP Statement"
- "FOR LOOP Statement"
- "WHILE LOOP Statement"
- "Basic LOOP Statement"
14.6 Block

The block, which groups related declarations and statements, is the basic unit of a PL/SQL source program.

It has an optional declarative part, a required executable part, and an optional exception-handling part. Declarations are local to the block and cease to exist when the block completes execution. Blocks can be nested.

An anonymous block is an executable statement.

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
plsql_block ::= \\
\langle\langle label \rangle\rangle \text{DECLARE} \ declare_section \ \langle body \rangle
```

(body ::=)

```
declare_section ::= \\
\langle item_list_1 \rangle \langle item_list_2 \rangle
```

(item_list_2 ::=)

```
item_list_1 ::= \\
type_definition \ cursor_declaration \ item_declaration \ function_declaration \ procedure_declaration
```
(cursor_declaration ::=, function_declaration ::=, item_declaration ::=, procedure_declaration ::=, type_definition ::=)

item_list_2 ::= 

(type_definition ::=, cursor_definition ::=, function_definition ::=, function_declaration ::=, procedure_definition ::=, procedure_declaration ::=)

diagram of item_list_2

(type_definition ::=, collection_type_definition ::=, ref_cursor_type_definition ::=, record_type_definition ::=, subtype_definition ::=)

diagram of type_definition

(subtype_definition ::=, constraint ::=, precision ::=, scale ::=, RANGE low_value .. high_value, NOT NULL, base_type)

diagram of constraint

Chapter 14
Block
item_declaration ::= 

(collection_variable_decl, constant_declaration, cursor_declaration, 
cursor_variable_declaration, exception_declaration, 
record_variable_declaration, variable_declaration).

body ::= 

(exception_handler ::=)

(collection_variable_decl ::=, constant_declaration ::=, cursor_declaration ::=, 
cursor_variable_declaration ::=, exception_declaration ::=, 
record_variable_declaration ::=, variable_declaration ::=)

Chapter 14
Block

14-37
statement ::= 

```
<< label ... plsql_block
procedure_call
raise_statement
return_statement
select_into_statement
sql_statement
while_loop_statement

(plsql_block ::=, procedure_call ::=, sql_statement ::=)
```

procedure_call ::= 

```
procedure
(parameter
'
)
;```
\textit{sql\_statement} ::= 

\begin{itemize}
  \item \textit{commit\_statement}
  \item \textit{collection\_method\_call}
  \item \textit{delete\_statement}
  \item \textit{insert\_statement}
  \item \textit{lock\_table\_statement}
  \item \textit{merge\_statement}
  \item \textit{rollback\_statement}
  \item \textit{savepoint\_statement}
  \item \textit{set\_transaction\_statement}
  \item \textit{update\_statement}
\end{itemize}

Semantics

\textit{plsql\_block}

\textbf{label}

Undeclared identifier, unique for the block.

\textbf{DECLARE}

Starts the declarative part of the block.

\textit{declare\_section}

Contains local declarations, which exist only in the block and its sub-blocks and are not visible to enclosing blocks.

\textbf{Restrictions on \textit{declare\_section}}

\begin{itemize}
  \item A \textit{declare\_section} in \textit{create\_package}, \textit{create\_package\_body}, or \textit{compound\_trigger\_block} cannot include \texttt{PRAGMA AUTONOMOUS\_TRANSACTION}.
  \item A \textit{declare\_section} in \textit{trigger\_body} or \textit{tps\_body} cannot declare variables of the data type \texttt{LONG} or \texttt{LONG\_RAW}.
\end{itemize}

\textbf{See Also:}

\begin{itemize}
  \item "\textit{CREATE PACKAGE Statement}" for more information about \textit{create\_package}
  \item "\textit{CREATE PACKAGE BODY Statement}" for more information about \textit{create\_package\_body}
  \item "\textit{CREATE TRIGGER Statement}" for more information about \textit{compound\_trigger\_block}, \textit{trigger\_body}, and \textit{tps\_body}
\end{itemize}
**subtype_definition**

Static expressions can be used in subtype declarations. See Static Expressions for more information.

**subtype**

Name of the user-defined subtype that you are defining.

**base_type**

Base type of the subtype that you are defining. base_type can be any scalar or user-defined PL/SQL datatype specifier such as CHAR, DATE, or RECORD.

**CHARACTER SET character_set**

Specifies the character set for a subtype of a character data type.

Restriction on CHARACTER SET character_set:

Do not specify this clause if base_type is not a character data type.

**NOT NULL**

Imposes the NOT NULL constraint on data items declared with this subtype. For information about this constraint, see "NOT NULL Constraint".

**constraint**

Specifies a constraint for a subtype of a numeric data type.

Restriction on constraint:

Do not specify constraint if base_type is not a numeric data type.

**precision**

Specifies the precision for a constrained subtype of a numeric data type.

Restriction on precision:

Do not specify precision if base_type cannot specify precision.

**scale**

Specifies the scale for a constrained subtype of a numeric data type.

Restriction on scale:

Do not specify scale if base_type cannot specify scale.

**RANGE low_value .. high_value**

Specifies the range for a constrained subtype of a numeric data type. The low_value and high_value must be numeric literals.

Restriction on RANGE high_value .. low_value:

Specify this clause only if base_type is PLS_INTEGER or a subtype of PLS_INTEGER (either predefined or user-defined). (For a summary of the predefined subtypes of PLS_INTEGER, see Table 4-3. For information about user-defined subtypes with ranges, see "Constrained Subtypes".)
body

BEGIN

Starts the executable part of the block, which contains executable statements.

EXCEPTION

Starts the exception-handling part of the block. When PL/SQL raises an exception, normal execution of the block stops and control transfers to the appropriate exception_handler. After the exception handler completes, execution resumes with the statement following the block. For more information about exception-handling, see PL/SQL Error Handling.

exception_handler

See "Exception Handler".

END

Ends the block.

name

The name of the block to which END applies—a label, function name, procedure name, or package name.

statement

label

Undeclared identifier, unique for the statement.

assignment_statement

See "Assignment Statement".

basic_loop_statement

See "Basic LOOP Statement".

case_statement

See "CASE Statement".

close_statement

See "CLOSE Statement".

collection_method_call

Invocation of one of these collection methods, which are procedures:

- DELETE
- EXTEND
- TRIM

For syntax, see "Collection Method Invocation".

continue_statement
See "CONTINUE Statement".

`cursor_for_loop_statement`
See "Cursor FOR LOOP Statement".

`executeImmediate_statement`
See "EXECUTE IMMEDIATE Statement".

`exit_statement`
See "EXIT Statement".

`fetch_statement`
See "FETCH Statement".

`for_loop_statement`
See "FOR LOOP Statement".

`forall_statement`
See "FORALL Statement".

`goto_statement`
See "GOTO Statement".

`if_statement`
See "IF Statement".

`null_statement`
See "NULL Statement".

`open_statement`
See "OPEN Statement".

`open_for_statement`
See "OPEN FOR Statement".

`pipe_row_statement`
See "PIPE ROW Statement".

**Restriction on pipe_row_statement**
This statement can appear only in the body of a pipelined table function; otherwise, PL/SQL raises an exception.

`raise_statement`
See "RAISE Statement".

`return_statement`
See "RETURN Statement".

`select_into_statement`
See "SELECT INTO Statement".
while_loop_statement

See "WHILE LOOP Statement".

procedure_call

procedure

Name of the procedure that you are invoking.

parameter [, parameter ]...

List of actual parameters for the procedure that you are invoking. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter. The mode of the formal parameter determines what the actual parameter can be:

<table>
<thead>
<tr>
<th>Formal Parameter Mode</th>
<th>Actual Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Constant, initialized variable, literal, or expression</td>
</tr>
<tr>
<td>OUT</td>
<td>Variable whose data type is not defined as NOT NULL</td>
</tr>
<tr>
<td>IN OUT</td>
<td>Variable (typically, it is a string buffer or numeric accumulator)</td>
</tr>
</tbody>
</table>

If the procedure specifies a default value for a parameter, you can omit that parameter from the parameter list. If the procedure has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

See Also:

"Positional, Named, and Mixed Notation for Actual Parameters"

sql_statement

commit_statement

SQL COMMIT statement. For syntax, see Oracle Database SQL Language Reference.

delete_statement

SQL DELETE statement. For syntax, see Oracle Database SQL Language Reference. See also "DELETE Statement Extension".

insert_statement

SQL INSERT statement. For syntax, see Oracle Database SQL Language Reference. See also "INSERT Statement Extension".

lock_table_statement

SQL LOCK TABLE statement. For syntax, see Oracle Database SQL Language Reference.

merge_statement

SQL MERGE statement. For syntax, see Oracle Database SQL Language Reference.
rollback_statement

SQL ROLLBACK statement. For syntax, see Oracle Database SQL Language Reference.

savepoint_statement

SQL SAVEPOINT statement. For syntax, see Oracle Database SQL Language Reference.

set_transaction_statement

SQL SET TRANSACTION statement. For syntax, see Oracle Database SQL Language Reference.

update_statement

SQL UPDATE statement. For syntax, see Oracle Database SQL Language Reference. See also "UPDATE Statement Extensions".

Examples

- Example 2-1, "PL/SQL Block Structure"
- Example 3-23, "Block with Multiple and Duplicate Labels"

Related Topics

- "Comment"
- "Blocks"
- "Identifiers"
- "Pragmas"
- "PL/SQL Data Types"
- "User-Defined PL/SQL Subtypes"

14.7 Call Specification

A call specification declares a Java method or a C language subprogram so that it can be invoked from PL/SQL. You can also use the SQL CALL statement to invoke such a method or subprogram. The call specification tells the database which Java method, or which named subprogram in which shared library, to invoke when an invocation is made. It also tells the database what type conversions to make for the arguments and return value.

A call specification can appear in the following SQL statements:

- ALTER TYPE Statement
- CREATE FUNCTION Statement
- CREATE PROCEDURE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement
Topics

• Syntax
• Semantics
• Examples
• Related Topics

Prerequisites

To invoke a call specification, you may need additional privileges, for example, `EXECUTE` privileges on a C library for a C call specification.

Syntax

```
call_spec ::= java_declaration | c_declaration
```

```
java_declaration ::= LANGUAGE JAVA NAME string
```

```
c_declaration ::= LANGUAGE C EXTERNAL NAME name LIBRARY lib_name LIBRARY lib_name NAME name AGENT IN (argument,) WITH CONTEXT PARAMETERS (external_parameter,) external_parameter
```

```
external_parameter ::= CONTEXT SELF TDO property parameter_name RETURN property BY REFERENCE external_datatype
```
property ::= 

Semantics

call_spec
Maps a C procedure or Java method name, parameter types, and return type to their SQL counterparts.

Call specifications can appear in PL/SQL standalone subprograms, package specifications and bodies, and type specifications and bodies. They cannot appear inside PL/SQL blocks.

java_declaration

string
Identifies the Java implementation of the method.

c_declaration

LIBRARY lib_name
Identifies a library created by the "CREATE LIBRARY Statement".

EXTERNAL
Depreciated way of declaring a C subprogram, supported only for backward compatibility. Use EXTERNAL in a C call specification if it contains defaulted arguments or constrained PL/SQL types, otherwise use the LANGUAGE C syntax.

Examples

Example 14-11  External Function Example

The hypothetical following statement creates a PL/SQL standalone function get_val that registers the C subprogram c_get_val as an external function. (The parameters have been omitted from this example.)

CREATE FUNCTION get_val
( x_val IN NUMBER,
  y_val IN NUMBER,
  image IN LONG RAW )
RETURN BINARY_INTEGER AS LANGUAGE C
14.8 **CASE Statement**

The **CASE** statement chooses from a sequence of conditions and runs a corresponding statement.

The **simple** **CASE** statement evaluates a single expression and compares it to several potential values.

The **searched** **CASE** statement evaluates multiple Boolean expressions and chooses the first one whose value is **TRUE**.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```
simple_case_statement ::=  
CASE selector WHEN selector_value THEN statement ;
ELSE statement ;
END CASE
label
```

In this chapter:

- Function Declaration and Definition
- Procedure Declaration and Definition

In other chapters:

- CREATE LIBRARY Statement
- External Subprograms

In other books:

- *Oracle Database SQL Language Reference* for information about the **CALL** statement
- *Oracle Database Development Guide* for information about restrictions on user-defined functions that are called from SQL statements
- *Oracle Database Java Developer's Guide* to learn how to write Java call specifications
- *Oracle Database Development Guide* to learn how to write C call specifications
searched_case_statement ::= 

CASE \rightarrow \begin{array}{c}
| WHEN \rightarrow boolean_expression | \rightarrow THEN | \rightarrow statement \\
| END \rightarrow CASE \rightarrow \rightarrow label |
\end{array} 

\begin{array}{c}
| ELSE \rightarrow statement |
\end{array} 

Semantics

simple_case_statement

selector

Expression whose value is evaluated once and used to select one of several alternatives. selector can have any PL/SQL data type except BLOB, BFILE, or a user-defined type.

WHEN selector_value THEN statement

selector_value can be an expression of any PL/SQL type except BLOB, BFILE, or a user-defined type.

The selector_values are evaluated sequentially. If the value of a selector_value equals the value of selector, then the statement associated with that selector_value runs, and the CASE statement ends. Subsequent selector_values are not evaluated.

Caution:

A statement can modify the database and invoke nondeterministic functions. There is no fall-through mechanism, as there is in the C switch statement.

ELSE statement [statement ]...

The statements run if and only if no selector_value has the same value as selector.

Without the ELSE clause, if no selector_value has the same value as selector, the system raises the predefined exception CASE_NOT_FOUND.

label

A label that identifies the statement (see "statement ::=" and "label").

searched_case_statement

WHEN boolean_expression THEN statement

The boolean_expressions are evaluated sequentially. If the value of a boolean_expression is TRUE, the statement associated with that boolean_expression
runs, and the **CASE** statement ends. Subsequent *boolean_expression* are not evaluated.

**Caution:**

A **statement** can modify the database and invoke nondeterministic functions. There is no fall-through mechanism, as there is in the C **switch** statement.

**ELSE statement [**statement** ]...**

The **statement**s run if and only if no *boolean_expression* has the value TRUE.

Without the **ELSE** clause, if no *boolean_expression* has the value TRUE, the system raises the predefined exception **CASE_NOT_FOUND**.

**label**

A label that identifies the statement (see "**statement ::=**" and "**label**").

**Examples**

- **Example 4-2**, "Printing BOOLEAN Values"
- **Example 5-6**, "Simple CASE Statement"
- **Example 5-7**, "Searched CASE Statement"

**Related Topics**

In this chapter:

- "**IF Statement**"

In other chapters:

- "**CASE Expressions**"
- "**Conditional Selection Statements**"
- "**Simple CASE Statement**"
- "**Searched CASE Statement**"

**See Also:**

- *Oracle Database SQL Language Reference* for information about the **NULLIF** function
- *Oracle Database SQL Language Reference* for information about the **COALESCE** function
14.9 CLOSE Statement

The `CLOSE` statement closes a named cursor, freeing its resources for reuse.

After closing an explicit cursor, you can reopen it with the `OPEN` statement. You must close an explicit cursor before reopening it.

After closing a cursor variable, you can reopen it with the `OPEN FOR` statement. You need not close a cursor variable before reopening it.

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
close_statement ::= CLOSE cursor cursor_variable host_cursor_variable;
```

Semantics

```
close_statement

cursor
Name of an open explicit cursor.

cursor_variable
Name of an open cursor variable.

:host_cursor_variable
Name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and `host_cursor_variable`.
```

Examples

- Example 7-6, “FETCH Statements Inside LOOP Statements”

Related Topics

In this chapter:
14.10 Collection Method Invocation

A collection method is a PL/SQL subprogram that either returns information about a collection or operates on a collection.

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
collection_method_call ::= collection .
     COUNT
     DELETE ( index , index )
     EXISTS ( index )
     EXTEND ( number , index )
     FIRST
     LAST
     LIMIT
     NEXT ( index )
     PRIOR ( index )
     TRIM ( number )
```
Semantics

\textit{collection\_method\_call}\n
\textit{collection}\n
Name of the collection whose method you are invoking.

**COUNT**

Function that returns the number of elements in the collection, explained in "COUNT Collection Method".

**DELETE**

Procedure that deletes elements from the collection, explained in "DELETE Collection Method".

**Restriction on DELETE**

If \textit{collection} is a varray, you cannot specify indexes with DELETE.

\textit{index}\n
Numeric expression whose data type either is \texttt{PLS\_INTEGER} or can be implicitly converted to \texttt{PLS\_INTEGER} (for information about the latter, see "s").

**EXISTS**

Function that returns \texttt{TRUE} if the \textit{index}th element of the collection exists and \texttt{FALSE} otherwise, explained in "EXISTS Collection Method".

**EXTEND**

Procedure that adds elements to the end of the collection, explained in "EXTEND Collection Method".

**Restriction on EXTEND**

You cannot use EXTEND if \textit{collection} is an associative array.

**FIRST**

Function that returns the first index in the collection, explained in "FIRST and LAST Collection Methods".

**LAST**

Function that returns the last index in the collection, explained in "FIRST and LAST Collection Methods".

**LIMIT**

Function that returns the maximum number of elements that the collection can have. If the collection has no maximum size, then LIMIT returns NULL. For an example, see "LIMIT Collection Method".

**NEXT**

Function that returns the index of the succeeding existing element of the collection, if one exists. Otherwise, NEXT returns NULL. For more information, see "PRIOR and NEXT Collection Methods".
PRIOR

Function that returns the index of the preceding existing element of the collection, if one exists. Otherwise, NEXT returns NULL. For more information, see "PRIOR and NEXT Collection Methods".

TRIM

Procedure that deletes elements from the end of a collection, explained in "TRIM Collection Method".

Restriction on TRIM

You cannot use TRIM if collection is an associative array.

number

Number of elements to delete from the end of a collection. Default: one.

Examples

• Example 6-22, "DELETE Method with Nested Table"
• Example 6-23, "DELETE Method with Associative Array Indexed by String"
• Example 6-24, "TRIM Method with Nested Table"
• Example 6-25, "EXTEND Method with Nested Table"
• Example 6-26, "EXISTS Method with Nested Table"
• Example 6-27, "FIRST and LAST Values for Associative Array Indexed by PLS_INTEGER"
• Example 6-28, "FIRST and LAST Values for Associative Array Indexed by String"
• Example 6-29, "Printing Varray with FIRST and LAST in FOR LOOP"
• Example 6-30, "Printing Nested Table with FIRST and LAST in FOR LOOP"
• Example 6-31, "COUNT and LAST Values for Varray"
• Example 6-32, "COUNT and LAST Values for Nested Table"
• Example 6-33, "LIMIT and COUNT Values for Different Collection Types"
• Example 6-34, "PRIOR and NEXT Methods"
• Example 6-35, "Printing Elements of Sparse Nested Table"

Related Topics

In this chapter:

• "Collection Variable Declaration"

In other chapters:

• "Collection Methods"
14.11 Collection Variable Declaration

A **collection variable** is a composite variable whose internal components, called elements, have the same data type.

The value of a collection variable and the values of its elements can change.

You reference an entire collection by its name. You reference a collection element with the syntax `collection(index)`.

PL/SQL has three kinds of collection types:

- Associative array (formerly called PL/SQL table or index-by table)
- Variable-size array (varray)
- Nested table

An associative array can be indexed by either a string type or `PLS_INTEGER`. Varrays and nested tables are indexed by integers.

You can create a collection variable in either of these ways:

- Define a collection type and then declare a variable of that type.
- Use `%TYPE` to declare a collection variable of the same type as a previously declared collection variable.

**Note:**

This topic applies to collection types that you define inside a PL/SQL block or package, which differ from standalone collection types that you create with the "CREATE TYPE Statement".

In a PL/SQL block or package, you can define all three collection types. With the `CREATE TYPE` statement, you can create nested table types and `VARRAY` types, but not associative array types.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
collection_type_definition ::= 
```

```
TYPE type IS
assoc_array_type_def
varray_type_def
nested_table_type_def
;
```
assoc_array_type_def ::= 

\[
\text{TABLE \ OF \ datatype \ NOT \ NULL \ INDEX \ BY}
\]

\[
\begin{array}{c}
\text{PLS\_INTEGER} \\
\text{BINARY\_INTEGER} \\
\text{VARCHAR2} \\
\text{VARCHAR} \\
\text{STRING} \\
\text{LONG} \\
\text{type\_attribute} \\
\text{rowtype\_attribute}
\end{array}
\]

(\text{datatype ::=, rowtype\_attribute ::=, type\_attribute ::=})

varray_type_def ::= 

\[
\begin{array}{c}
\text{VARRAY} \\
\text{VARYING} \\
\text{ARRAY}
\end{array}
\]

(\text{datatype ::=})

nested_table_type_def ::= 

\[
\text{TABLE \ OF \ datatype \ NOT \ NULL}
\]

(\text{datatype ::=})

collection_variable_decl ::= 

new_collection_var

assoc_array_type
:=
qualified_expression
function\_call
collection_var_1

varray_type
nested_table_type
:=
collection\_constructor
collection_var_1
collection_var_2 %TYPE
(collection_constructor ::=, function_call ::=, qualified_expression ::=)

Semantics

collection_type_definition

type
Name of the collection type that you are defining.

assoc_array_type_def
Type definition for an associative array.

Restriction on assoc_array_type_def
Can appear only in the declarative part of a block, subprogram, package specification, or package body.

datatype
Data type of the elements of the associative array. datatype can be any PL/SQL data type except REF CURSOR.

NOT NULL
Imposes the NOT NULL constraint on every element of the associative array. For information about this constraint, see "NOT NULL Constraint".

{ PLS_INTEGER | BINARY_INTEGER }
Specifies that the data type of the indexes of the associative array is PLS_INTEGER.

{ VARCHAR2 | VARCHAR | STRING } (v_size)
Specifies that the data type of the indexes of the associative array is VARCHAR2 (or its subtype VARCHAR or STRING) with length v_size.

You can populate an element of the associative array with a value of any type that can be converted to VARCHAR2 with the TO_CHAR function (described in Oracle Database SQL Language Reference).

⚠️ Caution:

Associative arrays indexed by strings can be affected by National Language Support (NLS) parameters. For more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String".

LONG
Specifies that the data type of the indexes of the associative array is LONG, which is equivalent to VARCHAR2 (32760).
Note:

Oracle supports LONG only for backward compatibility with existing applications. For new applications, use VARCHAR2 (32760).

**type_attribute, rowtype_attribute**

Specifies that the data type of the indexes of the associative array is a data type specified with either %ROWTYPE or %TYPE. This data type must represent either PLS_INTEGER, BINARY_INTEGER, or VARCHAR2 (v_size).

**varray_type_def**

Type definition for a variable-size array.

**size_limit**

Maximum number of elements that the varray can have. size_limit must be an integer literal in the range from 1 through 2147483647.

**datatype**

Data type of the varray element. datatype can be any PL/SQL data type except REF CURSOR.

**NOT NULL**

Imposes the NOT NULL constraint on every element of the varray. For information about this constraint, see "NOT NULL Constraint".

**nested_table_type_def**

Type definition for a nested table.

**datatype**

Data type of the elements of the nested table. datatype can be any PL/SQL data type except REF CURSOR or NCLOB.

If datatype is a scalar type, then the nested table has a single column of that type, called COLUMN_VALUE.

If datatype is an ADT, then the columns of the nested table match the name and attributes of the ADT.

**NOT NULL**

Imposes the NOT NULL constraint on every element of the nested table. For information about this constraint, see "NOT NULL Constraint".

**collection_variable_decl**

**new_collection_var**

Name of the collection variable that you are declaring.

**assoc_array_type**
Name of a previously defined associative array type; the data type of
new_collection_var.

`varray_type`

Name of a previously defined VARRAY type; the data type of new_collection_var.

`nested_table_type`

Name of a previously defined nested table type; the data type of new_collection_var.

`collection_constructor`

Collection constructor for the data type of new_collection_var, which provides the initial value of new_collection_var.

`collection_var_1`

Name of a previously declared collection variable of the same data type as new_collection_var, which provides the initial value of new_collection_var.


---

**Note:**

`collection_var_1` and `new_collection_var` must have the same data type, not only elements of the same type.

---

`collection_var_2`

Name of a previously declared collection variable.

`%TYPE`

See "%TYPE Attribute".

Examples

- Example 6-1, "Associative Array Indexed by String"
- Example 6-2, "Function Returns Associative Array Indexed by PLS_INTEGER"
- Example 6-4, "Varray (Variable-Size Array)"
- Example 6-5, "Nested Table of Local Type"
- Example 6-16, "Two-Dimensional Varray (Varray of Varrays)"
- Example 6-17, "Nested Tables of Nested Tables and Varrays of Integers"

Related Topics

- "Qualified Expressions Overview"
- "Collection Topics"
- "BULK COLLECT Clause"
- "CREATE TYPE Statement"
- "Collection Method Invocation"
- "FORALL Statement"
14.12 Comment

A comment is source program text that the PL/SQL compiler ignores. Its primary purpose is to document code, but you can also use it to disable obsolete or unfinished pieces of code (that is, you can turn the code into comments). PL/SQL has both single-line and multiline comments.

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

`comment ::=`

```
-- text
```

Semantics

`comment`

`--`

Turns the rest of the line into a single-line comment. Any text that wraps to the next line is not part of the comment.

⚠️ Caution:

Do not put a single-line comment in a PL/SQL block to be processed dynamically by an Oracle Precompiler program. The Oracle Precompiler program ignores end-of-line characters, which means that a single-line comment ends when the block ends.

```
/*

Begins a comment, which can span multiple lines.

*/

Ends a comment.
Restriction on text

In a multiline comment, text cannot include the multiline comment delimiter /* or */. Therefore, one multiline comment cannot contain another multiline comment. However, a multiline comment can contain a single-line comment.

Examples

• Example 3-6, “Single-Line Comments”
• Example 3-7, "Multiline Comments"

Related Topics

• "Comments"

14.13 COMPILE Clause

The compile clause explicitly recompiles a stored unit that has become invalid, thus eliminating the need for implicit runtime recompilation and preventing associated runtime compilation errors and performance overhead.

The COMPILE clause can appear in the following SQL statements:

• ALTER FUNCTION Statement
• ALTER PACKAGE Statement
• ALTER PROCEDURE Statement
• ALTER LIBRARY Statement
• ALTER TYPE Statement
• ALTER TRIGGER Statement

Topics

• Syntax
• Semantics
• Related Topics

Syntax

compile_clause ::=
**compiler_parameters_clause ::=**

- **parameter_name** = **parameter_value**

**Semantics**

**compile_clause**

**COMPILE**

Recompiles the PL/SQL unit, whether it is valid or invalid. The PL/SQL unit can be a library, package, package specification, package body, trigger, procedure, function, type, type specification, or type body.

First, if any of the objects upon which the unit depends are invalid, the database recompiles them.

The database also invalidates any local objects that depend upon the unit.

If the database recompiles the unit successfully, then the unit becomes valid. Otherwise, the database returns an error and the unit remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

**DEBUG**

Has the same effect as `PLSQL_OPTIMIZE_LEVEL=1`—instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Oracle recommends using `PLSQL_OPTIMIZE_LEVEL=1` instead of `DEBUG`.

**PACKAGE**

*(Default)* Recompiles both the package specification and (if it exists) the package body, whether they are valid or invalid. The recompilation of the package specification and body lead to the invalidation and recompilation of dependent objects as described for **SPECIFICATION** and **BODY**.

**Restriction on PACKAGE**

**PACKAGE** may only appear if compiling a package.

**SPECIFICATION**

Recompiles only the package or type specification, whether it is valid or invalid. You might want to recompile a package or type specification to check for compilation errors after modifying the specification.

When you recompile a specification, the database invalidates any local objects that depend on the specification, such as procedures that invoke procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.
Restriction on SPECIFICATION

SPECIFICATION may only appear if compiling a package or type specification.

BODY

Recompiles only the package or type body, whether it is valid or invalid. You might want to recompile a package or type body after modifying it. Recompiling a body does not invalidate objects that depend upon its specification.

When you recompile a package or type body, the database first recompiles the objects on which the body depends, if any of those objects are invalid. If the database recompiles the body successfully, then the body becomes valid.

Restriction on BODY

BODY may only appear if compiling a package or type body.

REUSE SETTINGS

Prevents Oracle Database from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in this statement.

See also DEFAULT COLLATION Clause compilation semantics.

compiler_parameters_clause

Specifies a value for a PL/SQL compilation parameter in Table 2-2. The compile-time value of each of these parameters is stored with the metadata of the PL/SQL unit being compiled.

You can specify each parameter only once in each statement. Each setting is valid only for the PL/SQL unit being compiled and does not affect other compilations in this session or system. To affect the entire session or system, you must set a value for the parameter using the ALTER SESSION or ALTER SYSTEM statement.

If you omit any parameter from this clause and you specify REUSE SETTINGS, then if a value was specified for the parameter in an earlier compilation of this PL/SQL unit, the database uses that earlier value. If you omit any parameter and either you do not specify REUSE SETTINGS or no value was specified for the parameter in an earlier compilation, then the database obtains the value for that parameter from the session environment.

Related Topics

In other books:

• Oracle Database Development Guide for information about debugging procedures
• Oracle Database Development Guide for information about debugging a trigger using the same facilities available for stored subprograms
14.14 Constant Declaration

A constant holds a value that does not change. A constant declaration specifies the name, data type, and value of the constant and allocates storage for it. The declaration can also impose the NOT NULL constraint.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\[
\text{constant\_declaration} ::= \text{constant} \ \text{CONSTANT} \ \text{datatype} \ \text{NOT} \ \text{NULL} \ \text{DEFAULT} \ \text{expression};
\]

\[(\text{datatype} ::=, \ \text{expression} ::=)\]

Semantics

\text{constant\_declaration}

\text{constant}

Name of the constant that you are declaring.

\text{datatype}

Data type for which a variable can be declared with an initial value.

\textbf{NOT NULL}

Imposes the NOT NULL constraint on the constant.

For information about this constraint, see "NOT NULL Constraint".

\text{expression}

Initial value for the constant. expression must have a data type that is compatible with datatype. When constant\_declaration is elaborated, the value of expression is assigned to constant.

Examples

- Example 3-12, "Constant Declarations"
- Example 3-13, "Variable and Constant Declarations with Initial Values"
14.15 CONTINUE Statement

The `CONTINUE` statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

If a `CONTINUE` statement exits a cursor `FOR` loop prematurely (for example, to exit an inner loop and transfer control to the next iteration of an outer loop), the cursor closes (in this context, `CONTINUE` works like `GOTO`).

The `CONTINUE WHEN` statement exits the current iteration of a loop when the condition in its `WHEN` clause is true, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

Each time control reaches the `CONTINUE WHEN` statement, the condition in its `WHEN` clause is evaluated. If the condition is not true, the `CONTINUE WHEN` statement does nothing.

Restrictions on CONTINUE Statement

- A `CONTINUE` statement must be inside a `LOOP` statement.
- A `CONTINUE` statement cannot cross a subprogram or method boundary.

Syntax

```
continue_statement ::= ...
```
(boolean_expression ::=)

Semantics

continue_statement

label

Name that identifies either the current loop or an enclosing loop.

Without label, the CONTINUE statement transfers control to the next iteration of the current loop. With label, the CONTINUE statement transfers control to the next iteration of the loop that label identifies.

WHEN boolean_expression

Without this clause, the CONTINUE statement exits the current iteration of the loop unconditionally. With this clause, the CONTINUE statement exits the current iteration of the loop if and only if the value of boolean_expression is TRUE.

Examples

Example 14-12 CONTINUE Statement in Basic LOOP Statement

In this example, the CONTINUE statement inside the basic LOOP statement transfers control unconditionally to the next iteration of the current loop.

DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/

Result:

Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Example 14-13  CONTINUE WHEN Statement in Basic LOOP Statement

In this example, the CONTINUE WHEN statement inside the basic LOOP statement transfers control to the next iteration of the current loop when x is less than 3.

```plsql
DECLARE
    x NUMBER := 0;
BEGIN
    LOOP -- After CONTINUE statement, control resumes here
        DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
        x := x + 1;
        CONTINUE WHEN x < 3;
        DBMS_OUTPUT.PUT_LINE ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
        EXIT WHEN x = 5;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5

Related Topics

- "LOOP Statements" for more conceptual information
- "Basic LOOP Statement" for more information about labelling loops
- "Cursor FOR LOOP Statement"
- "EXIT Statement"
- "Expression"
- "FOR LOOP Statement"
- "WHILE LOOP Statement"

14.16 COVERAGE Pragma

The COVERAGE pragma marks PL/SQL code which is infeasible to test for coverage. These marks improve coverage metric accuracy.

The COVERAGE pragma marks PL/SQL source code to indicate that the code may not be feasibly tested for coverage. The pragma marks a specific code section. Marking infeasible code improves the quality of coverage metrics used to assess how much testing has been achieved.
Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

coveragePragma ::=  

\[
\text{PRAGMA COVERAGE (coveragePragma_argument)}; 
\]

coveragePragma_argument ::=  

\[
'\text{NOT_FEASIBLE}' \\
'\text{NOT_FEASIBLE_START}' \\
'\text{NOT_FEASIBLE_END}' 
\]

Semantics

coveragePragma

The COVERAGE pragma may appear before any declaration or statement.

coveragePragma_argument

The COVERAGE pragma argument must have one of these values:

- 'NOT_FEASIBLE'
- 'NOT_FEASIBLE_START'
- 'NOT_FEASIBLE_END'

When the COVERAGE pragma appear with the argument 'NOT_FEASIBLE', it marks the entire basic block that includes the beginning of the first declaration or statement that follows the pragma.

A COVERAGE pragma with an argument of 'NOT_FEASIBLE_START' may appear before any declaration or any statement. It must be followed by the COVERAGE pragma with an argument of 'NOT_FEASIBLE_END'. The second pragma may appear before any declaration or any statement. It must appear in the same PL/SQL block as the first pragma and not in any nested subprogram definition.

An associated pair of COVERAGE pragmas marks basic blocks infeasible from the beginning of the basic block that includes the beginning of the first statement or declaration that follows the first pragma to the end of the basic block that includes the first statement or declaration that follows the second pragma.
A `COVERAGE` pragma whose range includes the definition or declaration of an inner subprogram does not mark the blocks of that subprogram as infeasible.

Examples

Example 14-14  Marking a Single Basic Block as Infeasible to Test for Coverage

This example shows the placement of the pragma `COVERAGE` preceding the assignments to z and zl basic blocks. These two basic blocks will be ignored for coverage calculation. The first `COVERAGE` pragma (marked 1) marks the first assignment to z infeasible; the second (marked 2) marks the third assignment to z. In each case, the affected basic block runs from the identifier z to the following `END IF`.

```plaintext
IF (x>0) THEN
  y :=2;
ELSE
  PRAGMA COVERAGE ('NOT_FEASIBLE'); -- 1
  z:=3;
END IF;
IF (y>0) THEN
  z :=2;
ELSE
  PRAGMA COVERAGE ('NOT_FEASIBLE'); -- 2
  z :=3;
END IF;
```

Example 14-15  Marking a Line Range as Infeasible to Test for Coverage

This example shows marking the entire line range as not feasible. A line range may contain more than one basic block. A line range is marked as not feasible for coverage using a pragma `COVERAGE` with a 'NOT_FEASIBLE_START' argument at the beginning of the range, and a pragma `COVERAGE` with a 'NOT_FEASIBLE_END' at the end of the range. The range paired `COVERAGE` pragmas mark all the blocks as infeasible.

```plaintext
PRAGMA COVERAGE ('NOT_FEASIBLE_START');
IF (x>0) THEN
  y :=2;
ELSE
  z:=3;
END IF;
IF (y>0) THEN
  z :=2;
ELSE
  PRAGMA COVERAGE ('NOT_FEASIBLE_START');
  z :=3;
END IF;
PRAGMA COVERAGE ('NOT_FEASIBLE_END');
```

Example 14-16  Marking Entire Units or Individual Subprograms as Infeasible to Test for Coverage

This example shows marking the entire procedure foo as not feasible for coverage. A subprogram is marked as completely infeasible by marking all of its body infeasible.

```plaintext
CREATE PROCEDURE foo IS
  PRAGMA COVERAGE ('NOT_FEASIBLE_START');
```

Chapter 14  COVERAGE Pragma
Example 14-17 Marking Internal Subprogram as Infeasible to Test for Coverage

This example shows that the outer COVERAGE pragma pair has no effect on coverage inside procedure inner. The COVERAGE pragma (marked 1) inside the body of inner does mark the second assignment to x as infeasible. Notice that the entire body of procedure outer is marked infeasible even though the pragma with argument ‘NOT_FEASIBLE_END’ is not the last line. The pragma does mark the basic block that includes the statement that follows the pragma and that block does extend to the end of the procedure.

CREATE OR REPLACE PROCEDURE outer IS
  x NUMBER := 7;
  PROCEDURE inner IS
    BEGIN
      IF x < 6 THEN
        x := 19;
      ELSE
        PRAGMA COVERAGE ('NOT_FEASIBLE'); -- 1
        x := 203;
      END IF;
    END;
    BEGIN
      DBMS_OUTPUT.PUT_LINE ('X= ');
      PRAGMA COVERAGE ('NOT_FEASIBLE_END');
      DBMS_OUTPUT.PUT_LINE (x);
    END;
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('X= ');
  PRAGMA COVERAGE ('NOT_FEASIBLE_END');
  DBMS_OUTPUT.PUT_LINE (x);
END;
/

Related Topics

In this book:

- Pragmas
- PL/SQL Units and Compilation Parameters for more information about the PLSQL_OPTIMIZE_LEVEL compilation parameter

In other books:

- Oracle Database Development Guide for more information about using PL/SQL basic block coverage to maintain quality
- Oracle Database PL/SQL Packages and Types Reference for more information about using the DBMS_PLSQL_CODE_COVERAGE package
14.17 Cursor FOR LOOP Statement

The cursor FOR LOOP statement implicitly declares its loop index as a record variable of the row type that a specified cursor returns, and then opens a cursor.

With each iteration, the cursor FOR LOOP statement fetches a row from the result set into the record. When there are no more rows to fetch, the cursor FOR LOOP statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or raises an exception.

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

cursor_for_loop_statement ::= 

Semantics

cursor_for_loop_statement
record
Name for the loop index that the cursor FOR LOOP statement implicitly declares as a %ROWTYPE record variable of the type that cursor or select_statement returns.

record is local to the cursor FOR LOOP statement. Statements inside the loop can reference record and its fields. They can reference virtual columns only by aliases. Statements outside the loop cannot reference record. After the cursor FOR LOOP statement runs, record is undefined.

cursor
Name of an explicit cursor (not a cursor variable) that is not open when the cursor FOR LOOP is entered.

actual_cursor_parameter
Actual parameter that corresponds to a formal parameter of `cursor`.

**select_statement**

SQL `SELECT` statement (not PL/SQL `SELECT INTO` statement). For `select_statement`, PL/SQL declares, opens, fetches from, and closes an implicit cursor. However, because `select_statement` is not an independent statement, the implicit cursor is internal—you cannot reference it with the name SQL.

**See Also:**

*Oracle Database SQL Language Reference* for `SELECT` statement syntax

**label**

Label that identifies `cursor_for_loop_statement` (see "statement ::=" and "label"). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label in the END LOOP statement matches a label at the beginning of the same LOOP statement (the compiler does not check).

**Examples**

- Example 7-18, "Implicit Cursor FOR LOOP Statement"
- Example 7-19, "Explicit Cursor FOR LOOP Statement"
- Example 7-20, "Passing Parameters to Explicit Cursor FOR LOOP Statement"
- Example 7-21, "Cursor FOR Loop References Virtual Columns"

**Related Topics**

In this chapter:

- "Basic LOOP Statement"
- "CONTINUE Statement"
- "EXIT Statement"
- "Explicit Cursor Declaration and Definition"
- "FETCH Statement"
- "FOR LOOP Statement"
- "FORALL Statement"
- "OPEN Statement"
- "WHILE LOOP Statement"

In other chapters:

- "Processing Query Result Sets With Cursor FOR LOOP Statements"
14.18 Cursor Variable Declaration

A cursor variable is like an explicit cursor that is not limited to one query.

To create a cursor variable, either declare a variable of the predefined type SYS_REFCURSOR or define a REF_CURSOR type and then declare a variable of that type.

Restrictions on Cursor Variables

• You cannot declare a cursor variable in a package specification. That is, a package cannot have a public cursor variable (a cursor variable that can be referenced from outside the package).

• You cannot store the value of a cursor variable in a collection or database column.

• You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.

• Using a cursor variable in a server-to-server remote procedure call (RPC) causes an error. However, you can use a cursor variable in a server-to-server RPC if the remote database is a non-Oracle database accessed through a Procedural Gateway.

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

ref_cursor_type_definition ::= 

```
TYPE type IS REF CURSOR
RETURN
db_table_or_view
cursor
cursor_variable
% ROWTYPE
record % TYPE
record_type
ref_cursor_type
;
```

cursor_variable_declaration ::= 

```
(cursor_variable type)
```
Semantics

ref_cursor_type_definition

type
Name of the REF CURSOR type that you are defining.

RETURN
Specifies the data type of the value that the cursor variable returns.

Specify RETURN to define a strong REF CURSOR type. Omit RETURN to define a weak REF CURSOR type. For information about strong and weak REF CURSOR types, see "Creating Cursor Variables".

db_table_or_view
Name of a database table or view, which must be accessible when the declaration is elaborated.

cursor
Name of a previously declared explicit cursor.

cursor_variable
Name of a previously declared cursor variable.

record
Name of a user-defined record.

record_type
Name of a user-defined type that was defined with the data type specifier RECORD.

ref_cursor_type
Name of a user-defined type that was defined with the data type specifier REF CURSOR.

cursor_variable_declaration

cursor_variable
Name of the cursor variable that you are declaring.

type
Type of the cursor variable that you are declaring—either SYS_REFCursor or the name of the REF CURSOR type that you defined previously.

SYS_REFCursor is a weak type. For information about strong and weak REF CURSOR types, see "Creating Cursor Variables".

Examples

• Example 7-24, "Cursor Variable Declarations"
• Example 7-25, "Cursor Variable with User-Defined Return Type"
• Example 7-28, "Variable in Cursor Variable Query—No Result Set Change"
• Example 7-29, "Variable in Cursor Variable Query—Result Set Change"
• Example 7-30, "Querying a Collection with Static SQL"
• Example 7-31, "Procedure to Open Cursor Variable for One Query"
• Example 7-32, "Opening Cursor Variable for Chosen Query (Same Return Type)"
• Example 7-33, "Opening Cursor Variable for Chosen Query (Different Return Types)"
• Example 7-34, "Cursor Variable as Host Variable in Pro*C Client Program"

Related Topics
In this chapter:
• "CLOSE Statement"
• "Named Cursor Attribute"
• "Explicit Cursor Declaration and Definition"
• "FETCH Statement"
• "OPEN FOR Statement"
• "%ROWTYPE Attribute"
• "%TYPE Attribute"

In other chapters:
• "Cursor Variables"
• "Passing CURSOR Expressions to Pipelined Table Functions"

14.19 Datatype Attribute

The data type attribute of an ADT element.

A datatype allows you to declare the data type of record variables fields, constants, functions return value, collection variables and collection types elements.

• Record Variable Declaration
• Constant Declaration
• Function Declaration and Definition
• Collection Variable Declaration
• CREATE FUNCTION Statement
• CREATE TYPE Statement

Topics
• Syntax
• Semantics
• Related Topics
Syntax

datatype ::= 

(collection_type
REF
object_type
record_type
ref_cursor_type
rowtype_attribute
scalar_datatype
type_attribute

(rowtype_attribute ::=, type_attribute ::=)

Semantics

datatype
collection_type
Name of a user-defined varray or nested table type (not the name of an associative array type).

object_type
Instance of a user-defined type.

record_type
Name of a user-defined type that was defined with the data type specifier RECORD.

ref_cursor_type
Name of a user-defined type that was defined with the data type specifier REF CURSOR.

scalar_datatype
Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics.

Related Topics
In other chapters:

• PL/SQL Data Types

14.20 DEFAULT COLLATION Clause

Collation (also called sort ordering) determines if a character string equals, precedes, or follows another string when the two strings are compared and sorted. Oracle
Database collations order strings following rules for sorted text used in different languages.

The `DEFAULT COLLATION` clause can appear in the following SQL statements:

- CREATE FUNCTION Statement
- CREATE PROCEDURE Statement
- CREATE PACKAGE Statement
- CREATE TRIGGER Statement
- CREATE TYPE Statement

**Topics**

- Prerequisites
- Syntax
- Semantics
- Compilation Semantics
- Related Topics

**Prerequisites**

The `COMPATIBLE` initialization parameter must be set to at least 12.2.0, and `MAX_STRING_SIZE` must be set to `EXTENDED` for collation declarations to be allowed in these SQL statements.

**Syntax**

```
default_collation_clause ::=  

   DEFAULT COLLATION collation_option
```

```
collation_option ::=  

   USING_NLS_COMP
```

**Semantics**

`default_collation_clause`

The `default_collation_clause` can appear in a package specification, a standalone type specification, and in standalone subprograms.

`collation_option`

The default collation of a procedure, function, package, type, or trigger must be `USING_NLS_COMP`. The `default_collation_clause` explicitly declares the default collation of a PL/SQL unit to be `USING_NLS_COMP`. Without this clause, the unit inherits its default collation from the effective schema default collation. If the effective schema default collation is not `USING_NLS_COMP`, the unit is invalid.
The effective schema default collation is determined as follows:

- If the session parameter `DEFAULT_COLLATION` is set, the effective schema default collation is the value of this parameter. The value of the parameter can be checked by querying `SYS_CONTEXT('USERENV', 'SESSION_DEFAULT_COLLATION')`. The function returns NULL if `DEFAULT_COLLATION` is not set. The value of the parameter `DEFAULT_COLLATION` can be set with the statement: `ALTER SESSION SET DEFAULT_COLLATION = collation_option;`

- If the session parameter `DEFAULT_COLLATION` is not set, the effective schema default collation is the declared default collation of the schema in which you create the PL/SQL unit. The default collation of a schema can be found in the static data dictionary *_USERS* views. It can be set with the DDL statements `CREATE USER` and `ALTER USER`.

The session parameter `DEFAULT_COLLATION` can be unset with the statement: `ALTER SESSION SET DEFAULT_COLLATION = NONE;`

Package body and type body use the default collation of the corresponding specification. All character data containers and attributes in procedures, functions and methods, including parameters and return values, behave as if their data-bound collation were the pseudo-collation `USING_NLS_COMP`.

Restrictions on `DEFAULT COLLATION`

It cannot be specified for nested or packaged subprograms or for type methods.

Compilation Semantics

If the resulting default object collation is different from `USING_NLS_COMP`, the database object is created as invalid with a compilation error.

If the `ALTER COMPILE` statement is issued for a PL/SQL unit with the `REUSE SETTINGS` clause, the stored default collation of the database object being compiled is not changed.

If an `ALTER COMPILE` statement is issued without the `REUSE SETTINGS` clause, the stored default collation of the database object being compiled is discarded and the effective schema default collation for the object owner at the time of execution of the statement is stored as the default collation of the object, unless the PL/SQL unit contains the `DEFAULT_COLLATION` clause. If the resulting default collation is not `USING_NLS_COMP`, a compilation error is raised.

An `ALTER COMPILE` statement for a package or type body references the stored collation of the corresponding specification.

Related Topics

In other chapters:

- `ALTER FUNCTION Statement`
- `ALTER PACKAGE Statement`
- `ALTER PROCEDURE Statement`
- `ALTER TRIGGER Statement`
- `ALTER TYPE Statement`

In other books:
14.21 DELETE Statement Extension

The PL/SQL extension to the where_clause of the SQL DELETE statement lets you specify a CURRENT OF clause, which restricts the DELETE statement to the current row of the specified cursor.

For information about the CURRENT OF clause, see “UPDATE Statement Extensions”.

See Also:
Oracle Database SQL Language Reference for the syntax of the SQL DELETE statement

14.22 DEPRECATE Pragma

The DEPRECATE pragma marks a PL/SQL element as deprecated. The compiler issues warnings for uses of pragma DEPRECATE or of deprecated elements.

The associated warnings tell users of a deprecated element that other code may need to be changed to account for the deprecation.

Topics
- Syntax
- Semantics
- DEPRECATE Pragma Compilation Warnings
- Examples
- Related Topics

Syntax

deprecate_pragma ::= 

Semantics

deprecate_pragma

The DEPRECATE pragma may only appear in the declaration sections of a package specification, an object specification, a top level procedure, or a top level function.
PL/SQL elements of these kinds may be deprecated:

- Subprograms
- Packages
- Variables
- Constants
- Types
- Subtypes
- Exceptions
- Cursors

The **DEPRECATE** pragma may only appear in the declaration section of a PL/SQL unit. It must appear immediately after the declaration of an item to be deprecated.

The **DEPRECATE** pragma applies to the PL/SQL element named in the declaration which precedes the pragma.

When the **DEPRECATE** pragma applies to a package specification, object specification, or subprogram, the pragma must appear immediately after the keyword *IS* or *AS* that terminates the declaration portion of the definition.

When the **DEPRECATE** pragma applies to a package or object specification, references to all the elements (of the kinds that can be deprecated) that are declared in the specification are also deprecated.

If the **DEPRECATE** pragma applies to a subprogram declaration, only that subprogram is affected; other overloads with the same name are not deprecated.

If the optional custom message appears in a use of the **DEPRECATE** pragma, the custom message will be added to the warning issued for any reference to the deprecated element.

The identifier in a **DEPRECATE** pragma must name the element in the declaration to which it applies.

Deprecation is inherited during type derivation. A child object type whose parent is deprecated is not deprecated. Only the attributes and methods that are inherited are deprecated.

When the base type is not deprecated but individual methods or attributes are deprecated, and when a type is derived from this type and the deprecated type or method is inherited, then references to these through the derived type will cause the compiler to issue a warning.

A reference to a deprecated element appearing anywhere except in the unit with the deprecation pragma or its body, will cause the PL/SQL compiler to issue a warning for the referenced elements. A reference to a deprecated element in an anonymous block will not cause the compiler to issue a warning; only references in named entities will draw a warning.

When a deprecated entity is referenced in the definition of another deprecated entity then no warning will be issued.

When an older client code refers to a deprecated entity, it is invalidated and recompiled. No warning is issued.

There is no effect when SQL code directly references a deprecated element.
A reference to a deprecated element in a PL/SQL static SQL statement may cause the PL/SQL compiler to issue a warning. However, such references may not be detectable.

**pls_identifier**
Identifier of the PL/SQL element being deprecated.

**character_literal**
An optional compile-time warning message.

**DEPRECATe Pragma Compilation Warnings**
The PL/SQL compiler issues warnings when the **DEPRECATe** pragma is used and when deprecated items are referenced.

- 6019 — The entity was deprecated and could be removed in a future release. Do not use the deprecated entity.
- 6020 — The referenced entity was deprecated and could be removed in a future release. Do not use the deprecated entity. Follow the specific instructions in the warning if any are given.
- 6021 — Misplaced pragma. The pragma **DEPRECATe** should follow immediately after the declaration of the entity that is being deprecated. Place the pragma immediately after the declaration of the entity that is being deprecated.
- 6022 — This entity cannot be deprecated. Deprecation only applies to entities that may be declared in a package or type specification as well as to top-level procedure and function definitions. Remove the pragma.

The **DEPRECATe** pragma warnings may be managed with the **PLSQL_WARNINGS** parameter or with the **DBMS_WARNING** package.

**Examples**

**Example 14-18  Enabling the Deprecation Warnings**
This example shows how to set the **PLSQL_WARNINGS** parameter to enable these warnings in a session.

```
Live SQL:
You can view and run this example on Oracle Live SQL at Restricting Access to Top-Level Procedures in the Same Schema

ALTER SESSION SET PLSQL_WARNINGS='ENABLE:(6019,6020,6021,6022)';
```

**Example 14-19  Deprecation of a PL/SQL Package**
This example shows the deprecation of a PL/SQL package as a whole. Warnings will be issued for any reference to package pack1, and to the procedures foo and bar when used outside of the package and its body.


Live SQL:

You can view and run this example on Oracle Live SQL at Deprecation of a PL/SQL Package

PACKAGE pack1 AS
  PRAGMA DEPRECATE (pack1);
  PROCEDURE foo;
  PROCEDURE bar;
END pack1;

Example 14-20  Deprecation of a PL/SQL Package with a Custom Warning

This example shows the deprecation of a PL/SQL package. The compiler issues a custom warning message when a reference in another unit for the deprecated procedure foo is compiled.

Live SQL:

You can view and run this example on Oracle Live SQL at Deprecation of a PL/SQL Package with a Custom Warning

PACKAGE pack5 AUTHID DEFINER AS
  PRAGMA DEPRECATE (pack5 , 'Package pack5 has been deprecated, use new_pack5 instead. ');
  PROCEDURE foo;
  PROCEDURE bar;
END pack5;

A reference to procedure pack5.foo in another unit would draw a warning like this.

SP2-0810: Package Body created with compilation warnings

Errors for PACKAGE BODY PACK6:
4/10    PLW-06020: reference to a deprecated entity: PACK5 declared in unit PACK5[1,9].
        Package pack5 has been deprecated, use new_pack5 instead

Example 14-21  Deprecation of a PL/SQL Procedure

This example shows the deprecation of a single PL/SQL procedure foo in package pack7.
You can view and run this example on Oracle Live SQL at Deprecation of a PL/SQL Procedure.

Example 14-22 Deprecation of an Overloaded Procedure

This example shows the DEPRECATED pragma applies only to a specific overload of a procedure name. Only the second declaration of proc1 is deprecated.

Example 14-23 Deprecation of a Constant and of an Exception

This example shows the deprecation of a constant and of an exception.
Example 14-24  Using Conditional Compilation to Deprecate Entities in Some Database Releases

This example shows the deprecation of procedure proc1 if the database release version is greater than 11.

```
CREATE PACKAGE pack11 AUTHID DEFINER AS
  $IF DBMS_DB_VERSION.VER_LE_11
  $THEN
    PROCEDURE proc1;
  $ELSE
    PROCEDURE proc1;
    PRAGMA DEPRECATE(proc1);
  $END
  PROCEDURE proc2;
  PROCEDURE proc3;
END pack11;
```

Example 14-25  Deprecation of an Object Type

This example shows the deprecation of an entire object type.

```
TYPE type01 AS OBJECT(
  PRAGMA DEPRECATE (type01),
  y NUMBER,
  MEMBER PROCEDURE proc(x NUMBER),
  MEMBER PROCEDURE proc2(x NUMBER)
);
```
Example 14-26  Deprecation of a Member Function in an Object Type Specification

This example shows the deprecation of member function add2 in an object type specification.

Live SQL:

You can view and run this example on Oracle Live SQL at Deprecation of a Member Function in an Object Type Specification

```sql
TYPE objdata AS OBJECT(
    n1 NUMBER,
    n2 NUMBER,
    n3 NUMBER,
    MEMBER FUNCTION add2 RETURN NUMBER,
    PRAGMA DEPRECATE (add2),
    MEMBER FUNCTION add_all RETURN NUMBER
);
```

Example 14-27  Deprecation of Inherited Object Types

This example shows that a reference to a deprecated entity x declared in unit type15_basetype type body will cause the compiler to issue a warning.

Live SQL:

You can view and run this example on Oracle Live SQL at Deprecation of Inherited Object Types

```sql
TYPE type15_basetype AS OBJECT
(
    x1 NUMBER,
    x  NUMBER,
    PRAGMA DEPRECATE (x),
    MEMBER PROCEDURE f0,
    PRAGMA DEPRECATE (f0),
    MEMBER PROCEDURE f1,
    PRAGMA DEPRECATE (f1),
    MEMBER PROCEDURE f2,
    PRAGMA DEPRECATE (f2),
    MEMBER PROCEDURE f3) NOT FINAL;

TYPE BODY type15_basetype AS
    MEMBER PROCEDURE f0
    IS
    BEGIN
    x := 1;
```
References to the deprecated entities x, f0, and f2 in type15_basetype type body will cause the compiler to issue a warning.

```pli
TYPE type15_subtype UNDER type15_basetype (    y NUMBER ,
    MEMBER PROCEDURE f1(z NUMBER),
    MEMBER PROCEDURE f1(z NUMBER , m1 NUMBER),
    PRAGMA DEPRECATE(f1),
    OVERRIDING MEMBER PROCEDURE f2 );
```

```pli
TYPE BODY type15_subtype AS
    MEMBER PROCEDURE f1(z NUMBER) IS
    BEGIN
        -- deprecation attribute inherited in derived type.
        x := 1;
        x1:= 2;
        SELF.f0;
    END;

    MEMBER PROCEDURE f1(z NUMBER ,
        m1 NUMBER) IS
    BEGIN
        NULL;
    END;
    OVERRIDING MEMBER PROCEDURE f2 IS
    BEGIN
        /* refer to deprecated f2 in supertype */
        (SELF AS type15_basetype).f2;
        /* No warning for a reference to a not deprecated data member in the
```
References to deprecated entities x, f1, and f0 in unit15_basetype will cause the compiler to issue a warning.

```
PROCEDURE test_types3
AS
  e type15_subtype ;
  d type15_basetype ;
BEGIN
  e := type15_subtype (1 ,1 ,1);
  d := type15_basetype (1, 1);
  d.x := 2; -- warning issued
  d.f1; -- warning issued
  e.f1 (4); -- overloaded in derived type. no warning. not deprecated in the derived type.
  e.f1 (1); -- no warning
  e.f0; -- f0 is deprecated in base type. deprecation is inherited.
  warning issued
    -- warning issued for deprecated x in d.x and e.x
    DBMS_OUTPUT.PUT_LINE(to_char(e.x) || to_char(' ') || to_char(d.x));
END;
```

**Example 14-28  Deprecation Only Applies to Top Level Subprogram**

This examples shows that the DEPRECATED pragma may not be used to deprecate a nested procedure. The compiler issues a warning about the misuse of the pragma on the entity. The pragma has no effect.

```
PROCEDURE foo
IS
  PROCEDURE inner_foo
  IS
    PRAGMA DEPRECATED (inner_foo, 'procedure inner_foo is deprecated');
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Executing inner_foo');
    END;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Executing foo');
  END;
```
Example 14-29  Misplaced DEPRECATE Pragma

The DEPRECATE pragma must appear immediately after the declaration of the deprecated item. A warning about the misplaced pragma will be issued and the pragma will have no effect.

Live SQL:
You can view and run this example on Oracle Live SQL at Misplaced DEPRECATE Pragma

```
PROCEDURE bar
IS
BEGIN
  PRAGMA DEPRECATE(bar);
  DBMS_OUTPUT.PUT_LINE('Executing bar.');
END;
```

Example 14-30  Mismatch of the Element Name and the DEPRECATE Pragma Argument

This example shows that if the argument for the pragma does not match the name in the declaration, the pragma is ignored and the compiler does not issue a warning.

Live SQL:
You can view and run this example on Oracle Live SQL at Mismatch of the Element Name and the DEPRECATE Pragma Argument

```
PACKAGE pkg13
AS
  PRAGMA DEPRECATE('pkg13', 'Package pkg13 is deprecated, use pkg03');
  Y NUMBER;
END pkg13;
```

If an identifier is applied with a mismatched name, then the compiler issues a warning about the pragma being misplaced. The pragma has no effect.

```
CREATE PACKAGE pkg17
IS
  PRAGMA DEPRECATE("pkg17");
END pkg17;
```

Related Topics
In this book:
- Pragmas
• PL/SQL Units and Compilation Parameters for more information about setting the PLSQL_WARNINGS compilation parameter

In other books:

• Oracle Development Guide for more information about deprecating packages, subprograms, and types
• Oracle Database PL/SQL Packages and Types Reference for more information about enabling the deprecation warnings using the DBMS_WARNING.ADD_WARNING_SETTING_NUM procedure
• Compile-Time Warnings for more information compilation warnings.

14.23 DETERMINISTIC Clause

The deterministic option marks a function that returns predictable results and has no side effects.

Function-based indexes, virtual column definitions that use PL/SQL functions, and materialized views that have query-rewrite enabled require special function properties. The DETERMINISTIC clause asserts that a function has those properties.

The DETERMINISTIC option can appear in the following statements:

• Function Declaration and Definition
• CREATE FUNCTION Statement
• CREATE PACKAGE Statement
• CREATE TYPE BODY Statement

Topics

• Syntax
• Semantics
• Usage Notes
• Related Topics

Syntax

deterministic_clause ::= deterministic

Semantics

deterministic_clause

DETERMINISTIC

A function is deterministic if the DETERMINISTIC clause appears in either a declaration or the definition of the function.

The DETERMINISTIC clause may appear at most once in a function declaration and at most once in a function definition.
A deterministic function must return the same value on two distinct invocations if the arguments provided to the two invocations are the same.

A deterministic function may not have side effects.

A deterministic function may not raise an unhandled exception.

If a function with a deterministic clause violates any of these semantic rules, the results of its invocation, its value, and the effect on its invoker are all undefined.

Usage Notes

The deterministic clause is an assertion that the function obeys the semantic rules. If the function does not, neither the compiler, SQL execution, or PL/SQL execution may diagnose the problem and wrong results may be silently produced.

You must specify this keyword if you intend to invoke the function in the expression of a function-based index, in a virtual column definition, or from the query of a materialized view that is marked refresh fast or enable query rewrite. When the database encounters a deterministic function, it tries to use previously calculated results when possible rather than reexecuting the function. If you change the function, then you must manually rebuild all dependent function-based indexes and materialized views.

Do not specify deterministic for a function whose result depends on the state of session variables or schema objects, because results might vary across invocations.

Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the return result of the function.

Specifying this clause for polymorphic table function is not allowed.

When the deterministic option appears, the compiler may use the mark to improve the performance of the execution of the function.

It is good programming practice to make functions that fall into these categories deterministic:

- Functions used in a WHERE, ORDER BY, or GROUP BY clause
- Functions that MAP or ORDER methods of a SQL type
- Functions that help determine whether or where a row appears in a result set

Related Topics

In other chapters:

- "Subprogram Side Effects"

In other books:

- CREATE INDEX statement in Oracle Database SQL Language Reference
- Oracle Database Data Warehousing Guide for information about materialized views
- Oracle Database SQL Language Reference for information about function-based indexes
14.24 Element Specification

An element specification specifies each attribute of the ADT.

An element specification can appear in the following SQL statements:

- ALTER TYPE Statement
- CREATE TYPE Statement

Topics

- Syntax
- Semantics

Syntax

\[
element\_spec ::= \\
\text{inheritance\_clauses} \cdot \text{subprogram\_spec} \cdot \text{constructor\_spec} \cdot \text{map\_order\_function\_spec} \cdot \text{restrict\_references\_pragma} ( \\
\text{subprogram\_spec ::= } , \text{constructor\_spec ::= } , \text{map\_order\_function\_spec ::= } , \\
\text{restrict\_references\_pragma ::= })
\]

\[
\text{inheritance\_clauses ::= }
\text{\text{NOT} \cdot \text{OVERRIDING} \cdot \text{FINAL} \cdot \text{INSTANTIABLE}}
\]

\[
\text{subprogram\_spec ::= }
\text{\text{MEMBER} \cdot \text{STATIC}} \cdot \text{procedure\_spec} \cdot \text{function\_spec} ( \\
\text{procedure\_spec ::= } , \text{function\_spec ::= })
\]

\[
\text{procedure\_spec ::= }
\text{PROCEDURE} \cdot \text{procedure\_name} ( \text{parameter} \cdot \text{datatype} ) \cdot \text{IS} \cdot \text{AS} \cdot \text{call\_spec}
\]
Element Specification

Chapter 14

Semantics

\(\text{element_spec}\)


*inheritance clauses*

Specifies the relationship between supertypes and subtypes.

**[NOT] OVERRIDING**

Specifies that this method overrides a *MEMBER* method defined in the supertype. This keyword is required if the method redefines a supertype method. **Default:** NOT OVERRIDING.

**[NOT] FINAL**

Specifies that this method cannot be overridden by any subtype of this type. **Default:** NOT FINAL.

**[NOT] INSTANTIABLE**

Specifies that the type does not provide an implementation for this method. **Default:** all methods are INSTANTIABLE.

**Restriction on NOT INSTANTIABLE**

If you specify **NOT INSTANTIABLE**, then you cannot specify **FINAL** or **STATIC**.

---

**See Also:**

* constructor_spec

---

*subprogram spec*

Specifies a subprogram to be referenced as an ADT attribute. For each such subprogram, you must specify a corresponding method body in the ADT body.

**Restriction on subprogram spec**

You cannot define a **STATIC spec** on a subtype that redefines a *MEMBER* method in its supertype, or vice versa.

**MEMBER**

A subprogram associated with the ADT that is referenced as an attribute. Typically, you invoke *MEMBER* methods in a selfish style, such as `object_expression.method()`. This class of method has an implicit first argument referenced as `SELF` in the method body, which represents the object on which the method was invoked.

---

**See Also:**

"Example 15-33"

---

**STATIC**
A subprogram associated with the ADT. Unlike MEMBER methods, STATIC methods do not have any implicit parameters. You cannot reference SELF in their body. They are typically invoked as `type_name.method()`.

**Restrictions on STATIC**

- You cannot map a MEMBER method in a Java class to a STATIC method in a SQLJ object type.
- For both MEMBER and STATIC methods, you must specify a corresponding method body in the type body for each procedure or function specification.

**See Also:**

"Example 15-34"

### `procedure_spec` or `function_spec`

Specifies the parameters and data types of the procedure or function. If this subprogram does not include the declaration of the procedure or function, then you must issue a corresponding `CREATE TYPE BODY` statement.

**Restriction on `procedure_spec` or `function_spec`**

If you are creating a subtype, then the name of the procedure or function cannot be the same as the name of any attribute, whether inherited or not, declared in the supertype chain.

### `return_clause`

The first form of the `return_clause` is valid only for a function. The syntax shown is an abbreviated form.

**See Also:**

"Collection Method Invocation" for information about method invocation and methods

### `constructor_spec`

Creates a user-defined constructor, which is a function that returns an initialized instance of an ADT. You can declare multiple constructors for a single ADT, if the parameters of each constructor differ in number, order, or data type.

- User-defined constructor functions are always FINAL and INSTANTIABLE, so these keywords are optional.
- The parameter-passing mode of user-defined constructors is always SELF IN OUT. Therefore you need not specify this clause unless you want to do so for clarity.
- `RETURN SELF AS RESULT` specifies that the runtime type of the value returned by the constructor is runtime type of the SELF argument.
map_order_function_spec

You can declare either one MAP method or one ORDER method in a type specification, regardless of how many MEMBER or STATIC methods you declare. If you declare either method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. You must not specify a comparison method to determine the equality of two ADTs.

You cannot declare either MAP or ORDER methods for subtypes. However, a subtype can override a MAP method if the supertype defines a NOT FINAL MAP method. A subtype cannot override an ORDER method at all.

You can specify either MAP or ORDER when mapping a Java class to a SQL type. However, the MAP or ORDER methods must map to MEMBER functions in the Java class.

If neither a MAP nor an ORDER method is specified, then only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method must be specified to determine the equality of two ADTs.

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 database uses the scalars 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. You cannot use an ORDER method because the hash mechanism hashes on the object value.

MAP MEMBER

 Specifies a MAP 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.

If the argument to the MAP method is NULL, then the MAP method returns NULL and the method is not invoked.

See Also:

Oracle Database Object-Relational Developer's Guide for more information about object value comparisons
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` method can have no arguments other than the implicit `SELF` argument.

**Note:**

If `type_name` is to be referenced in queries containing sorts (through an `ORDER BY`, `GROUP BY`, `DISTINCT`, or `UNION` clause) or containing joins, and you want those queries to be parallelized, then you must specify a `MAP` member function.

A subtype cannot define a new `MAP` method, but it can override an inherited `MAP` method.

**ORDER MEMBER**

Specifies an `ORDER` 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.

If either argument to the `ORDER` method is `NULL`, then the `ORDER` method returns `NULL` and the method is not invoked.

When instances of the same ADT definition are compared in an `ORDER BY` clause, the `ORDER` method function is invoked.

An object specification can contain only one `ORDER` method, which must be a function having the return type `NUMBER`.

A subtype can neither define nor override an `ORDER` method.

**Restriction on `map_order_function_spec`**

You cannot add an `ORDER` method to a subtype.

**restrict_referencesPragma**

Deprecated clause, described in "RESTRICT_REFERENCES Pragma".

**Restriction on `restrict_referencesPragma`**

This clause is not valid when dropping a method.

### 14.25 EXCEPTION_INIT Pragma

The `EXCEPTION_INIT` pragma associates a user-defined exception name with an error code.

The `EXCEPTION_INIT` pragma can appear only in the same declarative part as its associated exception, anywhere after the exception declaration.

**Topics**

- Syntax
Semantics

exception_init_pragma

exception

Name of a previously declared user-defined exception.

error_code

Error code to be associated with `exception.error_code` can be either 100 (the numeric code for "no data found" that "SQLCODE Function" returns) or any negative integer greater than -10000000 except -1403 (another numeric code for "no data found").

Note:

NO_DATA_FOUND is a predefined exception.

If two `EXCEPTION_INIT` pragmas assign different error codes to the same user-defined exception, then the later pragma overrides the earlier pragma.

Examples

- Example 12-5, "Naming Internally Defined Exception"
- Example 12-13, "Raising User-Defined Exception with RAISE_APPLICATION_ERROR"
- Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"

Related Topics

In this chapter:

- "Exception Declaration"
- "Exception Handler"
- "SQLCODE Function"
- "SQLERRM Function"

In other chapters:
14.26 Exception Declaration

An exception declaration declares the name of a user-defined exception. You can use the `EXCEPTION_INIT` pragma to assign this name to an internally defined exception.

**Syntax**

```
exception_declaration ::= exception EXCEPTION;
```

**Semantics**

```
exception_declaration
exception
Name of the exception that you are declaring.

Restriction on exception
You can use `exception` only in an `EXCEPTION_INIT` pragma, `RAISE` statement, `RAISE_APPLICATION_ERROR` invocation, or exception handler.
```

**Caution:**
Oracle recommends against using a predefined exception name for `exception`. For details, see "Redeclared Predefined Exceptions". For a list of predefined exception names, see Table 12-3.

**Examples**

- Example 12-5, "Naming Internally Defined Exception"
- Example 12-9, "Redeclared Predefined Identifier"
- Example 12-10, "Declaring, Raising, and Handling User-Defined Exception"
14.27 Exception Handler

An exception handler processes a raised exception.

Exception handlers appear in the exception-handling parts of anonymous blocks, subprograms, triggers, and packages.

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\[ \text{exception_handler ::= WHEN } \\text{exception}\ OR \OTHERS \THEN \text{statement} \]

Semantics

\[ \text{exception_handler}
\]

\[ \text{exception}
\]

Name of either a predefined exception (see Table 12-3) or a user-defined exception (see “Exception Declaration”).

If PL/SQL raises a specified exception, then the associated statements run.

\[ \OTHERS \]
Specifies all exceptions not explicitly specified in the exception handling part of the block. If PL/SQL raises such an exception, then the associated statements run.

**Note:**
Oracle recommends that the last statement in the *OTHERS* exception handler be either `RAISE` or an invocation of a subroutine marked with pragma `SUPPRESSES_WARNING_6009`.

If you do not follow this practice, and PL/SQL warnings are enabled, you get **PLW-06009**.

In the exception handling part of a block, the *WHEN OTHERS* exception handler is optional. It can appear only once, as the last exception handler in the exception handling part of the block.

**Examples**

- **Example 12-3**, "Single Exception Handler for Multiple Exceptions"
- **Example 12-4**, "Locator Variables for Statements that Share Exception Handler"
- **Example 12-6**, "Anonymous Block Handles *ZERO_DIVIDE*"
- **Example 12-7**, "Anonymous Block Avoids *ZERO_DIVIDE*"
- **Example 12-10**, "Declaring, Raising, and Handling User-Defined Exception"
- **Example 12-14**, "Exception that Propagates Beyond Scope is Handled"
- **Example 12-24**, "Exception Handler Runs and Execution Ends"
- **Example 12-25**, "Exception Handler Runs and Execution Continues"
- **Example 13-12**, "Handling FORALL Exceptions Immediately"
- **Example 13-13**, "Handling FORALL Exceptions After FORALL Statement Completes"

**Related Topics**

In this chapter:

- "Block"
- "EXCEPTION_INIT Pragma"
- "Exception Declaration"
- "RAISE Statement"
- "SQLCODE Function"
- "SQLERRM Function"

In other chapters:

- "Overview of Exception Handling"
- "Continuing Execution After Handling Exceptions"
- "Retrying Transactions After Handling Exceptions"
The EXECUTE IMMEDIATE statement builds and runs a dynamic SQL statement in a single operation.

Native dynamic SQL uses the EXECUTE IMMEDIATE statement to process most dynamic SQL statements.

Caution:
When using dynamic SQL, beware of SQL injection, a security risk. For more information about SQL injection, see "SQL Injection".

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```sql
execute_immediate_statement ::= EXECUTE IMMEDIATE dynamic_sql_stmt
into_clause
bulk_collect_into_clause
using_clause
using_clause
dynamic_returning_clause
dynamic_returning_clause
```

(bulk_collect_into_clause ::=, dynamic_returning_clause ::=, into_clause ::=)

**using_clause ::=**
Semantics

**execute_immediate_statement**

*dynamic_sql_stmt*

String literal, string variable, or string expression that represents a SQL statement. Its type must be either **CHAR**, **VARCHAR2**, or **CLOB**.

---

**Note:**

If *dynamic_sql_statement* is a SELECT statement, and you omit both *into_clause* and *bulk_collect_into_clause*, then *execute_immediate_statement* never executes.

For example, this statement never increments the sequence:

EXECUTE IMMEDIATE 'SELECT S.NEXTVAL FROM DUAL'

---

**into_clause**

Specifies the variables or record in which to store the column values that the statement returns. For more information about this clause, see "**RETURNING INTO Clause**".

**Restriction on into_clause**

Use if and only if *dynamic_sql_stmt* returns a single row.

**bulk_collect_into_clause**

Specifies one or more collections in which to store the rows that the statement returns. For more information about this clause, see "**RETURNING INTO Clause**".

**Restriction on bulk_collect_into_clause**

Use if and only if *dynamic_sql_stmt* can return multiple rows.

**dynamic_returning_clause**

Returns the column values of the rows affected by the dynamic SQL statement, in either individual variables or records. For more information about this clause, see "**RETURNING INTO Clause**".

**Restriction on dynamic_returning_clause**

Use if and only if *dynamic_sql_stmt* has a RETURNING INTO clause.

**using_clause**

Specifies bind variables, using positional notation.
Restrictions on using_clause

- Use if and only if dynamic_sql_stmt includes placeholders for bind variables.
- If dynamic_sql_stmt has a RETURNING INTO clause (static_returning_clause), then using_clause can contain only IN bind variables. The bind variables in the RETURNING INTO clause are OUT bind variables by definition.

IN, OUT, IN OUT

Parameter modes of bind variables. An IN bind variable passes its value to dynamic_sql_stmt. An OUT bind variable stores a value that dynamic_sql_stmt returns. An IN OUT bind variable passes its initial value to dynamic_sql_stmt and stores a value that dynamic_sql_stmt returns. Default: IN.

For DML a statement with a RETURNING clause, you can place OUT bind variables in the RETURNING INTO clause without specifying the parameter mode, which is always OUT.

bind_argument

An expression whose value replaces its corresponding placeholder in dynamic_sql_stmt at run time.

Every placeholder in dynamic_sql_stmt must be associated with a bind_argument in the USING clause or RETURNING INTO clause (or both) or with a define variable in the INTO clause.

You can run dynamic_sql_stmt repeatedly using different values for the bind variables. You incur some overhead, because EXECUTE IMMEDIATE prepares the dynamic string before every execution.

Restrictions on bind_argument

- bind_argument cannot be an associative array indexed by string.
- bind_argument cannot be the reserved word NULL.

To pass the value NULL to the dynamic SQL statement, use an uninitialized variable where you want to use NULL, as in Example 8-7.
14.29 EXIT Statement

The **EXIT** statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the end of either the current loop or an enclosing labeled loop.

The **EXIT WHEN** statement exits the current iteration of a loop when the condition in its **WHEN** clause is true, and transfers control to the end of either the current loop or an enclosing labeled loop.

Each time control reaches the **EXIT WHEN** statement, the condition in its **WHEN** clause is evaluated. If the condition is not true, the **EXIT WHEN** statement does nothing. To prevent an infinite loop, a statement inside the loop must make the condition true.

**Restriction on EXIT Statement**

An **EXIT** statement must be inside a **LOOP** statement.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```plaintext
exit_statement ::= EXIT label WHEN boolean_expression;
```

(\( boolean_expression ::= \))
Semantics

**exit_statement**

**label**

Name that identifies either the current loop or an enclosing loop.

Without **label**, the **EXIT** statement transfers control to the end of the current loop. With **label**, the **EXIT** statement transfers control to the end of the loop that **label** identifies.

**WHEN boolean_expression**

Without this clause, the **EXIT** statement exits the current iteration of the loop unconditionally. With this clause, the **EXIT** statement exits the current iteration of the loop if and only if the value of **boolean_expression** is **TRUE**.

Examples

**Example 14-31  Basic LOOP Statement with EXIT Statement**

In this example, the **EXIT** statement inside the basic **LOOP** statement transfers control unconditionally to the end of the current loop.

```sql
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4

**Example 14-32  Basic LOOP Statement with EXIT WHEN Statement**

In this example, the **EXIT WHEN** statement inside the basic **LOOP** statement transfers control to the end of the current loop when **x** is greater than 3.

```sql
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;  -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
END;
/
```
END LOOP;
-- After EXIT statement, control resumes here
DBMS_OUTPUT.PUT_LINE('After loop:  x = ' || TO_CHAR(x));
END;
/

Result:
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4

Related Topics

• "Basic LOOP Statement"
• "CONTINUE Statement"

14.30 Explicit Cursor Declaration and Definition

An explicit cursor is a named pointer to a private SQL area that stores information for processing a specific query or DML statement—typically, one that returns or affects multiple rows.

You can use an explicit cursor to retrieve the rows of a result set one at a time.

Before using an explicit cursor, you must declare and define it. You can either declare it first (with cursor_declaration) and then define it later in the same block, subprogram, or package (with cursor_definition) or declare and define it at the same time (with cursor_definition).

An explicit cursor declaration and definition are also called a cursor specification and cursor body, respectively.

Note:

An explicit cursor declared in a package specification is affected by the AUTHID clause of the package. For more information, see "CREATE PACKAGE Statement".

Topics

• Syntax
• Semantics
• Examples
• Related Topics
Syntax

cursor_declaration ::= 

CURSOR cursor (cursor_parameter_dec, ) RETURN rowtype ;

cursor_definition ::= 

CURSOR cursor (cursor_parameter_dec, ) RETURN rowtype IS select_statement ;

cursor_parameter_dec ::= 

parameter_name IN datatype ::= DEFAULT expression

rowtype ::= 

db_table_or_view cursor cursor_variable % ROWTYPE record % TYPE record_type

Semantics

cursor_declaration

cursor

Name of the explicit cursor that you are declaring now and will define later in the same block, subprogram, or package. cursor can be any identifier except the reserved word SQL. Oracle recommends against giving a cursor the same name as a database table.
Explicit cursor names follow the same scoping rules as variables (see “Scope and Visibility of Identifiers”).

**cursor_definition**

Either defines an explicit cursor that was declared earlier or both declares and defines an explicit cursor.

**cursor**

Either the name of the explicit cursor that you previously declared and are now defining or the name of the explicit cursor that you are both declaring and defining. **cursor** can be any identifier except the reserved word SQL. Oracle recommends against giving a cursor the same name as a database table.

**select_statement**

A SQL SELECT statement (not a PL/SQL SELECT INTO statement). If the cursor has formal parameters, each parameter must appear in **select_statement**. The **select_statement** can also reference other PL/SQL variables in its scope.

**Restriction on select_statement**

This **select_statement** cannot have a WITH clause.

---

See:

 Oracle Database SQL Language Reference for SELECT statement syntax

---

**cursor_parameter_dec**

A cursor parameter declaration.

**parameter**

The name of the formal cursor parameter that you are declaring. This name can appear anywhere in **select_statement** that a constant can appear.

**IN**

Whether or not you specify **IN**, a formal cursor parameter has the characteristics of an **IN** subprogram parameter, which are summarized in **Table 9-1**. When the cursor opens, the value of the formal parameter is that of either its actual parameter or default value.

**datatype**

The data type of the parameter.

**Restriction on datatype**

This **datatype** cannot have constraints (for example, NOT NULL, or precision and scale for a number, or length for a string).

**expression**

Specifies the default value for the formal cursor parameter. The data types of **expression** and the formal cursor parameter must be compatible.
If an OPEN statement does not specify an actual parameter for the formal cursor parameter, then the statement evaluates expression and assigns its value to the formal cursor parameter.

If an OPEN statement does specify an actual parameter for the formal cursor parameter, then the statement assigns the value of the actual parameter to the formal cursor parameter and does not evaluate expression.

rowtype
Data type of the row that the cursor returns. The columns of this row must match the columns of the row that select_statement returns.

db_table_or_view
Name of a database table or view, which must be accessible when the declaration is elaborated.

cursor
Name of a previously declared explicit cursor.

cursor_variable
Name of a previously declared cursor variable.

record
Name of a previously declared record variable.

record_type
Name of a user-defined type that was defined with the data type specifier RECORD.

Examples
• Example 7-5, "Explicit Cursor Declaration and Definition"
• Example 7-8, "Variable in Explicit Cursor Query—No Result Set Change"
• Example 7-9, "Variable in Explicit Cursor Query—Result Set Change"
• Example 7-10, "Explicit Cursor with Virtual Column that Needs Alias"
• Example 7-11, "Explicit Cursor that Accepts Parameters"
• Example 7-12, "Cursor Parameters with Default Values"
• Example 7-13, "Adding Formal Parameter to Existing Cursor"
• Example 7-22, "Subquery in FROM Clause of Parent Query"
• Example 7-23, "Correlated Subquery"
• Example 7-35, "CURSOR Expression"
• Example 7-41, "FETCH with FOR UPDATE Cursor After COMMIT Statement"

Related Topics
In this chapter:
• "CLOSE Statement"
• "Cursor FOR LOOP Statement"
14.31 Expression

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function invocations, and placeholders) and operators.

The simplest expression is a single variable.

The PL/SQL compiler determines the data type of an expression from the types of the operands and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results.

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

expression ::=
(boolean_expression ::=, character_expression ::=, collection_constructor ::=, date_expression ::=, numeric_expression ::=, qualified_expression ::=, searched_case_expression ::=, simple_case_expression ::=)

boolean_expression ::= 

(function_call ::=)

boolean_literal ::= 

conditional_predicate ::= 

Chapter 14
Expression

14-110
other_boolean_form ::= 

(expression ::=, named_cursor ::=) 

collection_constructor ::= 

(function_call ::=, placeholder ::=)
\[
date_{expression} ::= \\
\text{date_constant} \quad \text{date_function_call} \quad \text{date_literal} \quad \text{date_variable} \quad \text{placeholder} \\
\quad + \quad \text{numeric_expression} \\
\quad - \\
\]

\[
\text{function_call} ::=, \quad \text{placeholder} ::= \\
\text{numeric_expression} ::= \\
\text{numeric_subexpression} + - \text{numeric_subexpression} \\
\text{numeric_subexpression} ::= \\
\text{collection} . \text{COUNT} \quad \text{FIRST} \quad \text{LAST} \quad \text{LIMIT} \quad \text{NEXT} \quad \text{PRIOR} (\text{index}) \\
\text{named_cursor} \% \text{ROWCOUNT} \\
\text{numeric_constant} \quad \text{numeric_function_call} \quad \text{numeric_literal} \quad \text{numeric_variable} \quad \text{placeholder} \\
\text{SQL} \% \text{ROWCOUNT} \quad \text{BULK_ROWCOUNT} (\text{index}) \\
\text{exponent} \\
\]

\[
\text{function_call} ::=, \quad \text{named_cursor} ::=, \quad \text{placeholder} ::= \\
\]
function_call ::=  

\[
\text{function}\left(\text{parameter}'\right)
\]

searched_case_expression ::=  

\[
\text{CASE}\quad \text{WHEN}\quad \text{boolean_expression}\quad \text{THEN}\quad \text{result}\quad \text{ELSE}\quad \text{result}\quad \text{END}
\]

\[(\text{boolean_expression} ::=)\]

simple_case_expression ::=  

\[
\text{CASE}\quad \text{selector}\quad \text{WHEN}\quad \text{selector_value}\quad \text{THEN}\quad \text{result}\quad \text{ELSE}\quad \text{result}\quad \text{END}
\]

Semantics

boolean_expression

Expression whose value is TRUE, FALSE, or NULL. For more information, see "BOOLEAN Expressions".

Restriction on boolean_expression

Because SQL has no data type equivalent to BOOLEAN, you cannot:

- Assign a BOOLEAN value to a database table column
- Select or fetch the value of a database table column into a BOOLEAN variable
- Use a BOOLEAN value in a SQL function
  (However, a SQL query can invoke a PL/SQL function that has a BOOLEAN parameter, as in Example 4-3.)
- Use a BOOLEAN expression in a SQL statement, except as an argument to a PL/SQL function invoked in a SQL query, or in a PL/SQL anonymous block.

Note:
An argument to a PL/SQL function invoked in a static SQL query cannot be a BOOLEAN literal. The workaround is to assign the literal to a variable and then pass the variable to the function, as in Example 4-3.
NOT, AND, OR
See "Logical Operators".

boolean_constant
Name of a constant of type BOOLEAN.

boolean_function_call
Invocation of a previously defined function that returns a BOOLEAN value. For more semantic information, see "function_call".

boolean_variable
Name of a variable of type BOOLEAN.

conditional_predicate
See "Conditional Predicates for Detecting Triggering DML Statement".

other_boolean_form

collection
Name of a collection variable.

EXISTS
Collection method (function) that returns TRUE if the indexth element of collection exists and FALSE otherwise. For more information, see "EXISTS Collection Method".

Restriction on EXISTS
You cannot use EXISTS if collection is an associative array.

index
Numeric expression whose data type either is PLS_INTEGER or can be implicitly converted to PLS_INTEGER (for information about the latter, see "Predefined PLS_INTEGER Subtypes").

IS [NOT] NULL
See "IS [NOT] NULL Operator".

BETWEEN expression AND expression
See "BETWEEN Operator".

IN expression [, expression ]...
See "IN Operator".

LIKE pattern
See "LIKE Operator".

relational_operator
See "Relational Operators".
SQL

Implicit cursor associated with the most recently run SELECT or DML statement. For more information, see "Implicit Cursors".

%FOUND, %ISOPEN, %NOTFOUND

Cursor attributes explained in "Implicit Cursor Attribute" and "Named Cursor Attribute".

character_expression

Expression whose value has a character data type (that is, a data type in the CHAR family, described in "CHAR Data Type Family").

character_constant

Name of a constant that has a character data type.

character_function_call

Invocation of a previously defined function that returns a value that either has a character data type or can be implicitly converted to a character data type. For more semantic information, see "function_call".

character_literal

Literal of a character data type.

character_variable

Name of a variable that has a character data type.

||

Concatenation operator, which appends one string operand to another. For more information, see "Concatenation Operator".

collection_constructor

Constructs a collection of the specified type with elements that have the specified values.

For more information, see "Collection Constructors".

collection_type

Name of a previously declared nested table type or VARRAY type (not an associative array type).

value

Valid value for an element of a collection of collection_type.

If collection_type is a varray type, then it has a maximum size, which the number of values cannot exceed. If collection_type is a nested table type, then it has no maximum size.

If you specify no values, then the constructed collection is empty but not null (for the difference between empty and null, see "Collection Types").
**date_expression**
Expression whose value has a date data type (that is, a data type in the DATE family, described in "DATE Data Type Family").

**date_constant**
Name of a constant that has a date data type.

**date_function_call**
Invocation of a previously defined function that returns a value that either has a date data type or can be implicitly converted to a date data type. For more semantic information, see "function_call".

**date_literal**
Literal whose value either has a date data type or can be implicitly converted to a date data type.

**date_variable**
Name of a variable that has a date data type.
+ , -
Addition and subtraction operators.

**numeric_expression**
Expression whose value has a date numeric type (that is, a data type in the DATE family, described in "NUMBER Data Type Family").
+ , - , I , * , **
Addition, subtraction, division, multiplication, and exponentiation operators.

**numeric_subexpression**

**collection**
Name of a collection variable.

**COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR**
Collection methods explained in "Collection Method Invocation".

**named_cursor%ROWCOUNT**
See "Named Cursor Attribute".

**numeric_constant**
Name of a constant that has a numeric data type.

**numeric_function_call**
Invocation of a previously defined function that returns a value that either has a numeric data type or can be implicitly converted to a numeric data type. For more semantic information, see "function_call".

**numeric_literal**
Literal of a numeric data type.

\textit{numeric\_variable}

Name of variable that has a numeric data type.

\textit{SQL\%ROWCOUNT}

Cursor attribute explained in "Implicit Cursor Attribute".

\textit{SQL\%BULK\_ROWCOUNT]}

Cursor attribute explained in "SQL\%BULK\_ROWCOUNT".

\textit{exponent}

Numeric expression.

\textit{function\_call}

\textit{function}

Name of a previously defined function.

\textit{parameter [, parameter]...}

List of actual parameters for the function being called. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter. The mode of the formal parameter determines what the actual parameter can be:

<table>
<thead>
<tr>
<th>Formal Parameter Mode</th>
<th>Actual Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Constant, initialized variable, literal, or expression</td>
</tr>
<tr>
<td>OUT</td>
<td>Variable whose data type is not defined as \textit{NOT NULL}</td>
</tr>
<tr>
<td>IN OUT</td>
<td>Variable (typically, it is a string buffer or numeric accumulator)</td>
</tr>
</tbody>
</table>

If the function specifies a default value for a parameter, you can omit that parameter from the parameter list. If the function has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

\textit{searched\_case\_expression}

**WHEN boolean\_expression THEN result**

The \textit{boolean\_expressions} are evaluated sequentially. If a \textit{boolean\_expression} has the value \textit{TRUE}, then the \textit{result} associated with that \textit{boolean\_expression} is returned. Subsequent \textit{boolean\_expressions} are not evaluated.

**ELSE result**

The \textit{result} is returned if and only if no \textit{boolean\_expression} has the value \textit{TRUE}.

\textit{See Also:}

"Positional, Named, and Mixed Notation for Actual Parameters"
If you omit the ELSE clause, the searched case expression returns NULL.

See Also:

"Searched CASE Statement"

**simple_case_expression**

**selector**

An expression of any PL/SQL type except BLOB, BFILE, or a user-defined type. The selector is evaluated once.

**WHEN selector_value THEN result**

The selector_values are evaluated sequentially. If a selector_value is the value of selector, then the result associated with that selector_value is returned. Subsequent selector_values are not evaluated.

A selector_value can be of any PL/SQL type except BLOB, BFILE, an ADT, a PL/SQL record, an associative array, a varray, or a nested table.

**ELSE result**

The result is returned if and only if no selector_value has the same value as selector.

If you omit the ELSE clause, the simple case expression returns NULL.

Note:

If you specify the literal NULL for every result (including the result in the ELSE clause), then error PLS-00617 occurs.

See Also:

"Simple CASE Statement"

Examples

- Example 3-28, "Concatenation Operator Examples"
- Example 3-30, "Controlling Evaluation Order with Parentheses"
- Example 3-31, "Expression with Nested Parentheses"
- Example 3-32, "Improving Readability with Parentheses"
- Example 3-33, "Operator Precedence"
- Example 3-43, "Relational Operators in Expressions"
- Example 3-44, "LIKE Operator in Expression"
14.32 FETCH Statement

The `FETCH` statement retrieves rows of data from the result set of a multiple-row query—one row at a time, several rows at a time, or all rows at once—and stores the data in variables, records, or collections.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
fetch_statement ::= 
```
(bulk_collect_into_clause ::=, into_clause ::=, numeric_expression ::=)

Semantics

fetch_statement
cursor
Name of an open explicit cursor. To open an explicit cursor, use the "OPEN Statement".

If you try to fetch from an explicit cursor before opening it or after closing it, PL/SQL raises the predefined exception INVALID_CURSOR.
cursor_variable
Name of an open cursor variable. To open a cursor variable, use the "OPEN FOR Statement". The cursor variable can be a formal subprogram parameter (see "Cursor Variables as Subprogram Parameters").

If you try to fetch from a cursor variable before opening it or after closing it, PL/SQL raises the predefined exception INVALID_CURSOR.
:host_cursor_variable
Name of a cursor variable declared in a PL/SQL host environment, passed to PL/SQL as a bind variable, and then opened. To open a host cursor variable, use the "OPEN FOR Statement". Do not put space between the colon (:) and host_cursor_variable.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

into_clause
To have the FETCH statement retrieve one row at a time, use this clause to specify the variables or record in which to store the column values of a row that the cursor returns. For more information about into_clause, see "into_clause ::=".

bulk_collect_into_clause [ LIMIT numeric_expression ]

Use bulk_collect_into_clause to specify one or more collections in which to store the rows that the FETCH statement returns. For more information about bulk_collect_into_clause, see "bulk_collect_into_clause ::=".

To have the FETCH statement retrieve all rows at once, omit LIMIT numeric_expression.

To limit the number of rows that the FETCH statement retrieves at once, specify LIMIT numeric_expression.

Restrictions on bulk_collect_into_clause
• You cannot use bulk_collect_into_clause in client programs.
• When the FETCH statement requires implicit data type conversions, bulk_collect_into_clause can have only one collection or host_array.

Examples
• Example 6-54, "FETCH Assigns Values to Record that Function Returns"
14.33 FOR LOOP Statement

With each iteration of the FOR LOOP statement, its statements run, its index is either incremented or decremented, and control returns to the top of the loop.

The FOR LOOP statement ends when its index reaches a specified value, or when a statement inside the loop transfers control outside the loop or raises an exception. An index is also called an iterand. Statements outside the loop cannot reference the iterand. After the FOR LOOP statement runs, the iterand is undefined.
Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

for_loop_statement ::= 

ITERATOR, statement ::= 

(Iterator, statement ::=)

Semantics

for_loop_statement

iterator

See iterator

statement

An EXIT, EXIT WHEN, CONTINUE, or CONTINUE WHEN in the statements can cause the loop or the current iteration of the loop to end early. See "statement ::=" for the list of all possible statements.

label

A label that identifies for_loop_statement (see "label"). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label in the END LOOP statement matches a label at the beginning of the same LOOP statement (the compiler does not check).

Examples

• Example 5-19, "Simple Step Filter Using FOR LOOP Stepped Range Iterator"
• Example 5-15, "FOR LOOP Statement Range Iteration Control"
• Example 5-16, "Reverse FOR LOOP Statement Range Iteration Control"
• Example 5-27, "Using FOR LOOP Stopping Predicate Clause"
• Example 5-28, "Using FOR LOOP Skipping Predicate Clause"
• Example 5-10, "Outside Statement References FOR LOOP Statement Index"
Example 14-33  EXIT WHEN Statement in FOR LOOP Statement

Suppose that you must exit a FOR LOOP statement immediately if a certain condition arises. You can put the condition in an EXIT WHEN statement inside the FOR LOOP statement.

In this example, the FOR LOOP statement executes 10 times unless the FETCH statement inside it fails to return a row, in which case it ends immediately.

```sql
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;
  -- Fetch entire row into v_employees record:
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_employees;
    EXIT WHEN c1%NOTFOUND;
    -- Process data here
    END LOOP;
  CLOSE c1;
END;
/
```

Example 14-34  EXIT WHEN Statement in Inner FOR LOOP Statement

Now suppose that the FOR LOOP statement that you must exit early is nested inside another FOR LOOP statement. If, when you exit the inner loop early, you also want to exit the outer loop, then label the outer loop and specify its name in the EXIT WHEN statement.

```sql
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      EXIT outer_loop WHEN c1%NOTFOUND;
      -- Process data here
      END LOOP;
    END LOOP outer_loop;

    CLOSE c1;
  END;
END;
/
```
Example 14-35  CONTINUE WHEN Statement in Inner FOR LOOP Statement

If you want to exit the inner loop early but complete the current iteration of the outer loop, then label the outer loop and specify its name in the CONTINUE WHEN statement.

```sql
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      CONTINUE outer_loop WHEN c1%NOTFOUND;
        -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

Related Topics

- "FOR LOOP Iterand"
- "FOR LOOP Statement Overview" for more conceptual information
- "Basic LOOP Statement"
- "CONTINUE Statement"
- "Cursor FOR LOOP Statement"
- "EXIT Statement"
- "FETCH Statement"
- "FOREALL Statement"
- "OPEN Statement"
- "WHILE LOOP Statement"
- "Overview of Exception Handling" for information about exceptions, which can also cause a loop to end immediately if a certain condition arises

14.34 FORALL Statement

The FORALL statement runs one DML statement multiple times, with different values in the VALUES and WHERE clauses.

The different values come from existing, populated collections or host arrays. The FORALL statement is usually much faster than an equivalent FOR LOOP statement.
Note:
You can use the FORALL statement only in server programs, not in client programs.

Topics
• Syntax
• Semantics
• Examples
• Related Topics

Syntax
forall_statement ::= 

FORALL index IN bounds_clause
SAVE EXCEPTIONS
dml_statement ;

bounds_clause ::= 

lower_bound .. upper_bound
INDICES OF collection BETWEEN lower_bound AND upper_bound
VALUES OF index_collection

Semantics
forall_statement

index
Name for the implicitly declared integer variable that is local to the FORALL statement. Statements outside the FORALL statement cannot reference index. Statements inside the FORALL statement can reference index as an index variable, but cannot use it in expressions or change its value. After the FORALL statement runs, index is undefined.

dml_statement
A static or dynamic INSERT, UPDATE, DELETE, or MERGE statement that references at least one collection in its VALUES or WHERE clause. Performance benefits apply only to collection references that use index as an index.

Every collection that dml_statement references must have indexes that match the values of index. If you apply the DELETE, EXTEND, or TRIM method to one collection, apply it to the other collections also, so that all collections have the same set of indexes. If any collection lacks a referenced element, PL/SQL raises an exception.
Restriction on \textit{dml\_statement}

If \textit{dml\_statement} is a dynamic SQL statement, then values in the \texttt{USING} clause (bind variables for the dynamic SQL statement) must be simple references to the collection, not expressions. For example, \texttt{collection(i)} is valid, but \texttt{UPPER(collection(i))} is invalid.

SAVE EXCEPTIONS

Lets the \texttt{FORALL} statement continue even if some of its DML statements fail. For more information, see "Handling \texttt{FORALL} Exceptions After \texttt{FORALL} Statement Completes".

\textit{bounds\_clause}

Specifies the collection element indexes that provide values for the variable \texttt{index}. For each value, the SQL engine runs \textit{dml\_statement} once.

\textit{lower\_bound} .. \textit{upper\_bound}

Both \texttt{lower\_bound} and \texttt{upper\_bound} are numeric expressions that PL/SQL evaluates once, when the \texttt{FORALL} statement is entered, and rounds to the nearest integer if necessary. The resulting integers must be the lower and upper bounds of a valid range of consecutive index numbers. If an element in the range is missing or was deleted, PL/SQL raises an exception.

\textbf{INDICES OF} \textit{collection} \texttt{[ BETWEEN lower\_bound AND upper\_bound ]}

Specifies that the values of \texttt{index} correspond to the indexes of the elements of \textit{collection}. The indexes need not be consecutive.

Both \texttt{lower\_bound} and \texttt{upper\_bound} are numeric expressions that PL/SQL evaluates once, when the \texttt{FORALL} statement is entered, and rounds to the nearest integer if necessary. The resulting integers are the lower and upper bounds of a valid range of index numbers, which need not be consecutive.

Restriction on \textit{collection}

If \textit{collection} is an associative array, it must be indexed by \texttt{PLS\_INTEGER}.

\textbf{VALUES OF} \textit{index\_collection}

Specifies that the values of \texttt{index} are the elements of \textit{index\_collection}, a collection of \texttt{PLS\_INTEGER} elements that is indexed by \texttt{PLS\_INTEGER}. The indexes of \textit{index\_collection} need not be consecutive. If \textit{index\_collection} is empty, PL/SQL raises an exception and the \texttt{FORALL} statement does not run.

Examples

• \texttt{Example 13-8}, "DELETE Statement in \texttt{FORALL} Statement"
• \texttt{Example 13-9}, "Time Difference for INSERT Statement in \texttt{FOR LOOP} and \texttt{FORALL} Statements"
• \texttt{Example 13-10}, "\texttt{FORALL} Statement for Subset of Collection"
• \texttt{Example 13-11}, "\texttt{FORALL} Statements for Sparse Collection and Its Subsets"
• \texttt{Example 13-12}, "Handling \texttt{FORALL} Exceptions Immediately"
• \texttt{Example 13-13}, "Handling \texttt{FORALL} Exceptions After \texttt{FORALL} Statement Completes"
14.35 Formal Parameter Declaration

A formal parameter declaration specifies the name and data type of the parameter, and (optionally) its mode and default value.

A formal parameter declaration can appear in the following:

- "Function Declaration and Definition"
- "Procedure Declaration and Definition"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
parameter_declaration ::= ...
```

Semantics

**parameter_declaration**

**parameter**

Name of the formal parameter that you are declaring, which you can reference in the executable part of the subprogram.

**IN, OUT, IN OUT**

Mode that determines the behavior of the parameter, explained in "Subprogram Parameter Modes". Default: IN.

**Note:**

Avoid using OUT and IN OUT for function parameters. The purpose of a function is to take zero or more parameters and return a single value. Functions must be free from side effects, which change the values of variables not local to the subprogram.

**NOCOPY**

Requests that the compiler pass the corresponding actual parameter by reference instead of value (for the difference, see "Subprogram Parameter Passing Methods"). Each time the subprogram is invoked, the optimizer decides, silently, whether to obey or disregard NOCOPY.

**Caution:**

NOCOPY increases the likelihood of aliasing. For details, see "Subprogram Parameter Aliasing with Parameters Passed by Reference".

The compiler ignores NOCOPY in these cases:

- The actual parameter must be implicitly converted to the data type of the formal parameter.
- The actual parameter is the element of a collection.
- The actual parameter is a scalar variable with the NOT NULL constraint.
- The actual parameter is a scalar numeric variable with a range, size, scale, or precision constraint.
- The actual and formal parameters are records, one or both was declared with %ROWTYPE or %TYPE, and constraints on corresponding fields differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor FOR LOOP statement, and constraints on corresponding fields differ.
- The subprogram is invoked through a database link or as an external subprogram.
**Note:**
The preceding list might change in a subsequent release.

**datatype**
Data type of the formal parameter that you are declaring. The data type can be a constrained subtype, but cannot include a constraint (for example, `NUMBER(2)` or `VARCHAR2(20)`).

If `datatype` is a constrained subtype, the corresponding actual parameter inherits the `NOT NULL` constraint of the subtype (if it has one), but not the size (see Example 9-10).

**Caution:**
The data type `REF CURSOR` increases the likelihood of subprogram parameter aliasing, which can have unintended results. For more information, see "Subprogram Parameter Aliasing with Cursor Variable Parameters".

**expression**
Default value of the formal parameter that you are declaring. The data type of `expression` must be compatible with `datatype`.

If a subprogram invocation does not specify an actual parameter for the formal parameter, then that invocation evaluates `expression` and assigns its value to the formal parameter.

If a subprogram invocation does specify an actual parameter for the formal parameter, then that invocation assigns the value of the actual parameter to the formal parameter and does not evaluate `expression`.

Examples

- Example 3-26, "Assigning Value to Variable as IN OUT Subprogram Parameter"
- Example 9-9, "Formal Parameters and Actual Parameters"
- Example 9-14, "Parameter Values Before, During, and After Procedure Invocation"
- Example 9-15, "OUT and IN OUT Parameter Values After Exception Handling"
- Example 9-20, "Procedure with Default Parameter Values"
- Example 9-21, "Function Provides Default Parameter Value"
- Example 9-22, "Adding Subprogram Parameter Without Changing Existing Invocations"

Related Topics

In this chapter:

- "Function Declaration and Definition"
- "Procedure Declaration and Definition"
Before invoking a function, you must declare and define it. You can either declare it first (with `function_declaration`) and then define it later in the same block, subprogram, or package (with `function_definition`) or declare and define it at the same time (with `function_definition`).

A **function** is a subprogram that returns a value. The data type of the value is the data type of the function. A function invocation (or call) is an expression, whose data type is that of the function.

A function declaration is also called a **function specification** or **function spec**.

**Note:**

This topic applies to nested functions.

For information about standalone functions, see "CREATE FUNCTION Statement".

For information about package functions, see "CREATE PACKAGE Statement".

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```
function_declaration ::= (function_heading
DETERMINISTIC
PIPELINED
PARALLEL_ENABLE
RESULT_CACHE
);
```
**function_heading ::=**

```
FUNCTION function_name (parameter_declaration) RETURN datatype
```

(datatype ::= , parameter_declaration ::= )

**function_definition ::=**

```
function_heading
DETERMINISTIC
PIPELINED
PARALLEL_ENABLE
RESULT_CACHE
relies_on_clause
IS
AS
declare_section
body
call_spec
```

(body ::= , declare_section ::= , pipelined_clause ::= , deterministic_clause ::= , parallel_enable_clause ::= , result_cache_clause ::= , call_spec ::= )

**Semantics**

**function_declaration**

Declares a function, but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.

**function_heading**

The function heading specifies the function name and its parameter list.

**function_name**

Name of the function that you are declaring or defining.

**RETURN datatype**

Specifies the data type of the value that the function returns, which can be any PL/SQL data type (see PL/SQL Data Types).

**Restriction on datatype**

You cannot constrain this data type (with \texttt{NOT NULL}, for example). If \texttt{datatype} is a constrained subtype, then the returned value does not inherit the constraints of the subtype (see "Formal Parameters of Constrained Subtypes").
function_definition

Either defines a function that was declared earlier or both declares and defines a function.

declare_section

Declares items that are local to the function, can be referenced in body, and cease to exist when the function completes execution.

body

Required executable part and optional exception-handling part of the function. In the executable part, at least one execution path must lead to a RETURN statement; otherwise, a runtime error occurs.

Examples

• Example 9-2, “Declaring, Defining, and Invoking a Simple PL/SQL Function”

Related Topics

• Formal Parameter Declaration
• Procedure Declaration and Definition
• PL/SQL Subprograms

14.37 GOTO Statement

The GOTO statement transfers control to a labeled block or statement.

If a GOTO statement exits a cursor FOR LOOP statement prematurely, the cursor closes.

Restrictions on GOTO Statement

• A GOTO statement cannot transfer control into an IF statement, CASE statement, LOOP statement, or sub-block.

• A GOTO statement cannot transfer control from one IF statement clause to another, or from one CASE statement WHEN clause to another.

• A GOTO statement cannot transfer control out of a subprogram.

• A GOTO statement cannot transfer control into an exception handler.

• A GOTO statement cannot transfer control from an exception handler back into the current block (but it can transfer control from an exception handler into an enclosing block).

Topics

• Syntax
• Semantics
• Examples
• Related Topics
Syntax

goto_statement ::= 

Semantics

goto_statement

label

Identifies either a block or a statement (see "plsql_block ::=", "statement ::=", and "label").

If label is not in the current block, then the GOTO statement transfers control to the first enclosing block in which label appears.

Examples

Example 14-36  GOTO Statement

A label can appear before a statement.

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;
  p := ' is a prime number';

  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/ 
```

Result:

37 is a prime number

Example 14-37  Incorrect Label Placement

A label can only appear before a block or before a statement.

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
```
<<end_loop>>
END LOOP;
END;
/

Result:

END LOOP;
*
ERROR at line 9:
ORA-06550: line 9, column 3:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( begin case declare exit for goto if loop mod null raise
return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql run commit forall merge pipe purge

Example 14-38  GOTO Statement Goes to Labeled NULL Statement

A label can appear before a NULL statement.

DECLARE
   done  BOOLEAN;
BEGIN
   FOR i IN 1..50 LOOP
      IF done THEN
         GOTO end_loop;
      END IF;
      <<end_loop>>
      NULL;
   END LOOP;
END;
/

Example 14-39  GOTO Statement Transfers Control to Enclosing Block

A GOTO statement can transfer control to an enclosing block from the current block.

DECLARE
   v_last_name  VARCHAR2(25);
   v_emp_id     NUMBER(6) := 120;
BEGIN
   <<get_name>>
   SELECT last_name INTO v_last_name
   FROM employees
   WHERE employee_id = v_emp_id;
BEGIN
   DBMS_OUTPUT.PUT_LINE (v_last_name);
   v_emp_id := v_emp_id + 5;
   IF v_emp_id < 120 THEN
      GOTO get_name;
   END IF;
END;
END;
/

Result:
Example 14-40  GOTO Statement Cannot Transfer Control into IF Statement

The GOTO statement transfers control into an IF statement, causing an error.

DECLARE
   valid BOOLEAN := TRUE;
BEGIN
   GOTO update_row;
   IF valid THEN
      NULL;
   END IF;
END;
/

Result:
     GOTO update_row;
     *
     ERROR at line 4:
     ORA-06550: line 4, column 3:
     PLS-00375: illegal GOTO statement; this GOTO cannot transfer control to label 'UPDATE_ROW'
     ORA-06550: line 6, column 12:
     PL/SQL: Statement ignored

Related Topics
- "Block"
- "GOTO Statement"

14.38 IF Statement

The IF statement either runs or skips a sequence of one or more statements, depending on the value of a BOOLEAN expression.

Topics
- Syntax
- Semantics
- Examples
- Related Topics
Syntax

\[ \text{if\_statement ::= IF boolean\_expression THEN statement} \]

\[ \text{ELSIF boolean\_expression THEN statement} \]

\[ \text{ELSE statement END IF ;} \]

Semantics

**boolean\_expression**

Expression whose value is TRUE, FALSE, or NULL.

The first boolean\_expression is always evaluated. Each other boolean\_expression is evaluated only if the values of the preceding expressions are FALSE.

If a boolean\_expression is evaluated and its value is TRUE, the statements after the corresponding THEN run. The succeeding expressions are not evaluated, and the statements associated with them do not run.

**ELSE**

If no boolean\_expression has the value TRUE, the statements after ELSE run.

Examples

- Example 5-1, "IF THEN Statement"
- Example 5-2, "IF THEN ELSE Statement"
- Example 5-3, "Nested IF THEN ELSE Statements"
- Example 5-4, "IF THEN ELSIF Statement"

Related Topics

In this chapter:

- "CASE Statement"
- "Expression"

In other chapters:

- "Conditional Selection Statements"
14.39 Implicit Cursor Attribute

An implicit cursor has attributes that return information about the most recently run `SELECT` or DML statement that is not associated with a named cursor.

**Note:**
You can use cursor attributes only in procedural statements, not in SQL statements.

**Topics**
- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```
implicit_cursor_attribute ::= SQL %
```

**Semantics**

**%ISOPEN**

`SQL%ISOPEN` always has the value `FALSE`.

**%FOUND**

`SQL%FOUND` has one of these values:
- If no `SELECT` or DML statement has run, `NULL`.
- If the most recent `SELECT` or DML statement returned a row, `TRUE`.
- If the most recent `SELECT` or DML statement did not return a row, `FALSE`. 
%NOTFOUND

SQL%NOTFOUND has one of these values:

- If no SELECT or DML statement has run, NULL.
- If the most recent SELECT or DML statement returned a row, FALSE.
- If the most recent SELECT or DML statement did not return a row, TRUE.

%ROWCOUNT

SQL%ROWCOUNT has one of these values:

- If no SELECT or DML statement has run, NULL.
- If a SELECT or DML statement has run, the number of rows fetched so far.

SQL%BULK_ROWCOUNT

Composite attribute that is like an associative array whose \( i \)th element is the number of rows affected by the \( i \)th DML statement in the most recently completed FORALL statement. For more information, see "Getting Number of Rows Affected by FORALL Statement".

Restriction on SQL%BULK_ROWCOUNT

You cannot assign the value of SQL%BULK_ROWCOUNT\((\text{index})\) to another collection.

SQL%BULK_EXCEPTIONS

Composite attribute that is like an associative array of information about the DML statements that failed during the most recently run FORALL statement. SQL%BULK_EXCEPTIONS.COUNT is the number of DML statements that failed. If SQL%BULK_EXCEPTIONS.COUNT is not zero, then for each index value \( i \) from 1 through SQL%BULK_EXCEPTIONS.COUNT:

- SQL%BULK_EXCEPTIONS\((i)\).ERROR_INDEX is the number of the DML statement that failed.
- SQL%BULK_EXCEPTIONS\((i)\).ERROR_CODE is the Oracle Database error code for the failure.

Typically, this attribute appears in an exception handler for a FORALL statement that has a SAVE EXCEPTIONS clause. For more information, see "Handling FORALL Exceptions After FORALL Statement Completes".

Examples

- Example 7-3, "SQL%FOUND Implicit Cursor Attribute"
- Example 7-4, "SQL%ROWCOUNT Implicit Cursor Attribute"
- Example 7-15, "%FOUND Explicit Cursor Attribute"
- Example 7-14, "%ISOPEN Explicit Cursor Attribute"
- Example 7-16, "%NOTFOUND Explicit Cursor Attribute"
- Example 7-17, "%ROWCOUNT Explicit Cursor Attribute"
Example 13-13, "Handling FORALL Exceptions After FORALL Statement Completes"

Example 13-14, "Showing Number of Rows Affected by Each DELETE in FORALL"

Example 13-15, "Showing Number of Rows Affected by Each INSERT SELECT in FORALL"

Related Topics

In this chapter:

"FORALL Statement"
"Named Cursor Attribute"

In other chapters:

"Implicit Cursors"
"Processing Query Result Sets"

14.40 INLINEPragma

The `INLINE` pragma specifies whether a subprogram invocation is to be inlined.

Inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit).

Note:

The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements. For details, see "Subprogram Inlining".

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

```
inline pragma ::= PRAGMA INLINE ( subprogram , 'YES' | 'NO' )
```

```
PRAGMA INLINE subprogram ;
```
Semantics

*subprogram*

Name of a subprogram. If *subprogram* is overloaded, then the INLINE pragma applies to every subprogram with that name.

**YES**

If `PLSQL_OPTIMIZE_LEVEL=2`, 'YES' specifies that the subprogram invocation is to be inlined.

If `PLSQL_OPTIMIZE_LEVEL=3`, 'YES' specifies that the subprogram invocation has a high priority for inlining.

**NO**

Specifies that the subprogram invocation is not to be inlined.

Examples

- Example 13-1, "Specifying that Subprogram Is To Be Inlined"
- Example 13-2, "Specifying that Overloaded Subprogram Is To Be Inlined"
- Example 13-3, "Specifying that Subprogram Is Not To Be Inlined"
- Example 13-4, "PRAGMA INLINE ... 'NO' Overrides PRAGMA INLINE ... 'YES'"

Related Topics

- "Subprogram Inlining"

### 14.41 Invoker’s Rights and Definer’s Rights Clause

Specifies the *AUTHID* property of a stored PL/SQL subprogram. The *AUTHID* property affects the name resolution and privilege checking of SQL statements that the unit issues at run time.

The *invoker_rights_clause* can appear in the following SQL statements:

- ALTER TYPE Statement
- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PROCEDURE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

Topics

- Syntax
- Semantics
- Related Topics
Syntax

\[ \text{invoker_rights_clause ::= } \]

Semantics

\textit{invoker_rights_clause} \\
When it appears in the package declaration, it specifies the \texttt{AUTHID} property of functions and procedures in the package, and of the explicit cursors declared in the package specification.

When it appears in a standalone function declaration, it specifies the \texttt{AUTHID} property of the function.

When it appears in a standalone procedure declaration, it specifies the \texttt{AUTHID} property of the procedure.

The \textit{invoker_rights_clause} can appear only once in a subprogram declaration.

When it appears in an ADT, it specifies the \texttt{AUTHID} property of the member functions and procedures of the ADT.

Restrictions on \textit{invoker_rights_clause}

The following restrictions apply for types:

- This clause is valid only for ADTs, not for a nested table or \texttt{VARRAY} type.
- You can specify this clause for clarity if you are creating a subtype. However, a subtype inherits the \texttt{AUTHID} property of its supertype, so you cannot specify a different value than was specified for the supertype.
- If the supertype was created with \texttt{AUTHID DEFINER}, then you must create the subtype in the same schema as the supertype.
- You cannot specify the \texttt{AUTHID} property of SQL macros. They behave like IR units.

Related Topics

In this book:

- "Invoker's Rights and Definer's Rights (AUTHID Property)" for information about the \texttt{AUTHID} property
- "Subprogram Properties"
The PL/SQL extension to the SQL `INSERT` statement lets you specify a record name in the `values_clause` of the `single_table_insert` instead of specifying a column list in the `insert_into_clause`.

Effectively, this form of the `INSERT` statement inserts the record into the table; actually, it adds a row to the table and gives each column of the row the value of the corresponding record field.

**See Also:**

*Oracle Database SQL Language Reference* for the syntax of the SQL `INSERT` statement.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

\[
\text{insert_into_clause} ::= \\
\text{VALUES} \text{record}
\]

**Semantics**

\[
\text{insert_into_clause} \\
\text{dml_table_expression_clause} \\
\text{t_alias}
\]

Typically a table name. For complete information, see *Oracle Database SQL Language Reference*.

\[
\text{t_alias}
\]

An alias for `dml_table_expression_clause`. 

The `values_clause` section explains the definition of a `record` object. A record variable of type `RECORD` or `%ROWTYPE` must represent a row of the item explained by the `dml_table_expression_clause`. Each column in the row must have a field with a compatible data type. If a column has a `NOT NULL` constraint, the corresponding field cannot have a `NULL` value.

### See Also:

*Oracle Database SQL Language Reference* for the complete syntax of the `INSERT` statement.

#### Examples

- Example 6-57, "Initializing Table by Inserting Record of Default Values"

#### Related Topics

**In this chapter:**

- "Record Variable Declaration"
- "%ROWTYPE Attribute"

**In other chapters:**

- "Inserting Records into Tables"
- "Restrictions on Record Inserts and Updates"

### 14.44 Named Cursor Attribute

Every named cursor (explicit cursor or cursor variable) has four attributes, each of which returns information about the execution of a DML statement.

#### Note:

You can use cursor attributes only in procedural statements, not in SQL statements.

#### Topics

- Syntax
- Semantics
- Examples
- Related Topics
Syntax

\[ \text{named\_cursor\_attribute} ::= \]

```
| named\_cursor %
| ISOPEN
| FOUND
| NOTFOUND
| ROWCOUNT
```

\[ \text{named\_cursor} ::= \]

```
| explicit\_cursor
| cursor\_parameter
| cursor\_variable
| : host\_cursor\_variable
```

Semantics

\[ \text{named\_cursor\_attribute} \]

\%ISOPEN

\[ \text{named\_cursor}\%ISOPEN \text{ has the value } \text{TRUE} \text{ if the cursor is open, and } \text{FALSE} \text{ if it is not open.} \]

\%FOUND

\[ \text{named\_cursor}\%FOUND \text{ has one of these values:} \]

- If the cursor is not open, INVALID\_CURSOR.
- If cursor is open but no fetch was tried, NULL.
- If the most recent fetch returned a row, TRUE.
- If the most recent fetch did not return a row, FALSE.

\%NOTFOUND

\[ \text{named\_cursor}\%NOTFOUND \text{ has one of these values:} \]

- If cursor is not open, INVALID\_CURSOR.
- If cursor is open but no fetch was tried, NULL.
- If the most recent fetch returned a row, FALSE.
- If the most recent fetch did not return a row, TRUE.

\%ROWCOUNT

\[ \text{named\_cursor}\%ROWCOUNT \text{ has one of these values:} \]
• If cursor is not open, INVALID_CURSOR.
• If cursor is open, the number of rows fetched so far.

**named_cursor**

**explicit_cursor**

Name of an explicit cursor.

**cursor_parameter**

Name of a formal cursor parameter.

**cursor_variable**

Name of a cursor variable.

**:host_cursor_variable**

Name of a cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and `host_cursor_variable`.

**Examples**

• **Example 7-14**, "%ISOPEN Explicit Cursor Attribute"
• **Example 7-15**, "%FOUND Explicit Cursor Attribute"
• **Example 7-16**, "%NOTFOUND Explicit Cursor Attribute"
• **Example 7-17**, "%ROWCOUNT Explicit Cursor Attribute"

**Related Topics**

In this chapter:

• "Cursor Variable Declaration"
• "Explicit Cursor Declaration and Definition"
• "Implicit Cursor Attribute"

In other chapters:

• "Explicit Cursor Attributes"

**14.45 NULL Statement**

The NULL statement is a "no-op" (no operation)—it only passes control to the next statement.

**Note:**

The NULL statement and the BOOLEAN value NULL are not related.
14.46 OPEN Statement

The `OPEN` statement opens an explicit cursor, allocates database resources to process the associated query, identifies the result set, and positions the cursor before the first row of the result set.

If the query has a `FOR UPDATE` clause, the `OPEN` statement locks the rows of the result set.
Semantics

cursor

Name of an explicit cursor that is not open.

actual_cursor_parameter

List of actual parameters for the cursor that you are opening. An actual parameter can be a constant, initialized variable, literal, or expression. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter.

You can specify actual cursor parameters with either positional notation or named notation. For information about these notations, see "Positional, Named, and Mixed Notation for Actual Parameters".

If the cursor specifies a default value for a parameter, you can omit that parameter from the parameter list. If the cursor has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

Examples

• Example 7-11, "Explicit Cursor that Accepts Parameters"
• Example 7-12, "Cursor Parameters with Default Values"

Related Topics

In this chapter:

• "CLOSE Statement"
• "Explicit Cursor Declaration and Definition"
• "FETCH Statement"
• "OPEN FOR Statement"

In other chapters:

• "Opening and Closing Explicit Cursors"
• "Explicit Cursors that Accept Parameters"

14.47 OPEN FOR Statement

The OPEN FOR statement associates a cursor variable with a query, allocates database resources to process the query, identifies the result set, and positions the cursor before the first row of the result set.

If the query has a FOR UPDATE clause, then the OPEN FOR statement locks the rows of the result set.

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

`open_for_statement ::=`

```
OPEN
<cursor_variable>
:host_cursor_variable
FOR
<select_statement>
<dynamic_string>
<using_clause>
;
```

`using_clause ::=`

```
USING
<IN/OUT/bind_argument>
```

Semantics

`open_for_statement`

`<cursor_variable>`

Name of a cursor variable. If `<cursor_variable>` is the formal parameter of a subprogram, then it must not have a return type. For information about cursor variables as subprogram parameters, see “Cursor Variables as Subprogram Parameters”.

`:<host_cursor_variable>`

Name of a cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and `<host_cursor_variable>`.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

`<select_statement>`

SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement). Typically, `<select_statement>` returns multiple rows.

---

**See:**

*Oracle Database SQL Language Reference* for `SELECT` statement syntax
**dynamic_string**

String literal, string variable, or string expression of the data type CHAR, VARCHAR2, or CLOB, which represents a SQL SELECT statement. Typically, `dynamic_statement` represents a SQL SELECT statement that returns multiple rows.

**using_clause**

Specifies bind variables, using positional notation.

> **Note:**
>
> If you repeat placeholder names in `dynamic_sql_statement`, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement. For details, see "Repeated Placeholder Names in Dynamic SQL Statements."

**Restriction on using_clause**

Use if and only if `select_statement` or `dynamic_sql_stmt` includes placeholders for bind variables.

**IN, OUT, IN OUT**

Parameter modes of bind variables. An IN bind variable passes its value to the `select_statement` or `dynamic_string`. An OUT bind variable stores a value that `dynamic_string` returns. An IN OUT bind variable passes its initial value to `dynamic_string` and stores a value that `dynamic_string` returns. Default: IN.

**bind_argument**

Expression whose value replaces its corresponding placeholder in `select_statement` or `dynamic_string` at run time. You must specify a `bind_argument` for every placeholder.

> **Note:**
>
> Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.

**Restrictions on bind_argument**

- `bind_argument` cannot be an associative array indexed by string.
- `bind_argument` cannot be the reserved word NULL.

To pass the value NULL to the dynamic SQL statement, use an uninitialized variable where you want to use NULL, as in Example 8-7.

**Examples**

- Example 7-26, "Fetching Data with Cursor Variables"
• Example 7-30, "Querying a Collection with Static SQL"
• Example 7-31, "Procedure to Open Cursor Variable for One Query"
• Example 7-32, "Opening Cursor Variable for Chosen Query (Same Return Type)"
• Example 7-33, "Opening Cursor Variable for Chosen Query (Different Return Types)"
• Example 8-8, "Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements"
• Example 8-9, "Querying a Collection with Native Dynamic SQL"

Related Topics
In this chapter:
• "CLOSE Statement"
• "Cursor Variable Declaration"
• "EXECUTE IMMEDIATE Statement"
• "FETCH Statement"
• "OPEN Statement"
In other chapters:
• "Opening and Closing Cursor Variables"
• "OPEN FOR, FETCH, and CLOSE Statements"

14.48 PARALLEL_ENABLE Clause

Enables the function for parallel execution, making it safe for use in slave sessions of parallel DML evaluations.

Indicates that the function can run from a parallel execution server of a parallel query operation.

The PARALLEL_ENABLE clause can appear in the following SQL statements:

• CREATE FUNCTION Statement
• CREATE PACKAGE Statement
• CREATE TYPE BODY Statement

Topics
• Syntax
• Semantics
• Related Topics
Syntax

\[ \text{parallel\_enable\_clause ::= PARALLEL\_ENABLE ( PARTITION \text{argument} BY ANY HASH RANGE ( \text{column} , ) streaming\_clause VALUE ( \text{column} ) ) } \]

\[ \text{streaming\_clause ::= ORDER CLUSTER expr BY ( \text{column} , ) } \]

Semantics

**parallel\_enable\_clause**

The **parallel\_enable\_clause** can appear only once in the function.

The function must not use session state, such as package variables, because those variables are not necessarily shared among the parallel execution servers.

Use the optional **PARTITION \text{argument} BY** clause only with a function that has a **REF CURSOR** data type. This clause lets you define the partitioning of the inputs to the function from the **REF CURSOR** argument. Partitioning the inputs to the function affects the way the query is parallelized when the function is used as a table function in the **FROM clause** of the query.

**ANY**

Indicates that the data can be partitioned randomly among the parallel execution servers

**Note:**

You can partition weak cursor variable arguments to table functions only with **ANY**, not with **RANGE**, **HASH**, or **VALUE**.

**RANGE or HASH**
Partitions data into specified columns that are returned by the REF CURSOR argument of the function.

**streaming_clause**

The optional streaming_clause lets you order or cluster the parallel processing.

**ORDER BY | CLUSTER BY**

ORDER BY or CLUSTER BY indicates that the rows on a parallel execution server must be locally ordered and have the same key values as specified by the column list.

**VALUE**

Specifies direct-key partitioning, which is intended for table functions used when executing MapReduce workloads. The column must be of data type NUMBER. VALUE distributes row processing uniformly over the available reducers.

If the column has more reducer numbers than there are available reducers, then PL/SQL uses a modulus operation to map the reducer numbers in the column into the correct range.

When calculating the number of the reducer to process the corresponding row, PL/SQL treats a negative value as zero and rounds a positive fractional value to the nearest integer.

---

**See Also:**

*Oracle Database Data Cartridge Developer's Guide* for information about using parallel table functions

---

**expr**

expr identifies the REF CURSOR parameter name of the table function on which partitioning was specified, and on whose columns you are specifying ordering or clustering for each slave in a parallel query execution.

**Restriction on parallel_enable_clause**

You cannot specify the parallel_enable_clause for a nested function or SQL macro.

**Related Topics**

In this chapter:

- Function Declaration and Definition

In other chapters:

- Overview of Table Functions
- Creating Pipelined Table Functions
14.49 PIPE ROW Statement

The PIPE ROW statement, which can appear only in the body of a pipelined table function, returns a table row (but not control) to the invoker of the function.

**Note:**

- If a pipelined table function is part of an autonomous transaction, then it must COMMIT or ROLLBACK before each PIPE ROW statement, to avoid an error in the invoking subprogram.
- To improve performance, the PL/SQL runtime system delivers the piped rows to the invoker in batches.

**Topics**
- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```
pipe_row_statement ::= PIPE ROW ( row ) ;
```

**Semantics**

```
pipe_row_statement
row
```

Row (table element) that the function returns to its invoker, represented by an expression whose type is that of the table element.

If the expression is a record variable, it must be explicitly declared with the data type of the table element. It cannot be declared with a data type that is only structurally identical to the element type. For example, if the element type has a name, then the record variable cannot be declared explicitly with %TYPE or %ROWTYPE or implicitly with %ROWTYPE in a cursor FOR LOOP statement.

**Examples**

- Example 13-29, "Creating and Invoking Pipelined Table Function"
- Example 13-30, "Pipelined Table Function Transforms Each Row to Two Rows"
- Example 13-32, "Pipelined Table Function with Two Cursor Variable Parameters"
14.50 PIPELINED Clause

Instructs the database to iteratively return the results of a table function or polymorphic table function.

Use only with a table function, to specify that it is pipelined. A pipelined table function returns a row to its invoker immediately after processing that row and continues to process rows. To return a row (but not control) to the invoker, the function uses the "PIPE ROW Statement".

A table function returns a collection type.

A polymorphic table function is a table function whose return type is determined by the arguments.

You query both kinds of table functions by using the TABLE keyword before the function name in the FROM clause of the query. For example:

```sql
SELECT * FROM TABLE(function_name(...))
```

The TABLE operator is optional when the table function arguments list or empty list () appears. For example:

```sql
SELECT * FROM function_name()
```

the database then returns rows as they are produced by the function.

The PIPELINED option can appear in the following SQL statements:

- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement

Topics

- Syntax
- Semantics
- Examples
- Related Topics
Syntax

\[ \text{pipelined\_clause ::=} \]

Semantics

\[ \text{pipelined\_clause} \]

The \text{pipelined\_clause} can appear only once in the function.

\textbf{PIEPLINED}

To make a pipelined function, include the \text{pipelined\_clause} in the function definition. If you declare the pipelined function before defining it, you must specify the \text{PIEPLINED} option in the function declaration.

\{ IS | USING \}

- If you specify the keyword \text{PIEPLINED} alone (\text{PIEPLINED IS ...}), then the PL/SQL function body must use the \text{PIPE} keyword. This keyword instructs the database to return single elements of the collection out of the function, instead of returning the whole collection as a single value.

- You can specify the \text{PIEPLINED USING implementation\_type} clause to predefine an interface containing the start, fetch, and close operations. The implementation type must implement the \text{ODCITable} interface and must exist at the time the table function is created. This clause is useful for table functions implemented in external languages such as C++ and Java.

  If the return type of the function is \text{ANYDATASET}, then you must also define a describe method (\text{ODCITableDescribe}) as part of the implementation type of the function.

\[ \text{[schema.] implementation\_type} \]

The implementation type must be an ADT containing the implementation of the \text{ODCIAggregate} subprograms. If you do not specify \text{schema}, then the database assumes that the implementation type is in your schema.

\textbf{Restriction on PIEPLINED}

You cannot specify \text{PIEPLINED} for a nested function or a SQL macro.
Note:

You cannot run a pipelined table function over a database link. The reason is that the return type of a pipelined table function is a SQL user-defined type, which can be used only in a single database (as explained in Oracle Database Object-Relational Developer's Guide). Although the return type of a pipelined table function might appear to be a PL/SQL type, the database actually converts that PL/SQL type to a corresponding SQL user-defined type.

PIPELINED [ ROW | TABLE ] POLYMORPHIC [ USING [schema.] implementation_package ]

The polymorphic table function elaborator can appear in standalone function declaration or package function declaration.

PIPELINED

Required when defining a polymorphic table function.

ROW

Specify ROW when a single input argument of type TABLE determines new columns using any single row.

TABLE

Specify TABLE when a single input argument of type TABLE determines the new columns using the current row and operates on an entire table or a logical partition of a table.

POLYMORPHIC

Restrictions on POLYMORPHIC

The following are not allowed for POLYMORPHIC table functions:

- PARALLEL_ENABLE clause
- RESULT_CACHE clause
- DETERMINISTIC option
- AUTHID property (Invoker's Rights and Definer's Rights Clause)

[USING [schema.] implementation_package]

References the polymorphic table function (PTF) implementation package. The specification must include DESCRIBE method. The specification of OPEN, FETCH_ROWS and CLOSE methods is optional. The specification for the implementation package must already exist (unless the PTF and its implementation reside in the same package).

If a polymorphic table function and its implementation methods are defined in the same package, then the USING clause is optional.

Examples

- Examples for PIPE ROW statement examples
- Skip_col Polymorphic Table Function Example
14.51 Procedure Declaration and Definition

Before invoking a procedure, you must declare and define it. You can either declare it first (with `procedure_declaration`) and then define it later in the same block, subprogram, or package (with `procedure_definition`) or declare and define it at the same time (with `procedure_definition`).

A procedure is a subprogram that performs a specific action. A procedure invocation (or call) is a statement.

A procedure declaration is also called a procedure specification or procedure spec.

Note:

For more information about standalone procedures, see "CREATE PROCEDURE Statement". For more information about package procedures, see "CREATE PACKAGE Statement".

Topics

- Syntax
- Semantics
- Examples
Related Topics

Syntax

\[
\textit{procedure\_declaration ::= }
\]

( \textit{procedure\_properties ::= } )

\[
\textit{procedure\_heading ::= }
\]

See "\textit{parameter\_declaration ::=}".

\[
\textit{procedure\_properties ::= }
\]

( \textit{accessible\_by\_clause ::= , default\_collation\_clause ::= , invoker\_rights\_clause ::= } )

\[
\textit{procedure\_definition ::= }
\]

( \textit{body ::= , declare\_section ::= , call\_spec ::= } )

Semantics

\textit{procedure\_declaration}

Declares a procedure, but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.
procedure_heading

procedure
Name of the procedure that you are declaring or defining.

procedure_properties
Each procedure property can appear only once in the procedure declaration. The properties can appear in any order. Properties appear before the IS or AS keyword in the heading. The properties cannot appear in nested procedures. Only the ACCESSIBLE BY property can appear in package procedures.

Standalone procedures may have the following properties in their declaration.

- ACCESSIBLE BY Clause
- DEFAULT COLLATION Clause
- Invoker's Rights and Definer's Rights (AUTHID Property)

procedure_definition
Either defines a procedure that was declared earlier or both declares and defines a procedure.

declare_section
Declares items that are local to the procedure, can be referenced in body, and cease to exist when the procedure completes execution.

body
Required executable part and optional exception-handling part of the procedure.

Examples

- Example 9-1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure"

Related Topics

In this chapter:

- "Formal Parameter Declaration"
- "Function Declaration and Definition"

In other chapters:

- "PL/SQL Subprograms"
- "CREATE PROCEDURE Statement"

14.53 RAISE Statement

The RAISE statement explicitly raises an exception.

Outside an exception handler, you must specify the exception name. Inside an exception handler, if you omit the exception name, the RAISE statement reraises the current exception.
Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\[ \text{raise\_statement} ::= \]

Semantics

\text{exception}

Name of an exception, either predefined (see Table 12-3) or user-declared (see "Exception Declaration").

\text{exception} is optional only in an exception handler, where the default is the current exception (see "Reraising Current Exception with RAISE Statement").

Examples

- Example 12-10, "Declaring, Raising, and Handling User-Defined Exception"
- Example 12-11, "Explicitly Raising Predefined Exception"
- Example 12-12, "Reraising Exception"

Related Topics

In this chapter:

- "Exception Declaration"
- "Exception Handler"

In other chapters:

- "Raising Exceptions Explicitly"

14.54 Record Variable Declaration

A record variable is a composite variable whose internal components, called fields, can have different data types. The value of a record variable and the values of its fields can change.

You reference an entire record variable by its name. You reference a record field with the syntax record.field.

You can create a record variable in any of these ways:
- Define a record type and then declare a variable of that type.
- Use %ROWTYPE to declare a record variable that represents either a full or partial row of a database table or view.
- Use %TYPE to declare a record variable of the same type as a previously declared record variable.

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

**record_type_definition ::=**

```
TYPE record_type IS RECORD ( field_definition );
```

**field_definition ::=**

```
field datatype NOT NULL := DEFAULT expression ( datatype ::= , expression ::= )
```

**record_variable_declaration ::=**

```
record_type declaration rowtype_attribute record_2 %TYPE;
```

Semantics

**record_type_definition**

**record_type**

Name of the record type that you are defining.
**field_definition**

**field**
Name of the field that you are defining.

**datatype**
Data type of the field that you are defining.

**NOT NULL**
Imposes the NOT NULL constraint on the field that you are defining.
For information about this constraint, see "NOT NULL Constraint".

**expression**
Expression whose data type is compatible with **datatype**. When **record_variable_declaration** is elaborated, the value of **expression** is assigned to **record.field**. This value is the initial value of the field.

**record_variable_declaration**

**record_1**
Name of the record variable that you are declaring.

**record_type**
Name of a previously defined record type. **record_type** is the data type of **record_1**.

**rowtype_attribute**
See "%ROWTYPE Attribute".

**record_2**
Name of a previously declared record variable.

**%TYPE**
See "%TYPE Attribute".

**Examples**
- Example 6-39, "RECORD Type Definition and Variable Declaration"
- Example 6-40, "RECORD Type with RECORD Field (Nested Record)"
- Example 6-41, "RECORD Type with Varray Field"

**Related Topics**
In this chapter:
- "Collection Variable Declaration"
- "%ROWTYPE Attribute"
In other chapters:
- "Record Topics"
14.55 RESTRICT_REFERENCES Pragma

The RESTRICT_REFERENCES pragma asserts that a user-defined subprogram does not read or write database tables or package variables.

Note:
The RESTRICT_REFERENCES pragma is deprecated. Oracle recommends using DETERMINISTIC and PARALLEL_ENABLE instead of RESTRICT_REFERENCES.

Subprograms that read or write database tables or package variables are difficult to optimize, because any invocation of the subprogram might produce different results or encounter errors. If a statement in a user-defined subprogram violates an assertion made by RESTRICT_REFERENCES, then the PL/SQL compiler issues an error message when it parses that statement, unless you specify TRUST.

Typically, this pragma is specified for functions. If a function invokes procedures, then specify this pragma for those procedures also.

Restrictions on RESTRICT_REFERENCES Pragma

• This pragma can appear only in a package specification or ADT specification.
• Only one RESTRICT_REFERENCES pragma can reference a given subprogram.

Topics

• Syntax
• Semantics

Syntax

restrict_references_pragma ::= 

Semantics

subprogram

Name of a user-defined subprogram, typically a function. If subprogram is overloaded, the pragma applies only to the most recent subprogram declaration.

method

PRAGMA RESTRICT_REFERENCES method DEFAULT RNDS WNDS RNPS WNPS TRUST
Name of a MEMBER subprogram. See "CREATE TYPE Statement" for more information.

DEFAULT

Applies the pragma to all subprograms in the package specification or ADT specification (including the system-defined constructor for ADTs).

If you also declare the pragma for an individual subprogram, it overrides the DEFAULT pragma for that subprogram.

RNDS

Asserts that the subprogram reads no database state (does not query database tables).

WNDS

Asserts that the subprogram writes no database state (does not modify tables).

RNPS

Asserts that the subprogram reads no package state (does not reference the values of package variables)

Restriction on RNPS

You cannot specify RNPS if the subprogram invokes the SQLCODE or SQLERRM function.

WNPS

Asserts that the subprogram writes no package state (does not change the values of package variables).

Restriction on WNPS

You cannot specify WNPS if the subprogram invokes the SQLCODE or SQLERRM function.

TRUST

Asserts that the subprogram can be trusted not to violate the other specified assertions and prevents the PL/SQL compiler from checking the subprogram body for violations. Skipping these checks can improve performance.

If your PL/SQL subprogram invokes a C or Java subprogram, then you must specify TRUST for either the PL/SQL subprogram or the C or Java subprogram, because the PL/SQL compiler cannot check a C or Java subprogram for violations at run time.

Note:

To invoke a subprogram from a parallelized DML statement, you must specify all four constraints—RNDS, WNDS, RNPS, and WNPS. No constraint implies another.
14.56 RETURN Statement

The **RETURN** statement immediately ends the execution of the subprogram or anonymous block that contains it.

In a function, the **RETURN** statement assigns a specified value to the function identifier and returns control to the invoker, where execution resumes immediately after the invocation (possibly inside the invoking statement). Every execution path in a function must lead to a **RETURN** statement (otherwise, the PL/SQL compiler issues compile-time warning PLW-05005).

In a procedure, the **RETURN** statement returns control to the invoker, where execution resumes immediately after the invocation.

In an anonymous block, the **RETURN** statement exits its own block and all enclosing blocks.

A subprogram or anonymous block can contain multiple **RETURN** statements.

**Note:**

The **RETURN** statement differs from the **RETURN** clause in a function heading, which specifies the data type of the return value.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```
return_statement ::= 

RETURN

expression

( expression ::= )
```

**Semantics**

**expression**
Optional when the `RETURN` statement is in a pipelined table function. Required when the `RETURN` statement is in any other function. Not allowed when the `RETURN` statement is in a procedure or anonymous block.

The `RETURN` statement assigns the value of `expression` to the function identifier. Therefore, the data type of `expression` must be compatible with the data type in the `RETURN` clause of the function. For information about expressions, see "Expression".

**Examples**

- Example 9-3, “Execution Resumes After RETURN Statement in Function”
- Example 9-4, “Function Where Not Every Execution Path Leads to RETURN Statement”
- Example 9-5, “Function Where Every Execution Path Leads to RETURN Statement”
- Example 9-6, “Execution Resumes After RETURN Statement in Procedure”
- Example 9-7, “Execution Resumes After RETURN Statement in Anonymous Block”

**Related Topics**

In this chapter:

- "Block"
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"

In other chapters:

- "RETURN Statement"

### 14.57 RETURNING INTO Clause

The `RETURNING INTO` clause specifies the variables in which to store the values returned by the statement to which the clause belongs.

The variables can be either individual variables or collections. If the statement affects no rows, then the values of the variables are undefined.

The **static** `RETURNING INTO` clause belongs to a `DELETE`, `INSERT`, or `UPDATE` statement. The **dynamic** `RETURNING INTO` clause belongs to the `EXECUTE IMMEDIATE` statement.

**Note:**

You cannot use the `RETURNING INTO` clause for remote or parallel deletes.

**Topics**

- Syntax
- Semantics
- Examples
Related Topics

Syntax

static_returning_clause ::= 

\[
\text{RETURNING} \\text{RETURN} \\text{column} \\text{into_clause} \\text{bulk_collect_into_clause} 
\]

dynamic_returning_clause ::= 

\[
\text{RETURNING} \\text{RETURN} \\text{into_clause} \\text{bulk_collect_into_clause} 
\]

into_clause ::= 

\[
\text{INTO} \\text{variable} , \text{variable record} \]

bulk_collect_into_clause ::= 

\[
\text{BULK COLLECT INTO} \\text{collection} : \text{host_array} 
\]

Semantics

static_returning_clause

column
Expression whose value is the name of a column of a database table.

into_clause
Specifies the variables or record in which to store the column values that the statement returns.

Restriction on into_clause
Use into_clause in dynamic_returning_clause if and only if dynamic_sql_stmt (which appears in "EXECUTE IMMEDIATE Statement") returns a single row.
**record**

The name of a record variable in which to store the row that the statement returns. For each `select_list` item in the statement, the record must have a corresponding, type-compatible field.

**variable**

Either the name of a scalar variable in which to store a column that the statement returns or the name of a host cursor variable that is declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Each `select_list` item in the statement must have a corresponding, type-compatible variable. The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

**Restriction on variable**

`variable` cannot have the data type `BOOLEAN`.

**bulk_collect_into_clause**

Specifies one or more existing collections or host arrays in which to store the rows that the statement returns. For each `select_list` item in the statement, `bulk_collect_into_clause` must have a corresponding, type-compatible `collection` or `host_array`.

For the reason to use this clause, see "Bulk SQL and Bulk Binding".

**Restrictions on bulk_collect_into_clause**

- Use the `bulk_collect_into_clause` clause in `dynamic_returning_clause` if and only if `dynamic_sql_stmt` (which appears in "EXECUTE IMMEDIATE Statement") can return multiple rows.
- You cannot use `bulk_collect_into_clause` in client programs.
- When the statement that includes `bulk_collect_into_clause` requires implicit data type conversions, `bulk_collect_into_clause` can have only one `collection` or `host_array`.

**collection**

Name of a collection variable in which to store the rows that the statement returns.

**Restrictions on collection**

- `collection` cannot be the name of an associative array that is indexed by a string.
- When the statement requires implicit data type conversions, `collection` cannot be the name of a collection of a composite type.

**host_array**

Name of an array declared in a PL/SQL host environment and passed to PL/SQL as a bind variable. Do not put space between the colon (:) and `host_array`.

**Examples**

- Example 6-55, "UPDATE Statement Assigns Values to Record Variable"
- Example 7-1, "Static SQL Statements"
- Example 13-25, "Returning Deleted Rows in Two Nested Tables"
• Example 13-26, "DELETE with RETURN BULK COLLECT INTO in FORALL Statement"

Related Topics

In this chapter:
• "DELETE Statement Extension"
• "EXECUTE IMMEDIATE Statement"
• "FETCH Statement"
• "SELECT INTO Statement"
• "UPDATE Statement Extensions"

In other chapters:
• "Using SQL Statements to Return Rows in PL/SQL Record Variables"
• "EXECUTE IMMEDIATE Statement"
• "RETURNING INTO Clause with BULK COLLECT Clause"

14.58 RESULT_CACHE Clause

Indicates to store the function results into the server result cache.

The RESULT_CACHE clause can appear in the following SQL statements:
• CREATE FUNCTION Statement
• CREATE TYPE BODY Statement

Topics
• Syntax
• Semantics
• Examples
• Related Topics

Syntax

\[
\text{result_cache_clause ::=}
\]

Semantics

\[
\text{result_cache_clause}
\]

\[
\text{result_cache_clause can appear only once in the function.}
\]
RESULT_CACHE

To make a function result-cached, include the RESULT_CACHE clause in the function definition. If you declare the function before defining it, you must also include the RESULT_CACHE option in the function declaration.

Restriction on RESULT_CACHE

• RESULT_CACHE is disallowed on functions with OUT or IN OUT parameters.
• RESULT_CACHE is disallowed on functions with IN or RETURN parameter of (or containing) these types:
  – BLOB
  – CLOB
  – NCLOB
  – REF CURSOR
  – Collection
  – Object
  – Record or PL/SQL collection that contains an unsupported return type
• RESULT_CACHE is disallowed on function in an anonymous block.
• RESULT_CACHE is disallowed on pipelined table function, nested function and SQL macro.

RELIES_ON

Specifies the data sources on which the results of the function depend. Each data_source is the name of either a database table or view.

Note:

• This clause is deprecated. As of Oracle Database 12c, the database detects all data sources that are queried while a result-cached function is running, and RELIES_ON clause does nothing.
• You cannot use RELIES_ON clause in a function declared in an anonymous block.

Examples

• Examples of Result-Cached Functions

Related Topics

In this chapter:

• Function Declaration and Definition

In other chapters:

• PL/SQL Function Result Cache
In other books:

- *Oracle Database Performance Tuning Guide*

### 14.59 %ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record that represents either a full or partial row of a database table or view.

For every visible column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly. Making an invisible column visible changes the structure of some records declared with the `%ROWTYPE` attribute.

The record fields do not inherit the constraints or initial values of the corresponding columns.

The `%ROWTYPE` attribute cannot be used if the referenced character column has a collation other than `USING_NLS_COMP`.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

\[
\text{rowtype\_attribute} ::= \\
\hspace{1em} \text{explicit\_cursor\_name} \\text{cursor\_variable\_name} \text{db\_table\_or\_view\_name} \% \text{ROWTYPE}
\]

**Semantics**

\[
\text{rowtype\_attribute}
\]

**explicit\_cursor\_name**

Name of an explicit cursor. For every column selected by the query associated with `explicit\_cursor\_name`, the record has a field with the same name and data type.

**cursor\_variable\_name**

Name of a strong cursor variable. For every column selected by the query associated with `cursor\_variable\_name`, the record has a field with the same name and data type.

**db\_table\_or\_view\_name**

Name of a database table or view that is accessible when the declaration is elaborated. For every column of `db\_table\_or\_view\_name`, the record has a field with the same name and data type.
Examples

- **Example 6-43**, "%ROWTYPE Variable Represents Full Database Table Row"
- **Example 6-44**, "%ROWTYPE Variable Does Not Inherit Initial Values or Constraints"
- **Example 6-45**, "%ROWTYPE Variable Represents Partial Database Table Row"
- **Example 6-46**, "%ROWTYPE Variable Represents Join Row"
- **Example 6-49**, "%ROWTYPE Affected by Making Invisible Column Visible"
- **Example 6-51**, "Assigning %ROWTYPE Record to RECORD Type Record"

Related Topics

In this chapter:

- "Cursor Variable Declaration"
- "Explicit Cursor Declaration and Definition"
- "Record Variable Declaration"
- "%TYPE Attribute"

In other chapters:

- About Data-Bound Collation
- "Declaring Items using the %ROWTYPE Attribute"
- "%ROWTYPE Attribute and Invisible Columns"

### 14.60 Scalar Variable Declaration

A scalar variable stores a value with no internal components. The value can change. A scalar variable declaration specifies the name and data type of the variable and allocates storage for it.

The declaration can also assign an initial value and impose the **NOT NULL** constraint.

You reference a scalar variable by its name.

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

\[ \text{variable} := \text{variable} \mid \text{datatype} \mid \text{NOT} \mid \text{NULL} \mid \text{DEFAULT} \mid \text{expression} \]
( expression ::= )

Semantics

variable_declaration

variable

Name of the variable that you are declaring.

datatype

Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics.

For information about scalar data types, see "PL/SQL Data Types".

NOT NULL

Imposes the NOT NULL constraint on the variable. For information about this constraint, see "NOT NULL Constraint".

expression

Value to be assigned to the variable when the declaration is elaborated. expression and variable must have compatible data types.

Examples

• Example 3-11, "Scalar Variable Declarations"
• Example 3-13, "Variable and Constant Declarations with Initial Values"
• Example 3-14, "Variable Initialized to NULL by Default"
• Example 3-9, "Variable Declaration with NOT NULL Constraint"

Related Topics

In this chapter:

• "Assignment Statement"
• "Collection Variable Declaration"
• "Constant Declaration"
• "Expression"
• "Record Variable Declaration"
• "%ROWTYPE Attribute"
• "%TYPE Attribute"

In other chapters:

• "Declaring Variables"
• "PL/SQL Data Types"
14.61 SELECT INTO Statement

The SELECT INTO statement retrieves values from one or more database tables (as the SQL SELECT statement does) and stores them in variables (which the SQL SELECT statement does not do).

⚠ Caution:

The SELECT INTO statement with the BULK COLLECT clause is vulnerable to aliasing, which can cause unexpected results. For details, see "SELECT BULK COLLECT INTO Statements and Aliasing".

🔍 See Also:

Oracle Database SQL Language Reference for the syntax of the SQL SELECT statement

Topics

• Syntax
• Semantics
• Examples
• Related Topics

Syntax

select_into_statement ::=
Semantics

$select\_into\_statement$

DISTINCT | UNIQUE

Specify DISTINCT or UNIQUE if you want the database to return only one copy of each set of duplicate rows selected. These two keywords are synonymous. Duplicate rows are those with matching values for each expression in the select list.

Restrictions on DISTINCT and UNIQUE Queries

• The total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter $DB\_BLOCK\_SIZE$.

• You cannot specify DISTINCT if the select list contains LOB columns.

ALL

(Default) Causes the database to return all rows selected, including all copies of duplicates.

$select\_list$

If the SELECT INTO statement returns no rows, PL/SQL raises the predefined exception NO_DATA_FOUND. To guard against this exception, select the result of the aggregate function COUNT(*), which returns a single value even if no rows match the condition.

$into\_clause$

With this clause, the SELECT INTO statement retrieves one or more columns from a single row and stores them in either one or more scalar variables or one record variable. For more information, see "$into\_clause::=".

$bulk\_collect\_into\_clause$

With this clause, the SELECT INTO statement retrieves an entire result set and stores it in one or more collection variables. For more information, see "$bulk\_collect\_into\_clause::=".

$rest\_of\_statement$

Anything that can follow the keyword FROM in a SQL SELECT statement, described in Oracle Database SQL Language Reference.

Examples

• Example 3-25, "Assigning Value to Variable with SELECT INTO Statement"

• Example 6-53, "SELECT INTO Assigns Values to Record Variable"

• Example 7-37, "ROLLBACK Statement"

• Example 7-38, "SAVEPOINT and ROLLBACK Statements"

• Example 7-43, "Declaring Autonomous Function in Package"

• Example 8-20, "Validation Checks Guarding Against SQL Injection"
14.62 SERIALLY_REUSABLE Pragma

The **SERIALLY_REUSABLE** pragma specifies that the package state is needed for only one call to the server (for example, an OCI call to the database or a stored procedure invocation through a database link).

After this call, the storage for the package variables can be reused, reducing the memory overhead for long-running sessions.

This pragma is appropriate for packages that declare large temporary work areas that are used once in the same session.

The **SERIALLY_REUSABLE** pragma can appear in the *declare_section* of the specification of a bodiless package, or in both the specification and body of a package, but not in only the body of a package.

**Topics**

- Syntax
- Examples
- Related Topics
Syntax

```
serially_reusable_pragma ::= 
```

- `PRAGMA SERIALLY_REUSABLE ;`

Examples

- **Example 11-4**, "Creating SERIALLY_REUSABLE Packages"
- **Example 11-5**, "Effect of SERIALLY_REUSABLEPragma"
- **Example 11-6**, "Cursor in SERIALLY_REUSABLE Package Open at Call Boundary"

Related Topics

- "SERIALLY_REUSABLE Packages"
- "Pragmas"

### 14.63 SHARING Clause

The **SHARING** clause applies only when creating an object in an application root in the context of an application maintenance. This type of object is called an application common object and it can be shared with the application PDBs that belong to the application root.

The **SHARING** clause can appear in the context of creating application common PL/SQL and SQL objects with these SQL statements.

<table>
<thead>
<tr>
<th>Application Common Object Statement</th>
<th>Possible SHARING Attributes</th>
<th>Syntax and Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE ANALYTIC VIEW</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE ATTRIBUTE DIMENSION</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE CLUSTER</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE CONTEXT</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE DIRECTORY</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE FUNCTION</td>
<td>METADATA, NONE</td>
<td>CREATE FUNCTION Statement</td>
</tr>
<tr>
<td>CREATE HIERARCHY</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
</tbody>
</table>
### Table 14-2  (Cont.) Summary of Possible Sharing Attributes by Application Common Object Type

<table>
<thead>
<tr>
<th>Application Common Object Statement</th>
<th>Possible SHARING Attributes</th>
<th>Syntax and Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE INDEXTYPE</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE JAVA</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE LIBRARY</td>
<td>METADATA, NONE</td>
<td>CREATE LIBRARY Statement</td>
</tr>
<tr>
<td>CREATE MATERIALIZED VIEW LOG</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE OPERATOR</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>METADATA, NONE</td>
<td>CREATE PROCEDURE Statement</td>
</tr>
<tr>
<td>CREATE PACKAGE</td>
<td>METADATA, NONE</td>
<td>CREATE PACKAGE Statement</td>
</tr>
<tr>
<td>CREATE PACKAGE BODY</td>
<td>METADATA, NONE</td>
<td>CREATE PACKAGE BODY Statement</td>
</tr>
<tr>
<td>CREATE SEQUENCE</td>
<td>METADATA, DATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE SYNONYM</td>
<td>METADATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>METADATA, DATA, EXTENDED DATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
<tr>
<td>CREATE TRIGGER</td>
<td>METADATA, NONE</td>
<td>CREATE TRIGGER Statement</td>
</tr>
<tr>
<td>CREATE TYPE</td>
<td>METADATA, NONE</td>
<td>CREATE TYPE Statement</td>
</tr>
<tr>
<td>CREATE TYPE BODY</td>
<td>METADATA, NONE</td>
<td>CREATE TYPE BODY Statement</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>METADATA, DATA, EXTENDED DATA, NONE</td>
<td>Oracle Database SQL Language Reference</td>
</tr>
</tbody>
</table>

### Topics
- Syntax
- Semantics
- Related Topics

### Syntax

```
sharing_clause ::= 
```

```
SHARING = METADATA
NONE
```

### Diagram

```
<Diagram showing the sharing clause syntax>
```
Semantics

**sharing_clause**

Specifies how the object is shared using one of the following sharing attributes:

- **METADATA** - A metadata link shares the metadata, but its data is unique to each container. This type of object is referred to as a **metadata-linked application common object**.
- **NONE** - The object is not shared and can only be accessed in the application root.

If you omit this clause during an application operation, then the database uses the value of the DEFAULT_SHARING initialization parameter to determine the sharing attribute of the object. If the DEFAULT_SHARING initialization parameter does not have a value, then the default is **METADATA**.

Restrictions on **sharing_clause**

The sharing clause may only appear during an application installation, upgrade or patch in an application root. You must issue an **ALTER PLUGGABLE DATABASE APPLICATION ... BEGIN** statement to start the operation and an **ALTER PLUGGABLE DATABASE APPLICATION ... END** statement to end the operation. The **sharing_clause** is illegal outside this context and this implies the object is not shared.

You cannot change the sharing attribute of an object after it is created.

Generally, common objects cannot depend on local objects. The exceptions to this rule are:

- the common object being created is a synonym
- the common object being created depends on a local operator

If you try to create a common object that depends on local objects other than these two exceptions, it will result in the creation of a local object.

Related Topics

In other books:

- *Oracle Database Concepts* for more information about application maintenance
- *Oracle Database Concepts* for an example of patching an application using the automated technique
- *Oracle Database Reference* for more information on the DEFAULT_SHARING initialization parameter
- *Oracle Database Reference* for more information on the ALL_OBJECTS view initialization SHARING and APPLICATION columns
- *Oracle Database Administrator’s Guide* for complete information on creating application common objects
14.65 SQLCODE Function

In an exception handler, the SQLCODE function returns the numeric code of the exception being handled. (Outside an exception handler, SQLCODE returns 0.)

For an internally defined exception, the numeric code is the number of the associated Oracle Database error. This number is negative except for the error “no data found”, whose numeric code is +100.

For a user-defined exception, the numeric code is either +1 (default) or the error code associated with the exception by the EXCEPTION_INIT pragma.

A SQL statement cannot invoke SQLCODE.

If a function invokes SQLCODE, and you use the RESTRICT_REFERENCES pragma to assert the purity of the function, then you cannot specify the constraints WNPS and RNPS.

Topics

• Syntax
• Examples
• Related Topics

Syntax

sqlcode_function ::= SQLCODE

Examples

• Example 12-23, "Displaying SQLCODE and SQLERRM Values"

Related Topics

In this chapter:

• "Block"
• "EXCEPTION_INIT Pragma"
• "Exception Handler"
• "RESTRICT_REFERENCES Pragma"
• "SQLERRM Function"

In other chapters:

• "Retrieving Error Code and Error Message"
14.66 SQLERRM Function

The SQLERRM function returns the error message associated with an error code.

**Note:**
The language of the error message depends on the NLS_LANGUAGE parameter. For information about this parameter, see Oracle Database Globalization Support Guide.

A SQL statement cannot invoke SQLERRM.

If a function invokes SQLERRM, and you use the RESTRICT_REFERENCES pragma to assert the purity of the function, then you cannot specify the constraints WNPS and RNPS.

**Note:**
DBMS_UTILITY.FORMAT_ERROR_STACK is recommended over SQLERRM, unless you use the FORALL statement with its SAVE EXCEPTIONS clause. For more information, see "Retrieving Error Code and Error Message".

Topics
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

sqlerrm_function ::=  

```sql
   SQLERRM ( error_code )
```

See Also:

Oracle Database Error Messages Reference for a list of Oracle Database error messages and information about them, including their numbers.
Semantics

sqlerrm_function

error_code

Expression whose value is an Oracle Database error code.

Default: error code associated with the current value of SQLCODE.

Like SQLCODE, SQLERRM without error_code is useful only in an exception handler. Outside an exception handler, or if the value of error_code is zero, SQLERRM returns ORA-0000.

If the value of error_code is +100, SQLERRM returns ORA-01403.

If the value of error_code is a positive number other than +100, SQLERRM returns this message:

- error_code: non-ORACLE exception

If the value of error_code is a negative number whose absolute value is an Oracle Database error code, SQLERRM returns the error message associated with that error code. For example:

BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-6511): ' || TO_CHAR(SQLERRM(-6511)));
END;
/

Result:

SQLERRM(-6511): ORA-06511: PL/SQL: cursor already open

If the value of error_code is a negative number whose absolute value is not an Oracle Database error code, SQLERRM returns this message:

ORA-error_code: Message error_code not found; product=RDBMS;
facility=ORA

For example:

BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-50000): ' || TO_CHAR(SQLERRM(-50000)));
END;
/

Result:

SQLERRM(-50000): ORA-50000: Message 50000 not found; product=RDBMS;
facility=ORA

Examples

- Example 12-23, “Displaying SQLCODE and SQLERRM Values”
- Example 13-13, “Handling FORALL Exceptions After FORALL Statement Completes”
- Example 13-13, “Handling FORALL Exceptions After FORALL Statement Completes”
14.67 SUPPRESSES_WARNING_6009 Pragma

The SUPPRESSES_WARNING_6009 pragma marks a subroutine to indicate that the PLW-06009 warning is suppressed at its call site in an OTHERS exception handler. The marked subroutine has the same effect as a RAISE statement and suppresses the PLW-06009 compiler warning.

The OTHERS exception handler does not issue the compiler warning PLW-06009 if an exception is raised explicitly using either a RAISE statement or the RAISE_APPLICATION_ERROR procedure as the last statement. Similarly, a call to a subroutine marked with the SUPPRESSES_WARNING_6009 pragma, from the OTHERS exception handler, does not issue the PLW-06009 warning.

The SUPPRESSES_WARNING_6009 pragma can appear in the following SQL statements:

- CREATE FUNCTION Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement
- CREATE PROCEDURE Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

Topics

- Syntax
- Semantics
- Examples
- Related Topics
Syntax

suppresses_warning_6009 pragma ::= 

Semantics

suppresses_warning_6009 pragma

The SUPPRESSES_WARNING_6009 pragma applies to standalone subprograms, packaged subprograms and to the methods in an Abstract Data Type definition.

For a standalone subprogram, the SUPPRESSES_WARNING_6009 pragma may appear as the first item in the declaration block of a subprogram definition, immediately after the keyword IS or AS.

In a package specification, a package body and a type specification, the SUPPRESSES_WARNING_6009 pragma must appear immediately after the subprogram declaration.

If the subprogram has separate declaration and definition, the SUPPRESSES_WARNING_6009 pragma may be applied either to the subprogram declaration, or to the subprogram definition, or to both.

For overloaded subprograms, SUPPRESSES_WARNING_6009 pragma only applies to the marked overload.

When the SUPPRESSES_WARNING_6009 pragma is applied to a subprogram in a package specification, the subprogram is marked for use both in the package body and in the invokers of the package.

When the SUPPRESSES_WARNING_6009 pragma is applied to a subprogram in the definition of a package body, the subprogram is marked for use only in the package body even if the subprogram is declared in the package specification.

The SUPPRESSES_WARNING_6009 pragma applied to a subprogram in a base type object, is inherited in a derived type object unless there is an override without the pragma in the derived type object.

The SUPPRESSES_WARNING_6009 pragma may be terminated with a ",", " when applied to a subprogram in a type specification. In all other contexts, the pragma is terminated by "; ".

The SUPPRESSES_WARNING_6009 pragma on the subprogram provides a hint to the compiler to suppress the warning PLW-06009 at its call site.

pls_identifier

Identifier of the PL/SQL element to which the pragma is applied.

The identifier is a parameter to the SUPPRESSES_WARNING_6009 pragma that must name the subprogram to which it is applied.
If the identifier in the SUPPRESSES_WARNING_6009 does not identify a subroutine in the declaration section, the pragma has no effect.

Examples

Example 14-41  Enabling the PLW-6009 Warning

This example shows how to set the PLSQL_WARNINGS parameter to enable the PLW-6009 warning in a session for demonstration purpose.

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:(6009)';
```

Example 14-42  SUPPRESSES_WARNING_6009 Pragma in a Procedure

This example shows a standalone procedure p1 marked with the SUPPRESSES_WARNING_6009 pragma. The p1 procedure is invoked from an OTHERS exception handler in procedure p2, which does not raise an exception explicitly.

```
CREATE PROCEDURE p1
AUTHID DEFINER
IS
  PRAGMA SUPPRESSSE_WARNING_6009(p1);
BEGIN
  RAISE_APPLICATION_ERROR(-20000, 'Unexpected error raised');
END;
/
```

The PLW-6009 warning is not issued when compiling procedure p2.

```
CREATE PROCEDURE p2
AUTHID DEFINER
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('In procedure p2');
EXCEPTION
  WHEN OTHERS THEN
    p1;
END p2;
/
```

Example 14-43  SUPPRESSES_WARNING_6009 Pragma in a Function

This example shows a standalone function f1 marked with the SUPPRESSES_WARNING_6009 pragma. This function is invoked from an OTHERS exception handler in function f2, which does not explicitly have a RAISE statement.

```
CREATE FUNCTION f1(id NUMBER) RETURN NUMBER
AUTHID DEFINER
IS
  PRAGMA SUPPRESSSE_WARNING_6009(f1);
  x NUMBER;
BEGIN
  x := id + 1;
RETURN x;
```
The PLW-06009 warning is not issued when compiling function f2.

CREATE FUNCTION f2 (numval NUMBER) RETURN NUMBER
AUTHID DEFINER
IS
  i NUMBER;
BEGIN
  i := numval + 1;
  RETURN i;
EXCEPTION
  WHEN OTHERS THEN
    RETURN f1(i);
END;
/

Example 14-44  SUPPRESSES_WARNING_6009 Pragma in an Overloaded Subprogram in a Package Specification

This example shows an overloaded procedure p1, declared in a package specification. Only the second overload of p1 is marked with the SUPPRESSES_WARNING_6009 pragma. This marked overload is invoked from the OTHERS exception handler in procedure p6, which does not have an explicit RAISE statement.

CREATE PACKAGE pk1 IS
  PROCEDURE p1(x NUMBER);
  PROCEDURE p1;
  PRAGMA SUPPRESSES_WARNING_6009 (p1);
END;
/

CREATE OR REPLACE PACKAGE BODY pk1 IS
  PROCEDURE p1(x NUMBER) IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('In the first overload');
      END;

  PROCEDURE p1 IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('In the second overload');
      RAISE_APPLICATION_ERROR(-20000, 'Unexpected error');
      END;
    END;
  END;
/
The procedure p6 invokes the p1 second overloaded procedure. The compiler does not issue a PLW-06009 warning when compiling procedure p6.

CREATE OR REPLACE PROCEDURE p6 AUTHID DEFINER IS
  j NUMBER := 5;
BEGIN
  j := j + 2;
EXCEPTION
  WHEN OTHERS THEN
    pk1.p1;
END;
/

Example 14-45  SUPPRESSES_WARNING_6009 Pragma in a Forward Declaration in a Package Body

This example shows a forward declaration subprogram marked with the SUPPRESSES_WARNING_6009 pragma in a package body. This marked procedure pn is invoked from the OTHERS exception handler in procedure p5, which has no RAISE statement.

CREATE OR REPLACE PACKAGE pk2 IS
  PROCEDURE p5;
END;
/

The compiler does not issue a PLW-06009 warning when creating the package body.

CREATE OR REPLACE PACKAGE BODY pk2 IS
  PROCEDURE pn; /* Forward declaration */
  PRAGMA SUPPRESSES_WARNING_6009(pn);

  PROCEDURE p5 IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Computing');
    EXCEPTION
      WHEN OTHERS THEN
        pn;
    END;

  PROCEDURE pn IS
    BEGIN
      RAISE_APPLICATION_ERROR(-20000, 'Unexpected error');
    END;
END;
/
Example 14-46  SUPPRESSES_WARNING_6009 Pragma in Object Type Methods

This example shows the SUPPRESSES_WARNING_6009 pragma applied to a member method declared in the newid Abstract Data Type (ADT) definition. The marked procedure log_error is invoked from the OTHERS exception handler in the type body, which does not have a RAISE statement.

CREATE OR REPLACE TYPE newid AUTHID DEFINER
AS OBJECT(
    ID1 NUMBER,
    MEMBER PROCEDURE incr,
    MEMBER PROCEDURE log_error,
    PRAGMA SUPPRESSES_WARNING_6009(log_error)
);
/

The compiler does not issue the PLW-06009 warning when compiling the type body.

CREATE OR REPLACE TYPE BODY newid
AS
    MEMBER PROCEDURE incr
    IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE('Computing value');
        EXCEPTION
            WHEN OTHERS THEN
                log_error;
    END;

    MEMBER PROCEDURE log_error
    IS
        BEGIN
            RAISE_APPLICATION_ERROR(-20000, 'Unexpected error');
        END;
END;
/

Related Topics

- Exception Handler
- Pragmas
- PL/SQL Error Handling
- RAISE Statement
- Raising Exceptions Explicitly
14.68 %TYPE Attribute

The %TYPE attribute lets you declare a constant, variable, collection element, record field, or subprogram parameter to be of the same data type as a previously declared variable or column (without knowing what that type is).

The %TYPE attribute cannot be used if the referenced character column has a collation other than USING_NLS_COMP.

The item declared with %TYPE is the referencing item, and the previously declared item is the referenced item.

The referencing item inherits the following from the referenced item:

- Data type and size
- Constraints (unless the referenced item is a column)

The referencing item does not inherit the initial value of the referenced item.

If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

\[
\text{type\_attribute} ::= \\
\text{collection\_variable\_name} \text{cursor\_variable\_name} \text{db\_table\_or\_view\_name} . \text{column\_name} \text{object\_name} \text{record\_variable\_name} . \text{field\_name} \text{scalar\_variable\_name} \% \text{TYPE}
\]

Semantics

\text{type\_attribute}

\text{collection\_variable\_name}

Name of a collection variable.

Restriction on \text{collection\_variable\_name}
In a constant declaration, *collection_variable* cannot be an associative array variable.

**cursor_variable_name**
Name of a cursor variable.

**db_table_or_view_name**
Name of a database table or view that is accessible when the declaration is elaborated.

**column_name**
Name of a column of *db_table_or_view*.

**object_name**
Name of an instance of an ADT.

**record_variable_name**
Name of a record variable.

**field_name**
Name of a field of *record_variable*.

**scalar_variable_name**
Name of a scalar variable.

Examples

- Example 3-15, "Declaring Variable of Same Type as Column"
- Example 3-16, "Declaring Variable of Same Type as Another Variable"

Related Topics

In this chapter:

- "Constant Declaration"
- "%ROWTYPE Attribute"
- "Scalar Variable Declaration"

In other chapters:

- About Data-Bound Collation
- "Declaring Items using the %TYPE Attribute"
14.69 UDF Pragma

The `UDF` pragma tells the compiler that the PL/SQL unit is a user defined function that is used primarily in SQL statements, which might improve its performance.

**Syntax**

```
udf_pragma ::= PRAGMA UDF;
```

14.70 UPDATE Statement Extensions

PL/SQL extends the `update_set_clause` and `where_clause` of the SQL UPDATE statement as follows:

- In the `update_set_clause`, you can specify a record. For each selected row, the UPDATE statement updates each column with the value of the corresponding record field.
- In the `where_clause`, you can specify a CURRENT OF clause, which restricts the UPDATE statement to the current row of the specified cursor.

See Also:

*Oracle Database SQL Language Reference* for the syntax of the SQL UPDATE statement

**Topics**

- Syntax
- Semantics
- Examples
- Related Topics

**Syntax**

```
update_set_clause ::= SET ROW = record
```

```
where_clause ::= WHERE CURRENT OF for_update_cursor
```
Semantics

**update_set_clause**

**record**

Name of a record variable that represents a row of the item described by `dml_table_expression_clause`. That is, for every column of the row, the record must have a field with a compatible data type. If a column has a NOT NULL constraint, then its corresponding field cannot have a NULL value.

**where_clause**

**for_update_cursor**

Name of a FOR UPDATE cursor; that is, an explicit cursor associated with a FOR SELECT UPDATE statement.

Examples

- Example 6-58, "Updating Rows with Records"

Related Topics

In this chapter:

- "Explicit Cursor Declaration and Definition"
- "Record Variable Declaration"
- "%ROWTYPE Attribute"

In other chapters:

- "Updating Rows with Records"
- "Restrictions on Record Inserts and Updates"
- "SELECT FOR UPDATE and FOR UPDATE Cursors"

### 14.71 WHILE LOOP Statement

The **WHILE LOOP** statement runs one or more statements while a condition is TRUE.

The **WHILE LOOP** statement ends when the condition becomes FALSE or NULL, or when a statement inside the loop transfers control outside the loop or raises an exception.
Syntax

\[ \text{while\_loop\_statement ::= \text{WHILE boolean\_expression LOOP statement END LOOP}} \]

( \( boolean\_expression ::= \), \( statement ::= \) )

Semantics

\textit{while\_loop\_statement}

\textit{boolean\_expression}

Expression whose value is \textit{TRUE}, \textit{FALSE}, or \textit{NULL}.

\textit{boolean\_expression} is evaluated at the beginning of each iteration of the loop. If its value is \textit{TRUE}, the statements after \textit{LOOP} run. Otherwise, control transfers to the statement after the \textit{WHILE LOOP} statement.

\textit{statement}

To prevent an infinite loop, at least one statement must change the value of \textit{boolean\_expression} to \textit{FALSE} or \textit{NULL}, transfer control outside the loop, or raise an exception. The statements that can transfer control outside the loop are:

- "\text{CONTINUE Statement}" (when it transfers control to the next iteration of an enclosing labeled loop)
- "\text{EXIT Statement}"
- "\text{GOTO Statement}"
- "\text{RAISE Statement}"

\textit{label}

Label that identifies \textit{while\_loop\_statement} (see "\textit{statement ::=}\" and "\textit{label}"). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when \textit{LOOP} statements are nested, but only if you ensure that the label in the \textit{END LOOP} statement matches a label at the beginning of the same \textit{LOOP} statement (the compiler does not check).

Examples

\textbf{Example 14-47  \textit{WHILE LOOP Statements}}

The statements in the first \textit{WHILE LOOP} statement never run, and the statements in the second \textit{WHILE LOOP} statement run once.

\begin{verbatim}
DECLARE
  done BOOLEAN := FALSE;
BEGIN
  WHILE done LOOP
    DBMS_OUTPUT.PUT_LINE ('This line does not print.');
    done := TRUE;  -- This assignment is not made.
  END LOOP;
END;
\end{verbatim}
END LOOP;

WHILE NOT done LOOP
   DBMS_OUTPUT.PUT_LINE ('Hello, world!');
   done := TRUE;
END LOOP;
END;
/

Result:
Hello, world!

Related Topics
- "Basic LOOP Statement"
- "Cursor FOR LOOP Statement"
- "Explicit Cursor Declaration and Definition"
- "FETCH Statement"
- "FOR LOOP Statement"
- "FORALL Statement"
- "OPEN Statement"
- "WHILE LOOP Statement"
SQL Statements for Stored PL/SQL Units

This chapter explains how to use the SQL statements that create, change, and drop stored PL/SQL units.

CREATE [ OR REPLACE ] Statements

Each of these SQL statements creates a PL/SQL unit at schema level and stores it in the database:

- CREATE FUNCTION Statement
- CREATE LIBRARY Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement
- CREATE PROCEDURE Statement
- CREATE TRIGGER Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

Each of these CREATE statements has an optional OR REPLACE clause. Specify OR REPLACE to re-create an existing PL/SQL unit—that is, to change its declaration or definition without dropping it, re-creating it, and re-granting object privileges previously granted on it. If you redefine a PL/SQL unit, the database recompiles it.

⚠️ Caution:

A CREATE OR REPLACE statement does not issue a warning before replacing the existing PL/SQL unit.

None of these CREATE statements can appear in a PL/SQL block.

ALTER Statements

To recompile an existing PL/SQL unit without re-creating it (without changing its declaration or definition), use one of these SQL statements:

- ALTER FUNCTION Statement
- ALTER LIBRARY Statement
- ALTER PACKAGE Statement
- ALTER PROCEDURE Statement
- ALTER TRIGGER Statement
- ALTER TYPE Statement
Reasons to use an ALTER statement are:

- To explicitly recompile a stored unit that has become invalid, thus eliminating the need for implicit runtime recompilation and preventing associated runtime compilation errors and performance overhead.
- To recompile a stored unit with different compilation parameters.
- To enable or disable a trigger.
- To specify the EDITIONABLE or NONEDITIONABLE property of a stored unit whose schema object type is not yet editionable in its schema.

The ALTER TYPE statement has additional uses.

DROP Statements

To drop an existing PL/SQL unit from the database, use one of these SQL statements:

- DROP FUNCTION Statement
- DROP LIBRARY Statement
- DROP PACKAGE Statement
- DROP PROCEDURE Statement
- DROP TRIGGER Statement
- DROP TYPE Statement
- DROP TYPE BODY Statement

Related Topics

- For instructions for reading the syntax diagrams in this chapter, see Oracle Database SQL Language Reference.
- For information about editioned and noneditioned objects, see Oracle Database Development Guide.
- For information about compilation parameters, see "PL/SQL Units and Compilation Parameters".

15.1 ALTER FUNCTION Statement

The ALTER FUNCTION statement explicitly recompiles a standalone function.

Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

Note:

This statement does not change the declaration or definition of an existing function. To redeclare or redefine a standalone function, use the "CREATE FUNCTION Statement" with the OR REPLACE clause.

Topics

- Prerequisites
Prerequisites

If the function is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the function must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

```sql
alter_function ::= ALTER FUNCTION schema . function_name function_compile_clause EDITIONABLE NONEDITIONABLE ;
```

Semantics

`alter_function`

`schema`

Name of the schema containing the function. **Default:** your schema.

`function_name`

Name of the function to be recompiled.

```
{ EDITIONABLE | NONEDITIONABLE }
```

Specifies whether the function becomes an editioned or noneditioned object if editioning is later enabled for the schema object type `FUNCTION` in `schema`. **Default:** `EDITIONABLE`. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

`function_compile_clause`

Recompiles the function, whether it is valid or invalid.

See `compile_clause` semantics.
See also DEFAULT COLLATION Clause compilation semantics.

Example

**Example 15-1  Recompiling a Function**

To explicitly recompile the function *get_bal* owned by the sample user *oe*, issue this statement:

```
ALTER FUNCTION oe.get_bal COMPILE;
```

If the database encounters no compilation errors while recompiling *get_bal*, then *get_bal* becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling *get_bal* results in compilation errors, then the database returns an error, and *get_bal* remains invalid.

The database also invalidates all objects that depend upon *get_bal*. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

Related Topics

• "CREATE FUNCTION Statement"
• "DROP FUNCTION Statement"

15.2 ALTER LIBRARY Statement

The ALTER LIBRARY statement explicitly recompiles a library.

Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

**Note:**

This statement does not change the declaration or definition of an existing library. To redeclare or redefine a library, use the "CREATE LIBRARY Statement" with the OR REPLACE clause.

Topics

• Prerequisites
• Syntax
• Semantics
• Example
• Related Topics

Prerequisites

If the library is in the SYS schema, you must be connected as SYSDBA. Otherwise, the library must be in your schema or you must have the ALTER ANY LIBRARY system privilege.
Syntax

\[
\text{alter\_library} ::= \
\]

\[
\text{ALTER LIBRARY }\text{schema . }\text{library\_name }\text{library\_compile\_clause }\text{EDITIONABLE NONEDITIONABLE }\; \
\]

\[
\text{library\_compile\_clause} ::= \
\]

\[
\text{COMPILE DEBUG compiler\_parameters\_clause REUSE SETTINGS } \
\]

\[(\text{compiler\_parameters\_clause} ::= )\]

Semantics

\text{alter\_library}

\text{library\_name}

Name of the library to be recompiled.

\{ \text{EDITIONABLE | NONEDITIONABLE} \}

Specifies whether the library becomes an editioned or noneditioned object if editioning is later enabled for the schema object type \text{LIBRARY} in \text{schema}. \textbf{Default: EDITIONABLE}. For information about altering editioned and noneditioned objects, see \textit{Oracle Database Development Guide}.

\text{library\_compile\_clause}

Recompiles the library.

See \textit{compile\_clause} and \textit{compiler\_parameters\_clause} semantics.

Example

\textbf{Example 15-2 Recompiling a Library}

To explicitly recompile the library my\_ext\_lib owned by the sample user hr, issue this statement:

\[
\text{ALTER LIBRARY hr.my\_ext\_lib COMPILE;} \
\]

If the database encounters no compilation errors while recompiling my\_ext\_lib, then my\_ext\_lib becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling my\_ext\_lib results in compilation errors, then the database returns an error, and my\_ext\_lib remains invalid.
The database also invalidates all objects that depend upon my_ext_lib. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

Related Topics

- "CREATE LIBRARY Statement"
- "DROP LIBRARY Statement"

### 15.3 ALTER PACKAGE Statement

The ALTER PACKAGE statement explicitly recompiles a package specification, body, or both. Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

**Note:**

This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the "CREATE PACKAGE Statement", or the "CREATE PACKAGE BODY Statement" with the OR REPLACE clause.

**Topics**

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

**Prerequisites**

If the package is in the SYS schema, you must be connected as SYSDBA. Otherwise, the package must be in your schema or you must have ALTER ANY PROCEDURE system privilege.

**Syntax**

```
alter_package ::= 
```

```
<table>
<thead>
<tr>
<th>ALTER PACKAGE</th>
<th>schema .</th>
<th>package_name</th>
<th>package_compile_clause</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>EDITIONABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NONEDITIONABLE</td>
</tr>
</tbody>
</table>
```
package_compile_clause ::=  

ALTER PACKAGE emp_mgmt COMPILE PACKAGE;

If the database encounters no compilation errors while recompiling the emp_mgmt specification and body, then emp_mgmt becomes valid. The user hr can subsequently invoke or reference all package objects declared in the specification of emp_mgmt without runtime recompilation. If recompiling emp_mgmt results in compilation errors, then the database returns an error and emp_mgmt remains invalid.
The database also invalidates all objects that depend upon emp_mgmt. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

To recompile the body of the emp_mgmt package in the schema hr, issue this statement:

```
ALTER PACKAGE hr.emp_mgmt COMPILE BODY;
```

If the database encounters no compilation errors while recompiling the package body, then the body becomes valid. The user hr can subsequently invoke or reference all package objects declared in the specification of emp_mgmt without runtime recompilation. If recompiling the body results in compilation errors, then the database returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of emp_mgmt, the database does not invalidate dependent objects.

**Related Topics**

- "CREATE PACKAGE Statement"
- "DROP PACKAGE Statement"

### 15.4 ALTER PROCEDURE Statement

The ALTER PROCEDURE statement explicitly recompiles a standalone procedure.

Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the "ALTER PACKAGE Statement".

**Note:**

This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a standalone procedure, use the "CREATE PROCEDURE Statement" with the OR REPLACE clause.

The ALTER PROCEDURE statement is very similar to the ALTER FUNCTION statement. See "ALTER FUNCTION Statement" for more information.

**Topics**

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics
Prerequisites

If the procedure is in the \texttt{SYS} schema, you must be connected as \texttt{SYSDBA}. Otherwise, the procedure must be in your schema or you must have \texttt{ALTER ANY PROCEDURE} system privilege.

Syntax

\texttt{alter\_procedure ::=}

\begin{center}
\begin{tikzpicture}
  \node (alter) {\texttt{ALTER\ Procedure}};
  \node[below of=alter] (schema) {\texttt{\ .\ procedure\_name\ procedure\_compile\_clause\ .\ EDITIONABLE\ NONEDITIONABLE}};
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
  \node (compile) {\texttt{COMPILE\ \ DEBUG\ \ \ \ compiler\_parameters\_clause\ \ REUSE\ \ SETTINGS}};
\end{tikzpicture}
\end{center}

\begin{center}
\texttt{procedure\_compile\_clause ::=}
\end{center}

\begin{center}
\texttt{\texttt{(\texttt{compiler\_parameters\_clause ::=}\)}}
\end{center}

Semantics

\texttt{alter\_procedure}

\texttt{schema}

Name of the schema containing the procedure. \textbf{Default}: your schema.

\texttt{procedure\_name}

Name of the procedure to be recompiled.

\begin{center}
\texttt{\{ EDITIONABLE | NONEDITIONABLE \}}
\end{center}

Specifies whether the procedure becomes an editioned or noneditioned object if editioning is later enabled for the schema object type \texttt{PROCEDURE} in \texttt{schema}. \textbf{Default}: \texttt{EDITIONABLE}. For information about altering editioned and noneditioned objects, see \textit{Oracle Database Development Guide}.

\texttt{procedure\_compile\_clause}

See \texttt{compile\_clause} and \texttt{compiler\_parameters\_clause} semantics.

Example

\textbf{Example 15-4  Recompiling a Procedure}

To explicitly recompile the procedure \texttt{remove\_emp} owned by the user \texttt{hr}, issue this statement:

\begin{center}
\texttt{ALTER\ PROEDURE\ hr.remove\_emp\ COMPILE;}
\end{center}
If the database encounters no compilation errors while recompiling `remove_emp`, then `remove_emp` becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling `remove_emp` results in compilation errors, then the database returns an error and `remove_emp` remains invalid.

the database also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that invoke `remove_emp`. If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

Related Topics

- "CREATE PROCEDURE Statement"
- "DROP PROCEDURE Statement"

15.5 ALTER TRIGGER Statement

The **ALTER TRIGGER** statement enables, disables, compiles, or renames a database trigger.

**Note:**

This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the "CREATE TRIGGER Statement" with the **OR REPLACE** clause.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

If the trigger is in the **SYS** schema, you must be connected as **SYSDBA**. Otherwise, the trigger must be in your schema or you must have **ALTER ANY TRIGGER** system privilege.

In addition, to alter a trigger on **DATABASE**, you must have the **ADMINISTER DATABASE TRIGGER** system privilege.

**See Also:**

"CREATE TRIGGER Statement" for more information about triggers based on **DATABASE** triggers
Syntax

\[ alter\_trigger ::= \]

\[ trigger\_compile\_clause ::= \]

(\[ compiler\_parameters\_clause ::= \]

Semantics

\[ alter\_trigger \]

\[ schema \]
Name of the schema containing the trigger. **Default:** your schema.

\[ trigger\_name \]
Name of the trigger to be altered.

\[ [ ENABLE | DISABLE ] \]
Enables or disables the trigger.

\[ RENAME TO new\_name \]
Renames the trigger without changing its state.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the *_SOURCE static data dictionary views. As a result, comments and formatting may change in the TEXT column of those views even though the trigger source did not change.

\[ \{ EDITIONABLE | NONEDITIONABLE \} \]
Specifies whether the trigger becomes an editioned or noneditioned object if editioning is later enabled for the schema object type TRIGGER in schema. **Default:** EDITIONABLE. For information about altering editioned and noneditioned objects, see Oracle Database Development Guide.
Restriction on NONEDITIONABLE

You cannot specify NONEDITIONABLE for a crossedition trigger.

\textit{trigger\_compile\_clause}

Recompiles the trigger, whether it is valid or invalid.

See \textit{compile\_clause} and \textit{compiler\_parameters\_clause} semantics.

Examples

Example 15-5  Disabling Triggers

The sample schema hr has a trigger named \textit{update\_job\_history} created on the employees table. The trigger fires whenever an \textit{UPDATE} statement changes an employee's job_id. The trigger inserts into the job_history table a row that contains the employee's ID, begin and end date of the last job, and the job ID and department.

When this trigger is created, the database enables it automatically. You can subsequently disable the trigger with this statement:

\texttt{ALTER TRIGGER update\_job\_history DISABLE;}

When the trigger is disabled, the database does not fire the trigger when an \textit{UPDATE} statement changes an employee's job.

Example 15-6  Enabling Triggers

After disabling the trigger, you can subsequently enable it with this statement:

\texttt{ALTER TRIGGER update\_job\_history ENABLE;}

After you reenable the trigger, the database fires the trigger whenever an \textit{UPDATE} statement changes an employee's job. If an employee's job is updated while the trigger is disabled, then the database does not automatically fire the trigger for this employee until another transaction changes the job_id again.

Related Topics

In this chapter:

- "CREATE TRIGGER Statement"
- "DROP TRIGGER Statement"

In other chapters:

- "Trigger Compilation, Invalidation, and Recompilation"
- "Trigger Enabling and Disabling"

15.6 ALTER TYPE Statement

Use the ALTER TYPE statement to add or drop member attributes or methods. You can change the existing properties of an object type, and you can modify the scalar attributes of the type. You can also use this statement to recompile the specification or
body of the type or to change the specification of an object type by adding new object member subprogram specifications.

The **ALTER TYPE** statement does one of the following to a type that was created with "CREATE TYPE Statement" and "CREATE TYPE BODY Statement":

- **Evolves** the type; that is, adds or drops member attributes or methods.
  
  For more information about type evolution, see *Oracle Database Object-Relational Developer's Guide*.

- Changes the specification of the type by adding object member subprogram specifications.

- Recompiles the specification or body of the type.

- Resets the version of the type to 1, so that it is no longer considered to be evolved.

**Topics**

- **Prerequisites**
- **Syntax**
- **Semantics**
- **Examples**
- **Related Topics**

**Prerequisites**

If the type is in the **SYS** schema, you must be connected as **SYSDBA**. Otherwise, the type must be in your schema and you must have **CREATE TYPE** or **CREATE ANY TYPE** system privilege, or you must have **ALTER ANY TYPE** system privileges.

**Syntax**

```
alter_type ::= 
```

```plaintext
ALTER TYPE
  schema .
  type_name
  EDITIONABLE
  NONEDITIONABLE
alter_type_clause
;
```
**Chapter 15**

**ALTER TYPE Statement**

\[\text{alter_type_clause ::= }\]

\[
\text{type_compile_clause, type_replace_clause, alter_attribute_definition, alter_method_spec, alter_collection_clauses, dependent_handling_clause}
\]

\[
\text{type_compile_clause ::= COMPILE, DEBUG, SPECIFICATION, BODY, compiler_parameters_clause, REUSE, SETTINGS}
\]

\[
\text{type_replace_clause ::= REPLACE, invoker_rights_clause, accessible_by_clause, AS OBJECT, attribute, datatype, element_spec}
\]

\[
\text{accessible_by_clause ::=, invoker_rights_clause ::=, element_spec ::=}
\]
alter_method_spec ::= 

( map_order_function_spec ::=, subprogram_spec ::= )

alter_attribute_definition ::= 

alter_collections_clauses ::= 

dependent_handling_clause ::= 

exceptions_clause ::= 

Chapter 15
ALTER TYPE Statement
Semantics

**alter_type**

**schema**

Name of the schema containing the type. **Default:** your schema.

**type_name**

Name of an ADT, VARRAY type, or nested table type.

**Restriction on type_name**

You cannot evolve an editioned ADT.

The **ALTER TYPE** statement fails with ORA-22348 if either of the following is true:

- The type is an editioned ADT and the **ALTER TYPE** statement has no **type_compile_clause**.
  
  (You can use the **ALTER TYPE** statement to recompile an editioned object type, but not for any other purpose.)

- The type has a dependent that is an editioned ADT and the **ALTER TYPE** statement has a **CASCADE** clause.

An **editioned object** is a schema object that has an editionable object type and was created by a user for whom editions are enabled.

**{ EDITIONABLE | NONEDITIONABLE }**

Specifies whether the type becomes an editioned or noneditioned object if editioning is later enabled for the schema object type **TYPE** in **schema**. **Default:** EDITIONABLE. For information about altering editioned and noneditioned objects, see *Oracle Database Development Guide*.

**alter_type_clause**

**RESET**

 Resets the version of this type to 1, so that it is no longer considered to be evolved.

**Note:**

Resetting the version of this type to 1 invalidates all of its dependents.

**RESET** is intended for evolved ADTs that are preventing their owners from being editions-enabled. For information about enabling editions for users, see *Oracle Database Development Guide*.

To see the version number of an ADT, select **VERSION#** from the static data dictionary view ***_TYPE_VERSIONS**. For example:

```
SELECT Version#
FROM DBA_TYPE_VERSIONS
WHERE Owner = schema
```
AND Name = 'type_name'
AND Type = 'TYPE'

For an evolved ADT, the preceding query returns multiple rows with different version numbers. RESET deletes every row whose version number is less than the maximum version number, and resets the version number of the remaining rows to 1.

Restriction on RESET
You cannot specify RESET if the type has any table dependents (direct or indirect).

[NOT] INSTANTIABLE
Specify INSTANTIABLE if object instances of this type can be constructed.
Specify NOT INSTANTIABLE if no constructor (default or user-defined) exists for this type. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement).

Restriction on NOT INSTANTIABLE
You cannot change a user-defined type from INSTANTIABLE to NOT INSTANTIABLE if the type has any table dependents.

[NOT] FINAL
Specify FINAL if no further subtypes can be created for this type.
Specify NOT FINAL if further subtypes can be created under this type.
If you change the property from FINAL to NOT FINAL, or the reverse, then you must specify the CASCADE clause of the "dependent_handling_clause" to convert data in dependent columns and tables. Specifically:

• If you change a type from NOT FINAL to FINAL, then you must specify CASCADE [INCLUDING TABLE DATA]. You cannot defer data conversion with CASCADE NOT INCLUDING TABLE DATA.
• If you change a type from FINAL to NOT FINAL, then:
  – Specify CASCADE INCLUDING TABLE DATA if you want to create substitutable tables and columns of that type, but you are not concerned about the substitutability of the existing dependent tables and columns.
    The database marks all existing dependent columns and tables NOT SUBSTITUTABLE AT ALL LEVELS, so you cannot insert the subtype instances of the altered type into these existing columns and tables.
  – Specify CASCADE CONVERT TO SUBSTITUTABLE if you want to create substitutable tables and columns of the type and also store subtype instances of the altered type in existing dependent tables and columns.
    The database marks all existing dependent columns and tables SUBSTITUTEABLE AT ALL LEVELS except those that are explicitly marked NOT SUBSTITUTEABLE AT ALL LEVELS.
Restriction on FINAL

You cannot change a user-defined type from NOT FINAL to FINAL if the type has any subtypes.

**type_compile_clause**

*(Default)* Recompiles the type specification and body.

See `compile_clause` and `compiler_parameters_clause` semantics.

**type_replace_clause**

Starting with Oracle Database 12c Release 2 (12.2), the `type_replace_clause` is deprecated. Use the `alter_method_spec` clause instead. Alternatively, you can recreate the type using the `CREATE OR REPLACE TYPE` statement.

Adds member subprogram specifications.

Restriction on `type_replace_clause`

This clause is valid only for ADTs, not for nested tables or varrays.

**attribute**

Name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object.

**element_spec**

Specifies elements of the redefined object.

**alter_method_spec**

Adds a method to or drops a method from the type. The database disables any function-based indexes that depend on the type.

In one `ALTER TYPE` statement you can add or drop multiple methods, but you can reference each method only once.

**ADD**

When you add a method, its name must not conflict with any existing attributes in its type hierarchy.

**DROP**

When you drop a method, the database removes the method from the target type.

Restriction on DROP

You cannot drop from a subtype a method inherited from its supertype. Instead you must drop the method from the supertype.
**alter_attribute_definition**

Adds, drops, or modifies an attribute of an ADT. In one **ALTER TYPE** statement, you can add, drop, or modify multiple member attributes or methods, but you can reference each attribute or method only once.

**ADD ATTRIBUTE**

Name of the attribute must not conflict with existing attributes or methods in the type hierarchy. The database adds the attribute to the end of the locally defined attribute list.

If you add the attribute to a supertype, then it is inherited by all of its subtypes. In subtypes, inherited attributes always precede declared attributes. Therefore, you might need to update the mappings of the implicitly altered subtypes after adding an attribute to a supertype.

**DROP ATTRIBUTE**

When you drop an attribute from a type, the database drops the column corresponding to the dropped attribute and any indexes, statistics, and constraints referencing the dropped attribute.

You need not specify the data type of the attribute you are dropping.

**Restrictions on DROP ATTRIBUTE**

- You cannot drop an attribute inherited from a supertype. Instead you must drop the attribute from the supertype.
- You cannot drop an attribute that is part of a partitioning, subpartitioning, or cluster key.

⚠️ **Caution:**

If you use the **INVALIDATE** option, then the compiler does not check dependents; therefore, this rule is not enforced. However, dropping such an attribute leaves the table in an unusable state.

- You cannot drop an attribute of a primary-key-based object identifier of an object table or a primary key of an index-organized table.
- You cannot drop all of the attributes of a root type. Instead you must drop the type. However, you can drop all of the locally declared attributes of a subtype.

**MODIFY ATTRIBUTE**

Modifies the data type of an existing scalar attribute. For example, you can increase the length of a **VARCHAR2** or **RAW** attribute, or you can increase the precision or scale of a numeric attribute.

**Restriction on MODIFY ATTRIBUTE**

You cannot expand the size of an attribute referenced in a function-based index, domain index, or cluster key.
**alter_collection_clauses**

These clauses are valid only for collection types.

**MODIFY LIMIT integer**

Increases the number of elements in a varray. It is not valid for nested tables. Specify an integer greater than the current maximum number of elements in the varray.

**MODIFY ELEMENT TYPE datatype**

Increases the precision, size, or length of a scalar data type of a varray or nested table. This clause is not valid for collections of ADTs.

- For a collection of NUMBER, you can increase the precision or scale.
- For a collection of RAW, you can increase the maximum size.
- For a collection of VARCHAR2 or NVARCHAR2, you can increase the maximum length.

**dependent_handling_clause**

Specifies how the database is to handle objects that are dependent on the modified type. If you omit this clause, then the ALTER TYPE statement terminates if the type has any dependent type or table.

**INVALIDATE**

Invalidates all dependent objects without any checking mechanism. Starting with Oracle Database 12c Release 2 (12.2), the INVALIDATE command is deprecated. Oracle recommends that you use the CASCADE clause instead.

---

**Caution:**

The database does not validate the type change, so use this clause with caution. For example, if you drop an attribute that is a partitioning or cluster key, then the table becomes unusable.

**CASCADE**

Propagates the type change to dependent types and tables. The database terminates the statement if any errors are found in the dependent types or tables unless you also specify FORCE.

If you change the property of the type between FINAL and NOT FINAL, then you must specify this clause to convert data in dependent columns and tables.

**INCLUDING TABLE DATA**

*(Default)* Converts data stored in all user-defined columns to the most recent version of the column type.
Note:
You must specify this clause if your column data is in Oracle database version 8.0 image format. This clause is also required if you are changing the type property between FINAL and NOT FINAL.

- For each attribute added to the column type, the database adds an attribute to the data and initializes it to null.
- For each attribute dropped from the referenced type, the database removes the corresponding attribute data from each row in the table.

If you specify INCLUDING TABLE DATA, then all of the tablespaces containing the table data must be in read/write mode.

If you specify NOT INCLUDING TABLE DATA, then the database upgrades the metadata of the column to reflect the changes to the type but does not scan the dependent column and update the data as part of this ALTER TYPE statement. However, the dependent column data remains accessible, and the results of subsequent queries of the data reflect the type modifications.

CONVERT TO SUBSTITUTABLE
Specify this clause if you are changing the type from FINAL to NOT FINAL and you want to create substitutable tables and columns of the type and also store subtype instances of the altered type in existing dependent tables and columns.

exceptions_clause
FORCED
Specify FORCE if you want the database to ignore the errors from dependent tables and indexes and log all errors in the specified exception table. The exception table must have been created by running the DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE procedure.

Examples
See "CREATE TYPE Statement" for examples creating the types referenced in these examples.

Example 15-7 Adding a Member Function
This example uses the ADT data_typ1.

A method is added to data_typ1 and its type body is modified to correspond. The date formats are consistent with the order_date column of the oe.orders sample table.

```sql
ALTER TYPE data_typ1
  ADD MEMBER FUNCTION qtr(der_qtr DATE)
  RETURN CHAR CASCADE;

CREATE OR REPLACE TYPE BODY data_typ1 IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
    BEGIN
      RETURN (year + invent);
    END;
    MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
```

Chapter 15
ALTER TYPE Statement

15-21
BEGIN
    IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
        RETURN 'FIRST';
    ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
        RETURN 'SECOND';
    ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
        RETURN 'THIRD';
    ELSE
        RETURN 'FOURTH';
    END IF;
END;
/

Example 15-8 Adding a Collection Attribute

This example adds the `author` attribute to the `textdoc_tab` object column of the `text` table.

CREATE TABLE text (doc_id NUMBER, description textdoc_tab)
    NESTED TABLE description STORE AS text_store;

ALTER TYPE textdoc_typ
    ADD ATTRIBUTE (author VARCHAR2) CASCADE;

The `CASCADE` keyword is required because both the `textdoc_tab` and `text` table are dependent on the `textdoc_typ` type.

Example 15-9 Increasing the Number of Elements of a Collection Type

This example increases the maximum number of elements in the varray `phone_list_typ_demo`.

ALTER TYPE phone_list_typ_demo
    MODIFY LIMIT 10 CASCADE;

Example 15-10 Increasing the Length of a Collection Type

This example increases the length of the varray element type `phone_list_typ`.

ALTER TYPE phone_list_typ
    MODIFY ELEMENT TYPE VARCHAR(64) CASCADE;

Example 15-11 Recompiling a Type

This example recompiles type `cust_address_typ` in the `hr` schema.

ALTER TYPE cust_address_typ2 COMPILE;

Example 15-12 Recompiling a Type Specification

This 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
   BEGIN
     dbms_output.put_line(c1);
     RETURN c1;
   END;
END;
/

In this example, both the specification and body of link2 are invalidated because
link1, which is an attribute of link2, is altered.

ALTER TYPE link1 ADD ATTRIBUTE (b NUMBER) INVALIDATE;

You must recompile the type by recompiling the specification and body in separate
statements:

ALTER TYPE link2 COMPILE SPECIFICATION;

ALTER TYPE link2 COMPILE BODY;

Alternatively, you can compile both specification and body at the same time:

ALTER TYPE link2 COMPILE;

Example 15-13  Evolving and Resetting an ADT

This example creates an ADT in the schema Usr, evolves that ADT, and then tries to
enable editions for Usr, which fails.

Then the example resets the version of the ADT to 1 and succeeds in enabling
editions for Usr. To show the version numbers of the newly created, evolved, and reset
ADT, the example uses the static data dictionary view DBA_TYPE_VERSIONS.

-- Create ADT in schema Usr:
create type Usr.My_ADT authid Definer is object(a1 number)

-- Show version number of ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/

Result:

T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)

1 row selected.

-- Evolve ADT:
alter type Usr.My_ADT add attribute (a2 number)
/

-- Show version number of evolved ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADТ'
/

Result:

+---------------------------------------------------------------------+
|   type     My_ADТ authid Definer is object(a1 number)               |
| 1
| type     My_ADТ authid Definer is object(a1 number)               |
| 2
| alter type     My_ADТ add attribute (a2 number)                   |
| 2
+---------------------------------------------------------------------+

3 rows selected.

-- Try to enable editions for Usr:
alter user Usr enable editions
/

Result:

alter user Usr enable editions
*
ERROR at line 1:
ORA-38820: user has evolved object type

-- Reset version of ADТ to 1:
alter type Usr.My_ADТ reset
/

-- Show version number of reset ADТ:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADТ'
/

Result:

+---------------------------------------------------------------------+
|   type     My_ADТ authid Definer is object(a1 number)               |
| 1
| alter type     My_ADТ add attribute (a2 number)                   |
| 1
+---------------------------------------------------------------------+

2 rows selected.

-- Try to enable editions for Usr:
alter user Usr enable editions
/

Result:
15.7 CREATE FUNCTION Statement

The `CREATE FUNCTION` statement creates or replaces a standalone function or a call specification.

A **standalone function** is a function (a subprogram that returns a single value) that is stored in the database.

Note:

A standalone function that you create with the `CREATE FUNCTION` statement differs from a function that you declare and define in a PL/SQL block or package. For more information, see "Function Declaration and Definition" and `CREATE PACKAGE` Statement.

A **call specification** declares a Java method or a C function so that it can be invoked from PL/SQL. You can also use the SQL `CALL` statement to invoke such a method or subprogram. The call specification tells the database which Java method, or which named function in which shared library, to invoke when an invocation is made. It also tells the database what type conversions to make for the arguments and return value.

Note:

To be callable from SQL statements, a stored function must obey certain rules that control side effects. See "Subprogram Side Effects".

Topics

- Prerequisites
- Syntax
- Semantics
Prerequisites

To create or replace a standalone function in your schema, you must have the `CREATE PROCEDURE` system privilege.

To create or replace a standalone function in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

To invoke a call specification, you may need additional privileges, for example, `EXECUTE` privileges on a C library for a C call specification.

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.

See Also:

For more information about such prerequisites:

- *Oracle Database Development Guide*
- *Oracle Database Java Developer's Guide*

Syntax

```
create_function ::= CREATE OR REPLACE EDICTIONABLE NONEDITABLE FUNCTION plsql_function_source
```
Semantics

`create_function` OR REPLACE

Re-creates the function if it exists, and recompiles it.

Users who were granted privileges on the function before it was redefined can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, then the database marks the indexes `DISABLED`.

[EDITIONABLE | NONEDITIONABLE ]

Specifies whether the function is an editioned or noneditioned object if editioning is enabled for the schema object type `FUNCTION` in `schema`. Default: EDITIONABLE. For information about editioned and noneditioned objects, see Oracle Database Development Guide.
**plsql_function_source**

**schema**

Name of the schema containing the function. **Default:** your schema.

**function_name**

Name of the function to be created.

**Note:**

If you plan to invoke a stored subprogram using a stub generated by SQL*Module, then the stored subprogram name must also be a legal identifier in the invoking host 3GL language, such as Ada or C.

**RETURN datatype**

For **datatype**, specify the data type of the return value of the function. The return value can have any data type supported by PL/SQL.

**Note:**

Oracle SQL does not support invoking functions with **BOOLEAN** parameters or returns. Therefore, for SQL statements to invoke your user-defined functions, you must design them to return numbers (0 or 1) or character strings (‘TRUE’ or ‘FALSE’).

The data type cannot specify a length, precision, or scale. The database derives the length, precision, or scale of the return value from the environment from which the function is called.

If the return type is **ANYDATASET** and you intend to use the function in the **FROM** clause of a query, then you must also specify the **PIPELINED** clause and define a describe method (**ODCITableDescribe**) as part of the implementation type of the function.

You cannot constrain this data type (with **NOT NULL**, for example).

**body**

The required executable part of the function and, optionally, the exception-handling part of the function.

**declare_section**

The optional declarative part of the function. Declarations are local to the function, can be referenced in **body**, and cease to exist when the function completes execution.

**Examples**

**Example 15-14 Creating a Function**

This statement creates the function **get_bal** on the sample table **oe.orders**.
CREATE FUNCTION get_bal(acc_no IN NUMBER) RETURN NUMBER IS acc_bal NUMBER(11,2);
BEGIN
  SELECT order_total INTO acc_bal
  FROM orders
  WHERE customer_id = acc_no;
  RETURN(acc_bal);
END;
/

The `get_bal` function returns the balance of a specified account.

When you invoke the function, you must specify the argument `acc_no`, the number of the account whose balance is sought. The data type of `acc_no` is `NUMBER`.

The function returns the account balance. The `RETURN` clause of the `CREATE FUNCTION` statement specifies the data type 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 `orders` table. The function uses a `RETURN` statement to return this value to the environment in which the function is called.

The function created in the preceding example can be used in a SQL statement. For example:

```
SELECT get_bal(165) FROM DUAL;
```

```
GET_BAL(165)
------------
   2519
```

### Example 15-15 Creating Aggregate Functions

The next statement creates an aggregate function called `SecondMax` to aggregate over `number` values. It assumes that the ADT `SecondMaxImpl` subprograms contains the implementations of the ODCIAggregate subprograms:

```
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

### See Also:

*Oracle Database Data Cartridge Developer’s Guide* for the complete implementation of type and type body for `SecondMaxImpl`.

Use such an aggregate function in a query like this statement, which queries the sample table `hr.employees`:

```
SELECT SecondMax(salary) "SecondMax", department_id
FROM employees
GROUP BY department_id
HAVING SecondMax(salary) > 9000
ORDER BY "SecondMax", department_id;
```

```
SecondMax DEPARTMENT_ID
```

15-29
Example 15-16  Package Procedure in a Function

This statement creates a function that uses a `DBMS_LOB.GETLENGTH` procedure to return the length of a `CLOB` column.

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
    RETURN NUMBER DETERMINISTIC IS
BEGIN
    RETURN DBMS_LOB.GETLENGTH(a);
END;
```

Related Topics

In this chapter:

- "ALTER FUNCTION Statement"
- "CREATE PROCEDURE Statement"
- "DROP FUNCTION Statement"

In other chapters:

- "Overview of Polymorphic Table Functions"
- "Function Declaration and Definition" for information about creating a function in a PL/SQL block
- "Formal Parameter Declaration"
- "PL/SQL Subprograms"

In other books:

- Oracle Database SQL Language Reference for information about the `CALL` statement
- Oracle Database Development Guide for information about restrictions on user-defined functions that are called from SQL statements
- Oracle Database Development Guide for more information about call specifications
- Oracle Database Data Cartridge Developer's Guide for information about defining the `ODCITableDescribe` function
15.8 CREATE LIBRARY Statement

The CREATE LIBRARY statement creates a library, which is a schema object associated with an operating-system shared library.

Note:
The CREATE LIBRARY statement is valid only on platforms that support shared libraries and dynamic linking.

For instructions for creating an operating-system shared library, or DLL, see Oracle Database Development Guide.

You can use the name of the library schema object in the call_spec of CREATE FUNCTION or CREATE PROCEDURE statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can invoke third-generation-language (3GL) functions and procedures.

Topics

• Prerequisites
• Syntax
• Semantics
• Examples
• Related Topics

Prerequisites

To create a library in your schema, you must have the CREATE LIBRARY system privilege. To create a library in another user's schema, you must have the CREATE ANY LIBRARY system privilege.

To create a library that is associated with a DLL in a directory object, you must have the EXECUTE object privilege on the directory object.

To create a library that is associated with a credential name, you must have the EXECUTE object privilege on the credential name.

To use the library in the call_spec of a CREATE FUNCTION statement, or when declaring a function in a package or type, you must have the EXECUTE object privilege on the library and the CREATE FUNCTION system privilege.

To use the library in the call_spec of a CREATE PROCEDURE statement, or when declaring a procedure in a package or type, you must have the EXECUTE object privilege on the library and the CREATE PROCEDURE system privilege.

To execute a procedure or function defined with the call_spec (including a procedure or function defined within a package or type), you must have the EXECUTE object privilege on the procedure or function (but you do not need the EXECUTE object privilege on the library).
Syntax

```
create_library ::= CREATE [OR REPLACE] [EDITIONABLE | NONEDITIONABLE] LIBRARY plsql_library_source
```

```
plsql_library_source ::= schema . library_name [sharing_clause] IS AS 'full_path_name' 'file_name' IN directory_object
AGENT agent_dblink CREDENTIAL schema . credential_name;
```

Semantics

**create_library**

**OR REPLACE**

Re-creates the library if it exists, and recompiles it.

Users who were granted privileges on the library before it was redefined can still access it without being regranted the privileges.

**[ EDITIONABLE | NONEDITIONABLE ]**

Specifies whether the library is an editioned or noneditioned object if editioning is enabled for the schema object type **LIBRARY** in **schema**. Default: EDITIONABLE. For information about editioned and noneditioned objects, see Oracle Database Development Guide.

**plsql_library_source**

**schema**

Name of the schema containing the library. Default: your schema.

**library_name**

Name that represents this library when a user declares a function or procedure with a call_spec.

**'full_path_name'**

String literal enclosed in single quotation marks, whose value your operating system recognizes as the full path name of a shared library.
The full_path_name is not interpreted during execution of the CREATE LIBRARY statement. The existence of the shared library is checked when someone invokes one of its subprograms.

‘file_name’ IN directory_object

The file_name is a string literal enclosed in single quotation marks, whose value is the name of a dynamic link library (DLL) in directory_object. The string literal cannot exceed 2,000 bytes and cannot contain path delimiters. The compiler ignores file_name, but at run time, file_name is checked for path delimiters.

directory_object

The directory_object is a directory object, created with the CREATE DIRECTORY statement (described in Oracle Database SQL Language Reference). If directory_object does not exist or you do not have the EXECUTE object privilege on directory_object, then the library is created with errors. If directory_object is subsequently created, then the library becomes invalid. Other reasons that the library can become invalid are:

- directory_object is dropped.
- directory_object becomes invalid.
- Your EXECUTE object privilege on directory_object is revoked.

If you create a library object in a PDB that has a predefined PATH_PREFIX, the library must use a directory object. The directory object will enforce the rules of PATH_PREFIX for the library object. Failure to use a directory object in the library object will raise a compilation error.

If a database is plugged into a CDB as a PDB with a predefined PATH_PREFIX, attempts to use a library object that does not use a directory object result in an ORA-65394 error. The library object will not be invalidated, but to make it usable, you must recreate it using a directory object. See Oracle Multitenant Administrator’s Guide for more information about CDB administration.

AGENT ‘agent_dblink’

Causes external procedures to run from a database link other than the server. Oracle Database uses the database link that agent_dblink specifies to run external procedures. If you omit this clause, then the default agent on the server (extproc) runs external procedures.

CREDENTIAL [schema.]credential_name

Specifies the credentials of the operating system user that the extproc agent impersonates when running an external subprogram that specifies the library. Default: Owner of the Oracle Database installation.

If credential_name does not exist or you do not have the EXECUTE object privilege on credential_name, then the library is created with errors. If credential_name is subsequently created, then the library becomes invalid. Other reasons that the library can become invalid are:

- credential_name is dropped.
- credential_name becomes invalid.
- Your EXECUTE object privilege on credential_name is revoked.
For information about using credentials, see *Oracle Database Security Guide*.

**Examples**

**Example 15-17  Creating a Library**

The following statement creates library `ext_lib`, using a directory object:

```sql
CREATE LIBRARY ext_lib AS 'ddl_1' IN ddl_dir;
/
```

The following statement re-creates library `ext_lib`, using a directory object and a credential:

```sql
CREATE OR REPLACE LIBRARY ext_lib AS 'ddl_1' IN ddl_dir CREDENTIAL ddl_cred;
/
```

The following statement creates library `ext_lib`, using an explicit path:

```sql
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
/
```

The following statement re-creates library `ext_lib`, using an explicit path:

```sql
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
/
```

**Example 15-18  Specifying an External Procedure Agent**

The following example creates a library `app_lib` (using an explicit path) and specifies that external procedures run from the public database `sales.hq.example.com`:

```sql
CREATE LIBRARY app_lib as '${ORACLE_HOME}/lib/app_lib.so'
    AGENT 'sales.hq.example.com';
/
```

**See Also:**

*Oracle Database SQL Language Reference* for information about creating database links

**Related Topics**

- "ALTER LIBRARY Statement"
- "DROP LIBRARY Statement"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"
15.9 CREATE PACKAGE Statement

The CREATE PACKAGE statement creates or replaces the specification for a stored package, which is an encapsulated collection of related procedures, functions, and other program objects stored as a unit in the database.

The package specification declares these objects. The package body, specified subsequently, defines these objects.

Topics

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

To create or replace a package in your schema, you must have the CREATE PROCEDURE system privilege. To create or replace a package in another user's schema, you must have the CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE statement inside an Oracle database precompiler program, you must terminate the statement with the keyword END–EXEC followed by the embedded SQL statement terminator for the specific language.

Syntax

create_package ::=  

plsql_package_source ::=  

CREATE OR REPLACE EDITIONABLE NONEDITIONABLE PACKAGE plsql_package_source

plsql_package_source ::=  

schema . package_name sharing_clause default_collation_clause invoker_rights_clause accessible_by_clause IS AS package_item_list END package_name ;
( sharingClause ::= , defaultCollationClause ::= , invokerRightsClause ::= , accessibleByClause ::= ,)

packageItemList ::= 

( cursorDeclaration ::= , itemDeclaration ::= , typeDefinition ::= )

packageFunctionDeclaration ::= 

( functionHeading ::= , accessibleByClause ::= , deterministicClause ::= , pipelinedClause ::= , parallelEnableClause ::= , resultCacheClause ::= )

packageProcedureDeclaration ::= 

( procedureHeading ::= , accessibleByClause ::= )

Semantics

create_package

OR REPLACE

Re-creates the package if it exists, and recompiles it.

Users who were granted privileges on the package before it was redefined can still access the package without being regranted the privileges.
If any function-based indexes depend on the package, then the database marks the indexes **DISABLED**.

**[ EDITIONABLE | NONEDITIONABLE ]**

Specifies whether the package is an editioned or noneditioned object if editioning is enabled for the schema object type **PACKAGE** in **schema**. **Default:** EDITIONABLE. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

**plsql_package_source**

**schema**

Name of the schema containing the package. **Default:** your schema.

**package_name**

A package stored in the database. For naming conventions, see "Identifiers".

**package_item_list**

Defines every type in the package and declares every cursor and subprogram in the package. Except for polymorphic table functions, every declaration must have a corresponding definition in the package body. The headings of corresponding declarations and definitions must match word for word, except for white space. Package polymorphic table function must be declared in the same package as their implementation package.

**Restriction on package_item_list**

PRAGMA AUTONOMOUS_TRANSACTION cannot appear here.

**Example**

**Example 15-19  Creating the Specification for the emp_mgmt Package**

This statement creates the specification of the **emp_mgmt** package.

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
  FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2,
                 manager_id NUMBER, salary NUMBER,
                 commission_pct NUMBER, department_id NUMBER)
           RETURN NUMBER;
  FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
           RETURN NUMBER;
  PROCEDURE remove_emp(employee_id NUMBER);
  PROCEDURE remove_dept(department_id NUMBER);
  PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER);
  PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER);
  no_comm EXCEPTION;
  no_sal EXCEPTION;
END emp_mgmt;
```

The specification for the **emp_mgmt** package declares these public program objects:

- The functions **hire** and **create_dept**
- The procedures **remove_emp**, **remove_dept**, **increase_sal**, **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 invoke any of these public procedures or functions or raise any of the public exceptions of the package.

Before you can invoke 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 Statement".

Related Topics
In this chapter:

- "ALTER PACKAGE Statement"
- "CREATE PACKAGE Statement"
- "CREATE PACKAGE BODY Statement"
- "DROP PACKAGE Statement"

In other chapters:

- "PL/SQL Packages"
- "Package Specification"
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"

15.10 CREATE PACKAGE BODY Statement

The CREATE PACKAGE BODY statement creates or replaces the body of a stored package, which is an encapsulated collection of related procedures, stored functions, and other program objects stored as a unit in the database.

The package body defines these objects. The package specification, defined in an earlier CREATE PACKAGE statement, declares these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

To create or replace a package in your schema, you must have the CREATE PROCEDURE system privilege. To create or replace a package in another user's schema, you must have the CREATE ANY PROCEDURE system privilege. In both cases, the package body must be created in the same schema as the package.
To embed a CREATE PACKAGE BODY statement inside an the database precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

Syntax

create_package_body ::= 

plsql_package_body_source ::= 

initialize_section ::= 

Semantics

create_package_body

OR REPLACE

Re-creates the package body if it exists, and recompiles it.

Users who were granted privileges on the package body before it was redefined can still access the package without being regranted the privileges.

[ EDITIONABLE | NONEDITIONABLE ]

If you do not specify this property, then the package body inherits EDITIONABLE or NONEDITIONABLE from the package specification. If you do specify this property, then it must match that of the package specification.
**plsql_package_body_source**

**schema**
Name of the schema containing the package. **Default**: your schema.

**package_name**
Name of the package to be created.

**declare_section**
Has a definition for every cursor and subprogram declaration in the package specification. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

Can also declare and define private items that can be referenced only from inside the package.

**Restriction on declare_section**
**PRAGMA AUTONOMOUS_TRANSACTION** cannot appear here.

**initialize_section**
Initializes variables and does any other one-time setup steps.

**Examples**

**Example 15-20  Creating the emp_mgmt Package Body**
This statement creates the body of the emp_mgmt package created in "Example 15-19".

```sql
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
    tot_emps NUMBER;
    tot_depts NUMBER;
    FUNCTION hire
        (last_name VARCHAR2, job_id VARCHAR2,
         manager_id NUMBER, salary NUMBER,
         commission_pct NUMBER, department_id NUMBER)
        RETURN NUMBER IS new_empno NUMBER;
    BEGIN
        SELECT employees_seq.NEXTVAL
        INTO new_empno
        FROM DUAL;
        INSERT INTO employees
        VALUES (new_empno, 'First', 'Last', 'first.example@example.com',
                '(415)555-0100',
                TO_DATE('18-JUN-2002','DD-MON-YYYY'),
                'IT_PROG', 90000000, 00, 100, 110);
        tot_emps := tot_emps + 1;
        RETURN(new_empno);
    END;
    FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
    RETURN NUMBER IS new_deptno NUMBER;
    BEGIN
        SELECT departments_seq.NEXTVAL
        INTO new_deptno
        FROM DUAL;
    END;
END;
```
FROM dual;
INSERT INTO departments
VALUES (new_deptno, 'department name', 100, 1700);
tot_depts := tot_depts + 1;
RETURN(new_deptno);
END;

PROCEDURE remove_emp (employee_id NUMBER) IS
BEGIN
DELETE FROM employees
WHERE employees.employee_id = remove_emp.employee_id;
tot_emps := tot_emps - 1;
END;

PROCEDURE remove_dept (department_id NUMBER) IS
BEGIN
DELETE FROM departments
WHERE departments.department_id = remove_dept.department_id;
tot_depts := tot_depts - 1;
SELECT COUNT(*) INTO tot_emps FROM employees;
END;

PROCEDURE increase_sal (employee_id NUMBER, salary_incr NUMBER) IS
BEGIN
SELECT salary INTO curr_sal FROM employees
WHERE employees.employee_id = increase_sal.employee_id;
IF curr_sal IS NULL
THEN RAISE no_sal;
ELSE
UPDATE employees
SET salary = salary + salary_incr
WHERE employee_id = employee_id;
END IF;
END;

PROCEDURE increase_comm (employee_id NUMBER, comm_incr NUMBER) IS
BEGIN
SELECT commission_pct
INTO curr_comm
FROM employees
WHERE employees.employee_id = increase_comm.employee_id;
IF curr_comm IS NULL
THEN RAISE no_comm;
ELSE
UPDATE employees
SET commission_pct = commission_pct + comm_incr;
END IF;
END;
END emp_mgmt;

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 invokes the increase_comm procedure.
These objects are defined in the package body, so you can change their definitions without causing the database to invalidate dependent schema objects. For example, if you subsequently change the definition of `hire`, then the database need not recompile `increase_all_comms` before running 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

In this chapter:
- "CREATE PACKAGE Statement"

In other chapters:
- "PL/SQL Packages"
- "Package Body"
- "Function Declaration and Definition"
- "Procedure Declaration and Definition"

15.11 CREATE PROCEDURE Statement

The `CREATE PROCEDURE` statement creates or replaces a standalone procedure or a call specification.

A standalone procedure is a procedure (a subprogram that performs a specific action) that is stored in the database.

A call specification declares a Java method or a C language procedure so that it can be called from PL/SQL. You can also use the SQL `CALL` statement to invoke such a method or subprogram. The call specification tells the database which Java method, or which named procedure in which shared library, to invoke when an invocation is made. It also tells the database what type conversions to make for the arguments and return value.

Topics
- Prerequisites
- Syntax
Prerequisites

To create or replace a standalone procedure in your schema, you must have the `CREATE PROCEDURE` system privilege. To create or replace a standalone procedure in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege.

To invoke a call specification, you may need additional privileges, for example, the `EXECUTE` object privilege on the C library for a C call specification.

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.

Syntax

```
create_procedure ::= 
```

```
plsql_procedure_source ::= 
```

```
( sharing_clause ::=, default_collation_clause ::=, invoker_rights_clause ::=, accessible_by_clause ::=, call_spec ::=, body ::=, declare_section ::=, parameter_declaration ::= )
```

Semantics

```
create_procedure
```

```
OR REPLACE
```

Re-creates the procedure if it exists, and recompiles it.
Users who were granted privileges on the procedure before it was redefined can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the procedure, then the database marks the indexes **DISABLED**.

**[ EDITIONABLE | NONEDITIONABLE ]**

Specifies whether the procedure is an editioned or noneditioned object if editioning is enabled for the schema object type **PROCEDURE** in **schema**. **Default**: EDITIONABLE. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

**plsql_procedure_source**

**schema**

Name of the schema containing the procedure. **Default**: your schema.

**procedure_name**

Name of the procedure to be created.

---

**Note:**

If you plan to invoke a stored subprogram using a stub generated by SQL*Module, then the stored subprogram name must also be a legal identifier in the invoking host 3GL language, such as Ada or C.

**body**

The required executable part of the procedure and, optionally, the exception-handling part of the procedure.

**declare_section**

The optional declarative part of the procedure. Declarations are local to the procedure, can be referenced in **body**, and cease to exist when the procedure completes execution.

**Examples**

**Example 15-21  Creating a Procedure**

This statement creates the procedure **remove_emp** in the schema **hr**.

```sql
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
    tot_emps NUMBER;
    BEGIN
        DELETE FROM employees
        WHERE employees.employee_id = remove_emp.employee_id;
        tot_emps := tot_emps - 1;
        END;
/
```

The **remove_emp** procedure removes a specified employee. When you invoke the procedure, you must specify the **employee_id** of the employee to be removed.
The procedure uses a **DELETE** statement to remove from the `employees` table the row of `employee_id`.

### See Also:

"**CREATE PACKAGE BODY Statement**" to see how to incorporate this procedure into a package

#### Example 15-22  Creating an External Procedure

In this example, external procedure `c_find_root` expects a pointer as a parameter. Procedure `find_root` passes the parameter by reference using the **BY REFERENCE** phrase.

```sql
CREATE PROCEDURE find_root
  ( x IN REAL )
IS LANGUAGE C
  NAME c_find_root
  LIBRARY c_utils
  PARAMETERS ( x BY REFERENCE );
```

### Related Topics

In this chapter:

- "**ALTER PROCEDURE Statement**"
- "**CREATE FUNCTION Statement**"
- "**DROP PROCEDURE Statement**"

In other chapters:

- "**Formal Parameter Declaration**"
- "**Procedure Declaration and Definition**"
- "**PL/SQL Subprograms**"

In other books:

- *Oracle Database SQL Language Reference* for information about the **CALL** statement
- *Oracle Database Development Guide* for more information about call specifications
- *Oracle Database Development Guide* for more information about invoking stored PL/SQL subprograms

### 15.12 CREATE TRIGGER Statement

The **CREATE TRIGGER** statement creates or replaces a **database trigger**, which is either of these:

- A stored PL/SQL block associated with a table, a view, a schema, or the database
- An anonymous PL/SQL block or an invocation of a procedure implemented in PL/SQL or Java
The database automatically runs a trigger when specified conditions occur.

Topics
• Prerequisites
• Syntax
• Semantics
• Examples
• Related Topics

Prerequisites
• To create a trigger in your schema on a table in your schema or on your schema (SCHEMA), you must have the CREATE TRIGGER system privilege.
• To create a trigger in any schema on a table in any schema, or on another user’s schema (schema.SCHEMA), you must have the CREATE ANY TRIGGER system privilege.
• In addition to the preceding privileges, to create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.
• To create a trigger on a pluggable database (PDB), you must be connected to that PDB and have the ADMINISTER DATABASE TRIGGER system privilege. For information about PDBs, see Oracle Database Administrator’s Guide.
• In addition to the preceding privileges, to create a crossedition trigger, you must be enabled for editions. For information about enabling editions for a user, see Oracle Database Development Guide.

If the trigger issues SQL statements or invokes procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.

Syntax

\[
\text{create}\_\text{trigger} ::= \\
\]

```sql
CREATE OR REPLACE EDITIONABLE NONEDITIONABLE TRIGGER \text{plsql}\_\text{trigger}\_\text{source}
```

\[
\text{plsql}\_\text{trigger}\_\text{source} ::= \\
\]

```sql
\text{schema} . \text{trigger}\_\text{name} \text{sharing}\_\text{clause} \text{default}\_\text{collation}\_\text{clause}
```
(sharing_clause ::= , default_collation_clause ::= , compound_dml_trigger ::= , instead_of_dml_trigger ::= , system_trigger ::= )

simple_dml_trigger ::= 

BEFORE  
AFTER  
dml_event_clause 
referencing_clause 
FOR  
EACH  
ROW  

trigger_edition_clause  
trigger_ordering_clause  
ENABLE  
DISABLE  

WHEN  
condition  

trigger_body  

( dml_event_clause ::= , referencing_clause ::= , trigger_body ::= , trigger_edition_clause ::= , trigger_ordering_clause ::= )

instead_of_dml_trigger ::= 

INSTEAD  
OF  
DELETE  
INSERT  
UPDATE  

OR  

ON  
NESTED  
TABLE  
nested_table_column  
OF  

schema  
noneditioning_view  

referencing_clause 
FOR  
EACH  
ROW  

trigger_edition_clause  

trigger_ordering_clause  
ENABLE  
DISABLE  

trigger_body  

( referencing_clause ::= , trigger_body ::= , trigger_edition_clause ::= , trigger_ordering_clause ::= )
**Chapter 15**

CREATE TRIGGER Statement

### compound_dml_trigger ::=

FOR dml_event_clause referencing_clause trigger_edition_clause trigger_ordering_clause

ENABLE
DISABLE WHEN ( condition )

compound_trigger_block

( compound_trigger_block ::= , dml_event_clause ::= , referencing_clause ::= , trigger_edition_clause ::= , trigger_ordering_clause ::= )

### system_trigger ::= 

BEFORE
AFTER
INSTEAD OF
ddl_event
OR
database_event
OR ON
schema .
SCHEMA
PLUGGABLE
DATABASE

trigger_ordering_clause

ENABLE
DISABLE

trigger_body

( trigger_body ::= , trigger_ordering_clause ::= )

### dml_event_clause ::= 

DELETE
INSERT
UPDATE
OF column

OR

ON

schema . table
view

### referencing_clause ::= 

REFERENCING
OLD AS old
NEW AS new
PARENT AS parent
trigger_edition_clause ::= 

![Diagram of trigger_edition_clause]

trigger_ordering_clause ::= 

![Diagram of trigger_ordering_clause]

trigger_body ::= 

![Diagram of trigger_body]

( plsql_block ::= , 
  routine_clause in Oracle Database SQL Language Reference )

compound_trigger_block ::= 

![Diagram of compound_trigger_block]

( declare_section ::= )

timing_point_section ::= 

![Diagram of timing_point_section]

timing_point ::= 

![Diagram of timing_point]
### tps_body ::= 

( exception_handler ::= , statement ::= )

#### Semantics

**create_trigger**

**OR REPLACE**

Re-creates the trigger if it exists, and recompiles it.

Users who were granted privileges on the trigger before it was redefined can still access the procedure without being regranted the privileges.

**[ EDITIONABLE | NONEDITIONABLE ]**

Specifies whether the trigger is an editioned or noneditioned object if editioning is enabled for the schema object type TRIGGER in schema. **Default: EDITIONABLE.** For information about editioned and noneditioned objects, see Oracle Database Development Guide.

**Restriction on NONEDITIONABLE**

You cannot specify NONEDITIONABLE for a crossedition trigger.

**Restrictions on create_trigger**

See "Trigger Restrictions".

**plsql_trigger_source**

**schema**

Name of the schema for the trigger to be created. **Default: your schema.**

**trigger**

Name of the trigger to be created.

Triggers in the same schema cannot have the same names. Triggers can have the same names as other schema objects—for example, a table and a trigger can have the same name—however, to avoid confusion, this is not recommended.

If a trigger produces compilation errors, then it is still created, but it fails on execution. A trigger that fails on execution effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped. You can see the associated compiler error messages with the SQL*Plus command **SHOW ERRORS.**
**Note:**

If you create a trigger on a base table of a materialized view, then you must ensure that the trigger does not fire during a refresh of the materialized view. During refresh, the `DBMS_MVIEW` procedure `I_AM_A_REFRESH` returns `TRUE`.

*simple_dml_trigger*

Creates a simple DML trigger (described in "DML Triggers").

**BEFORE**

Causes the database to fire the trigger before running the triggering event. For row triggers, the trigger fires before each affected row is changed.

**Restrictions on BEFORE**

- You cannot specify a `BEFORE` trigger on a view unless it is an editioning view.
- In a `BEFORE` statement trigger, the trigger body cannot read `:NEW` or `:OLD`. (In a `BEFORE` row trigger, the trigger body can read and write the `:OLD` and `:NEW` fields.)

**AFTER**

Causes the database to fire the trigger after running the triggering event. For row triggers, the trigger fires after each affected row is changed.

**Restrictions on AFTER**

- You cannot specify an `AFTER` trigger on a view unless it is an editioning view.
- In an `AFTER` statement trigger, the trigger body cannot read `:NEW` or `:OLD`. (In an `AFTER` row trigger, the trigger body can read but not write the `:OLD` and `:NEW` fields.)

**Note:**

When you create a materialized view log for a table, the database implicitly creates an `AFTER` row trigger on the table. This trigger inserts a row into the materialized view log whenever an `INSERT`, `UPDATE`, or `DELETE` statement modifies data in the master table. You cannot control the order in which multiple row triggers fire. Therefore, do not write triggers intended to affect the content of the materialized view.

**See Also:**

- *Oracle Database SQL Language Reference* for more information about materialized view logs
- *Oracle Database Development Guide* for information about editioning views
FOR EACH ROW

Creates the trigger as a row trigger. The database fires a row trigger for each row that
is affected by the triggering statement and meets the optional trigger constraint defined
in the WHEN condition.

If you omit this clause, then the trigger is a statement trigger. The database fires a
statement trigger only when the triggering statement is issued if the optional trigger
constraint is met.

[ ENABLE | DISABLE ]

Creates the trigger in an enabled (default) or disabled state. Creating a trigger in a
disabled state lets you ensure that the trigger compiles without errors before you
enable it.

Note:

DISABLE is especially useful if you are creating a crossedition trigger, which
affects the online application being redefined if compilation errors occur.

WHEN (condition)

Specifies a SQL condition that the database evaluates for each row that the triggering
statement affects. If the value of condition is TRUE for an affected row, then
trigger_body runs for that row; otherwise, trigger_body does not run for that row. The
triggering statement runs regardless of the value of condition.

The condition can contain correlation names (see "referencing_clause ::=").

In condition, do not put a colon (:) before the correlation name NEW, OLD, or PARENT (in
this context, it is not a placeholder for a bind variable).

See Also:

Oracle Database SQL Language Reference for information about SQL
conditions

Restrictions on WHEN (condition)

• If you specify this clause, then you must also specify FOR EACH ROW.
• The condition cannot include a subquery or a PL/SQL expression (for example,
an invocation of a user-defined function).

trigger_body

The PL/SQL block or CALL subprogram that the database runs to fire the trigger. A
CALL subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL
wrapper.

If trigger_body is a PL/SQL block and it contains errors, then the CREATE [OR REPLACE]
statement fails.
Restriction on *trigger_body*

The *declare_section* cannot declare variables of the data type `LONG` or `LONG RAW`.

*instead_of_dml_trigger*

Creates an INSTEAD OF DML trigger (described in "INSTEAD OF DML Triggers").

Restriction on INSTEAD OF

An INSTEAD OF trigger can read the :OLD and :NEW values, but cannot change them.

---

Note:

- If the view is inherently updatable and has INSTEAD OF triggers, the triggers take precedence: The database fires the triggers instead of performing DML on the view.
- If the view belongs to a hierarchy, then the subviews do not inherit the trigger.
- The WITH CHECK OPTION for views is not enforced when inserts or updates to the view are done using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check. For information about WITH CHECK OPTION, see Oracle Database SQL Language Reference.
- The database fine-grained access control lets you define row-level security policies on views. These policies enforce specified rules in response to DML operations. If an INSTEAD OF trigger is also defined on the view, then the database does not enforce the row-level security policies, because the database fires the INSTEAD OF trigger instead of running the DML on the view.

---

DELETE

If the trigger is created on a noneditioning view, then DELETE causes the database to fire the trigger whenever a DELETE statement removes a row from the table on which the noneditioning view is defined.

If the trigger is created on a nested table column of a noneditioning view, then DELETE causes the database to fire the trigger whenever a DELETE statement removes an element from the nested table.

INSERT

If the trigger is created on a noneditioning view, then INSERT causes the database to fire the trigger whenever an INSERT statement adds a row to the table on which the noneditioning view is defined.

If the trigger is created on a nested table column of a noneditioning view, then INSERT causes the database to fire the trigger whenever an INSERT statement adds an element to the nested table.

UPDATE
If the trigger is created on a noneditioning view, then UPDATE causes the database to fire the trigger whenever an UPDATE statement changes a value in a column of the table on which the noneditioning view is defined.

If the trigger is created on a nested table column of a noneditioning view, then UPDATE causes the database to fire the trigger whenever an UPDATE statement changes a value in a column of the nested table.

**nested_table_column**

Name of the nested_table_column on which the trigger is to be created. The trigger fires only if the DML operates on the elements of the nested table. Performing DML operations directly on nested table columns does not cause the database to fire triggers defined on the table containing the nested table column. For more information, see "INSTEAD OF DML Triggers".

### See Also:

AS subquery clause of CREATE VIEW in Oracle Database SQL Language Reference for a list of constructs that prevent inserts, updates, or deletes on a view

**schema**

Name of the schema containing the noneditioning view. **Default**: your schema.

**noneditioning_view**

If you specify nested_table_column, then noneditioning_view is the name of the noneditioning view that includes nested_table_column. Otherwise, noneditioning_view is the name of the noneditioning view on which the trigger is to be created.

**FOR EACH ROW**

For documentation only, because an INSTEAD OF trigger is always a row trigger.

**ENABLE**

(**Default**) Creates the trigger in an enabled state.

**DISABLE**

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

### Note:

**DISABLE** is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors occur.

**trigger_body**
The PL/SQL block or CALL subprogram that the database runs to fire the trigger. A CALL subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If trigger_body is a PL/SQL block and it contains errors, then the CREATE [OR REPLACE] statement fails.

**Restriction on trigger_body**

The declare_section cannot declare variables of the data type LONG or LONG RAW.

**compound_dml_trigger**

Creates a compound DML trigger (described in "Compound DML Triggers").

**ENABLE**

(Default) Creates the trigger in an enabled state.

**DISABLE**

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

**Note:**

DISABLE is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors occur.

**WHEN (condition)**

Specifies a SQL condition that the database evaluates for each row that the triggering statement affects. If the value of condition is TRUE for an affected row, then tps_body runs for that row; otherwise, tps_body does not run for that row. The triggering statement runs regardless of the value of condition.

The condition can contain correlation names (see "referencing_clause ::="). In condition, do not put a colon (:) before the correlation name NEW, OLD, or PARENT (in this context, it is not a placeholder for a bind variable).

**See Also:**

*Oracle Database SQL Language Reference* for information about SQL conditions

**Restrictions on WHEN (condition)**

- If you specify this clause, then you must also specify at least one of these timing points:
  - BEFORE EACH ROW
  - AFTER EACH ROW
  - INSTEAD OF EACH ROW
• The condition cannot include a subquery or a PL/SQL expression (for example, an invocation of a user-defined function).

system_trigger
Defines a system trigger (described in "System Triggers").

BEFORE
Causes the database to fire the trigger before running the triggering event.

AFTER
Causes the database to fire the trigger after running the triggering event.

INSTEAD OF
Creates an INSTEAD OF trigger.

Restrictions on INSTEAD OF
• The triggering event must be a CREATE statement.
• You can create at most one INSTEAD OF DDL trigger (non_dml_trigger).
  For example, you can create an INSTEAD OF trigger on either the database or schema, but not on both the database and schema.

ddl_event
One or more types of DDL SQL statements that can cause the trigger to fire.

You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can create BEFORE and AFTER triggers for any of these events, but you can create INSTEAD OF triggers only for CREATE events. The database fires the trigger in the existing user transaction.

Note:
Some objects are created, altered, and dropped using PL/SQL APIs (for example, scheduler jobs are maintained by subprograms in the DBMS_SCHEDULER package). Such PL/SQL subprograms do not fire DDL triggers.

The following ddl_event values are valid:

• ALTER
  Causes the database to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary. An ALTER DATABASE statement does not fire the trigger.

• ANALYZE
  Causes the database to fire the trigger whenever the database collects or deletes statistics or validates the structure of a database object.
See Also:

Oracle Database SQL Language Reference for information about using the SQL statement ANALYZE to collect statistics

- ASSOCIATE STATISTICS
  Causes the database to fire the trigger whenever the database associates a statistics type with a database object.

- AUDIT
  Causes the database to fire the trigger whenever an AUDIT statement is issued.

- COMMENT
  Causes the database to fire the trigger whenever a comment on a database object is added to the data dictionary.

- CREATE
  Causes the database to fire the trigger whenever a CREATE statement adds a database object to the data dictionary. The CREATE DATABASE or CREATE CONTROLFILE statement does not fire the trigger.

- DISASSOCIATE STATISTICS
  Causes the database to fire the trigger whenever the database disassociates a statistics type from a database object.

- DROP
  Causes the database to fire the trigger whenever a DROP statement removes a database object from the data dictionary.

- GRANT
  Causes the database to fire the trigger whenever a user grants system privileges or roles or object privileges to another user or to a role.

- NOAUDIT
  Causes the database to fire the trigger whenever a NOAUDIT statement is issued.

- RENAME
  Causes the database to fire the trigger whenever a RENAME statement changes the name of a database object.

- REVOKE
  Causes the database to fire the trigger whenever a REVOKE statement removes system privileges or roles or object privileges from a user or role.

- TRUNCATE
  Causes the database to fire the trigger whenever a TRUNCATE statement removes the rows from a table or cluster and resets its storage characteristics.

- DDL
  Causes the database to fire the trigger whenever any of the preceding DDL statements is issued.
**database_event**

One of the following database events. You can create triggers for these events on either `DATABASE` or `SCHEMA` unless otherwise noted. For each of these triggering events, the database opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction).

- **AFTER STARTUP**
  Causes the database to fire the trigger whenever the database is opened. This event is valid only with `DATABASE`, not with `SCHEMA`.

- **BEFORE SHUTDOWN**
  Causes the database to fire the trigger whenever an instance of the database is shut down. This event is valid only with `DATABASE`, not with `SCHEMA`.

- **AFTER DB_ROLE_CHANGE**
  In an Oracle Data Guard configuration, causes the database to fire the trigger whenever a role change occurs from standby to primary or from primary to standby. This event is valid only with `DATABASE`, not with `SCHEMA`.

  **Note:**
  You cannot create an `AFTER DB_ROLE_CHANGE` trigger on a PDB.

- **AFTER SERVERERROR**
  Causes the database to fire the trigger whenever both of these conditions are true:
  - A server error message is logged.
  - Oracle relational database management system (RDBMS) determines that it is safe to fire error triggers.

  Examples of when it is unsafe to fire error triggers include:
  - RDBMS is starting up.
  - A critical error has occurred.

- **AFTER LOGON**
  Causes the database to fire the trigger whenever a client application logs onto the database.

- **BEFORE LOGOFF**
  Causes the database to fire the trigger whenever a client application logs off the database.

- **AFTER SUSPEND**
  Causes the database to fire the trigger whenever a server error causes a transaction to be suspended.

- **AFTER CLONE**
  Can be specified only if `PLUGGABLE DATABASE` is specified. After the PDB is copied (cloned), the database fires the trigger in the new PDB and then deletes the trigger. If the trigger fails, then the copy operation fails.
• **BEFORE UNPLUG**

  Can be specified only if `PLUGGABLE DATABASE` is specified. Before the PDB is unplugged, the database fires the trigger and then deletes it. If the trigger fails, then the unplug operation fails.

• **[ BEFORE | AFTER ] SET CONTAINER**

  Causes the database to fire the trigger either before or after an `ALTER SESSION SET CONTAINER` statement executes.

  **See Also:**

  "Triggers for Publishing Events" for more information about responding to database events through triggers

[`schema.`]**SCHEMA**

Defines the trigger on the specified schema. **Default:** current schema. The trigger fires whenever any user connected as the specified schema initiates the triggering event.

**[ PLUGGABLE ] DATABASE**

`DATABASE` defines the trigger on the root. In a multitenant container database (CDB), only a common user who is connected to the root can create a trigger on the entire database.

`PLUGGABLE DATABASE` defines the trigger on the PDB to which you are connected.

The trigger fires whenever any user of the specified database or PDB initiates the triggering event.

**Note:**

If you are connected to a PDB, then specifying `DATABASE` is equivalent to specifying `PLUGGABLE DATABASE` unless you want to specify an option that applies only to a PDB (such as `CLONE` or `UNPLUG`).

**ENABLE**

**(Default)** Creates the trigger in an enabled state.

**DISABLE**

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

**WHEN (condition)**

Specifies a SQL condition that the database evaluates. If the value of `condition` is `TRUE`, then `trigger_body` runs for that row; otherwise, `trigger_body` does not run for that row. The triggering statement runs regardless of the value of `condition`. 
Restrictions on WHEN (condition)

- You cannot specify this clause for a STARTUP, SHUTDOWN, or DB_ROLE_CHANGE trigger.
- If you specify this clause for a SERVERERROR trigger, then condition must be ERRNO = error_code.
- The condition cannot include a subquery, a PL/SQL expression (for example, an invocation of a user-defined function), or a correlation name.

trigger_body

The PL/SQL block or CALL subprogram that the database runs to fire the trigger. A CALL subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If trigger_body is a PL/SQL block and it contains errors, then the CREATE [OR REPLACE] statement fails.

Restrictions on trigger_body

- The declare_section cannot declare variables of the data type LONG or LONG RAW.
- The trigger body cannot specify either :NEW or :OLD.

dml_event_clause

Specifies the triggering statements for simple_dml_trigger or compound_dml_trigger. The database fires the trigger in the existing user transaction.

DELETE

Causes the database to fire the trigger whenever a DELETE statement removes a row from table or the table on which view is defined.

INSERT

Causes the database to fire the trigger whenever an INSERT statement adds a row to table or the table on which view is defined.

UPDATE [ OF column [, column ] ]

Causes the database to fire the trigger whenever an UPDATE statement changes a value in a specified column. Default: The database fires the trigger whenever an UPDATE statement changes a value in any column of table or the table on which view is defined.

If you specify a column, then you cannot change its value in the body of the trigger.

schema

Name of the schema that contains the database object on which the trigger is to be created. Default: your schema.
**table**
Name of the database table or object table on which the trigger is to be created.

**Restriction on schema.table**
You cannot create a trigger on a table in the schema `SYS`.

**view**
Name of the database view or object view on which the trigger is to be created.

---

**Note:**
A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section.

---

**referencing_clause**
Specifies correlation names, which refer to old, new, and parent values of the current row. **Defaults:** `OLD`, `NEW`, and `PARENT`.

If your trigger is associated with a table named `OLD`, `NEW`, or `PARENT`, then use this clause to specify different correlation names to avoid confusion between the table names and the correlation names.

If the trigger is defined on a nested table, then `OLD` and `NEW` refer to the current row of the nested table, and `PARENT` refers to the current row of the parent table. If the trigger is defined on a database table or view, then `OLD` and `NEW` refer to the current row of the database table or view, and `PARENT` is undefined.

**Restriction on referencing_clause**
The `referencing_clause` is not valid if `trigger_body` is CALL routine.

DML row-level triggers cannot reference fields of `OLD/NEW/PARENT` pseudorecords (correlation names) that correspond to columns with declared collation other than USING_NLS_COMP.

**trigger_edition_clause**
Creates the trigger as a crossedition trigger.

The handling of DML changes during edition-based redefinition (EBR) of an online application can entail multiple steps. Therefore, it is likely, though not required, that a crossedition trigger is also a **compound trigger**.

**Restrictions on trigger_edition_clause**

- You cannot define a crossedition trigger on a view.
- You cannot specify NONEDITIONABLE for a crossedition trigger.

**FORWARD**

(Default) Creates the trigger as a forward crossedition trigger. A forward crossedition trigger is intended to fire when DML changes are made in a database while an online application that uses the database is being patched or upgraded with EBR. The body...
of a crossedition trigger is designed to handle these DML changes so that they can be appropriately applied after the changes to the application code are completed.

**REVERSE**

Creates the trigger as a reverse crossedition trigger, which is intended to fire when the application, after being patched or upgraded with EBR, makes DML changes. This trigger propagates data to columns or tables used by the application before it was patched or upgraded.

### See Also:

*Oracle Database Development Guide* for more information on crossedition triggers

**trigger_ordering_clause**

**FOLLOWS | PRECEDES**

Specifies the relative firing of triggers that have the same timing point. It is especially useful when creating crossedition triggers, which must fire in a specific order to achieve their purpose.

Use **FOLLOWS** to indicate that the trigger being created must fire after the specified triggers. You can specify **FOLLOWS** for a conventional trigger or for a forward crossedition trigger.

Use **PRECEDES** to indicate that the trigger being created must fire before the specified triggers. You can specify **PRECEDES** only for a reverse crossedition trigger.

The specified triggers must exist, and they must have been successfully compiled. They need not be enabled.

If you are creating a noncrossedition trigger, then the specified triggers must be all of the following:

- Noncrossedition triggers
- Defined on the same table as the trigger being created
- Visible in the same edition as the trigger being created

If you are creating a crossedition trigger, then the specified triggers must be all of the following:

- Crossedition triggers
- Defined on the same table or editioning view as the trigger being created, unless you specify **FOLLOWS** or **PRECEDES**.

  If you specify **FOLLOWS**, then the specified triggers must be forward crossedition triggers, and if you specify **PRECEDES**, then the specified triggers must be reverse crossedition triggers. However, the specified triggers need not be on the same table or editioning view as the trigger being created.

- Visible in the same edition as the trigger being created

In the following definitions, A, B, C, and D are either noncrossedition triggers or forward crossedition triggers:
• If B specifies A in its `FOLLOWS` clause, then B **directly follows** A.
• If C directly follows B, and B directly follows A, then C **indirectly follows** A.
• If D directly follows C, and C indirectly follows A, then D indirectly follows A.
• If B directly or indirectly follows A, then B **explicitly follows** A (that is, the firing order of B and A is explicitly specified by one or more `FOLLOWS` clauses).

In the following definitions, A, B, C, and D are reverse crossedition triggers:

• If A specifies B in its `PRECEDES` clause, then A **directly precedes** B.
• If A directly precedes B, and B directly precedes C, then A **indirectly precedes** C.
• If A directly precedes B, and B indirectly precedes D, then A indirectly precedes D.
• If A directly or indirectly precedes B, then A **explicitly precedes** B (that is, the firing order of A and B is explicitly specified by one or more `PRECEDES` clauses).

Belongs to *compound_dml_trigger*.

**compound_trigger_block**

If the trigger is created on a noneditioning view, then *compound_trigger_block* must have only the **INSTEAD OF EACH ROW** section.

If the trigger is created on a table or editioning view, then timing point sections can be in any order, but no section can be repeated. The *compound_trigger_block* cannot have an **INSTEAD OF EACH ROW** section.

See Also:  
"Compound DML Trigger Structure"

**Restriction on compound_trigger_block**

The `declare_section` of *compound_trigger_block* cannot include PRAGMA AUTONOMOUS_TRANSACTION.

See Also:  
"Compound DML Trigger Restrictions"

**timing_point**

**BEFORE STATEMENT**

Specifies the **BEFORE STATEMENT** section of a *compound_dml_trigger* on a table or editioning view. This section causes the database to fire the trigger before running the triggering event.

**Restriction on BEFORE STATEMENT**

This section cannot specify **:NEW** or **:OLD**.
BEFORE EACH ROW

Specifies the BEFORE EACH ROW section of a compound_dml_trigger on a table or editioning view. This section causes the database to fire the trigger before running the triggering event. The trigger fires before each affected row is changed.

This section can read and write the :OLD and :NEW fields.

AFTER STATEMENT

Specifies the AFTER STATEMENT section of compound_dml_trigger on a table or editioning view. This section causes the database to fire the trigger after running the triggering event.

Restriction on AFTER STATEMENT

This section cannot specify :NEW or :OLD.

AFTER EACH ROW

Specifies the AFTER EACH ROW section of a compound_dml_trigger on a table or editioning view. This section causes the database to fire the trigger after running the triggering event. The trigger fires after each affected row is changed.

This section can read but not write the :OLD and :NEW fields.

INSTEAD OF EACH ROW

Specifies the INSTEAD OF EACH ROW section (the only timing point section) of a compound_dml_trigger on a noneditioning view. The database runs tps_body instead of running the triggering DML statement. For more information, see "INSTEAD OF DML Triggers".

Restriction on INSTEAD OF EACH ROW

- This section can appear only in a compound_dml_trigger on a noneditioning view.
- This section can read but not write the :OLD and :NEW values.

tps_body

The PL/SQL block or CALL subprogram that the database runs to fire the trigger. A CALL subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If tps_body is a PL/SQL block and it contains errors, then the CREATE [OR REPLACE] statement fails.

Restriction on tps_body

The declare_section cannot declare variables of the data type LONG or LONG RAW.

Examples

DML Triggers

- Example 10-1, "Trigger Uses Conditional Predicates to Detect Triggering Statement"
- Example 10-2, "INSTEAD OF Trigger"
- Example 10-3, "INSTEAD OF Trigger on Nested Table Column of View"
• Example 10-4, "Compound Trigger Logs Changes to One Table in Another Table"
• Example 10-5, "Compound Trigger Avoids Mutating-Table Error"

Triggers for Ensuring Referential Integrity
• Example 10-6, "Foreign Key Trigger for Child Table"
• Example 10-7, "UPDATE and DELETE RESTRICT Trigger for Parent Table"
• Example 10-8, "UPDATE and DELETE SET NULL Trigger for Parent Table"
• Example 10-9, "DELETE CASCADE Trigger for Parent Table"
• Example 10-10, "UPDATE CASCADE Trigger for Parent Table"
• Example 10-11, "Trigger Checks Complex Constraints"
• Example 10-12, "Trigger Enforces Security Authorizations"
• Example 10-13, "Trigger Derives New Column Values"

Triggers That Use Correlation Names and Pseudorecords
• Example 10-14, "Trigger Logs Changes to EMPLOYEES.SALARY"
• Example 10-15, "Conditional Trigger Prints Salary Change Information"
• Example 10-16, "Trigger Modifies CLOB Columns"
• Example 10-17, "Trigger with REFERENCING Clause"
• Example 10-18, "Trigger References OBJECT_VALUE Pseudocolumn"

System Triggers
• Example 10-19, "BEFORE Statement Trigger on Sample Schema HR"
• Example 10-20, "AFTER Statement Trigger on Database"
• Example 10-21, "Trigger Monitors Logons"
• Example 10-22, "INSTEAD OF CREATE Trigger on Schema"

Miscellaneous Trigger Examples
• Example 10-23, "Trigger Invokes Java Subprogram"
• Example 10-24, "Trigger Cannot Handle Exception if Remote Database is Unavailable"
• Example 10-25, "Workaround for Trigger Cannot Handle Exception if Remote Database is Unavailable"
• Example 10-26, "Trigger Causes Mutating-Table Error"
• Example 10-27, "Update Cascade"
• Example 10-28, "Viewing Information About Triggers"

Related Topics
In this chapter:
• "ALTER TRIGGER Statement"
• "DROP TRIGGER Statement"

In other chapters:
15.13 CREATE TYPE Statement

The CREATE TYPE statement specifies the name of the type and its attributes, methods, and other properties.

The CREATE TYPE statement creates or replaces the specification of one of these:

- Abstract Data Type (ADT)
- Standalone varying array (varray) type
- Standalone nested table type
- Incomplete object type

An incomplete type is a type created by a forward type definition. It is called incomplete because it has a name but no attributes or methods. It can be referenced by other types, allowing you to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

The CREATE TYPE BODY statement contains the code for the methods that implement the type.

**Note:**

- If you create a type whose specification declares only attributes but no methods, then you need not specify a type body.
- A standalone collection type that you create with the CREATE TYPE statement differs from a collection type that you define with the keyword TYPE in a PL/SQL block or package. For information about the latter, see “Collection Variable Declaration”.
- With the CREATE TYPE statement, you can create nested table and VARRAY types, but not associative arrays. In a PL/SQL block or package, you can define all three collection types.
• Related Topics

Prerequisites

To create a type in your schema, you must have the CREATE TYPE system privilege. To create a type in another user’s schema, you must have the CREATE ANY TYPE system privilege. You can acquire these privileges explicitly or be granted them through a role.

To create a subtype, you must have the UNDER ANY TYPE system privilege or the UNDER object privilege on the supertype.

The owner of the type must be explicitly granted the EXECUTE object privilege to access all other types referenced in the definition of the type, or the type owner must be granted the EXECUTE ANY TYPE system privilege. The owner cannot obtain these privileges through roles.

If the type owner intends to grant other users access to the type, then the owner must be granted the EXECUTE object privilege on the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

Syntax

```
create_type ::=  

plsql_type_source ::=  

object_base_type_def ::=  
```

```
object_base_type_def ::= IS AS (object_type_def | varray_type_spec | nested_table_type_spec)  

object_subtype_def ::= (sharing_clause ::= , default_collation_clause ::= , accessible_by_clause ::= , invoker_rights_clause ::= , object_base_type_def ::= , object_subtype_def ::=)
```
Semantics

create_type

OR REPLACE

Re-creates the type if it exists, and recompiles it.
Users who were granted privileges on the type before it was redefined can still access the type without being regranted the privileges.

If any function-based indexes depend on the type, then the database marks the indexes **DISABLED**.

**[ EDITIONABLE | NONEDITIONABLE ]**

Specifies whether the type is an editioned or noneditioned object if editioning is enabled for the schema object type `TYPE in schema`. **Default:** EDITIONABLE. For information about editioned and noneditioned objects, see *Oracle Database Development Guide*.

**plsql_type_source**

**schema**

Name of the schema containing the type. **Default:** your schema.

**type_name**

Name of an ADT, a nested table type, or a **VARRAY** type.

If creating the type results in compilation errors, then the database returns an error. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

The database 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 name of the user-defined type. You can also create a user-defined constructor using the **constructor_spec** syntax.

The parameters of the ADT constructor method are the data attributes of the ADT. They occur in the same order as the attribute definition order for the ADT. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

**FORCE**

If `type_name` exists and has type dependents, but not table dependents, **FORCE** forces the statement to replace the type. (If `type_name` has table dependents, the statement fails with or without **FORCE**.)

---

### Note:

If type `t1` has type dependent `t2`, and type `t2` has table dependents, then type `t1` also has table dependents.

### See Also:

*Oracle Database Object-Relational Developer's Guide*

**OID 'object_identifier'**
Establishes type equivalence of identical objects in multiple databases. See Oracle Database Object-Relational Developer's Guide for information about this clause.

**object_base_type_def**

Creates a schema-level ADT. Such ADTs are sometimes called **root** ADTs.

**IS | AS**

The keyword **IS** or **AS** is required when creating an ADT.

---

**object_subtype_def**

Creates a subtype of an existing type.

**UNDER supertype**

The existing supertype must be an ADT. The subtype you create in this statement inherits the properties of its supertype. It must either override some of those properties or add properties to distinguish it from the supertype.

---

**attribute**

Name of an ADT attribute. An ADT attribute is a data item with a name and a type specifier that forms the structure of the ADT. You must specify at least one attribute for each ADT. The name must be unique in the ADT, but can be used in other ADTs.

If you are creating a subtype, then the attribute name cannot be the same as any attribute or method name declared in the supertype chain.

**datatype**

The data type of an ADT attribute. This data type must be stored in the database; that is, either a predefined data type or a user-defined standalone collection type.

**Restrictions on datatype**

- You cannot impose the **NOT NULL** constraint on an attribute.
- You cannot specify attributes of type **ROWID**, **LONG**, or **LONG RAW**.
- You cannot specify a data type of **UROWID** for an ADT.
- If you specify an object of type **REF**, then the target object must have an object identifier.
• If you are creating a collection type for use as a nested table or varray column of a table, then you cannot specify attributes of type ANYTYPE, ANYDATA, or ANYDATASET.

• JSON cannot be an attribute of a user defined type (ADT).

**object_type_def**

Creates an ADT. The variables that form the data structure are called attributes. The member subprograms that define the behavior of the ADT are called methods.

**OBJECT**

The keyword **OBJECT** is required.

**[NOT] FINAL, [NOT] INSTANTIABLE , [NOT] PERSISTABLE**

At the schema level of the syntax, these clauses specify the inheritance attributes of the type.

**[NOT] FINAL**

Use the **[NOT] FINAL** clause to indicate whether any further subtypes can be created for this type:

• **(Default)** Specify **FINAL** if no further subtypes can be created for this type.

• Specify **NOT FINAL** if further subtypes can be created under this type.

**[NOT] INSTANTIABLE**

Use the **[NOT] INSTANTIABLE** clause to indicate whether any object instances of this type can be constructed:

• **(Default)** Specify **INSTANTIABLE** if object instances of this type can be constructed.

• Specify **NOT INSTANTIABLE** if no default or user-defined constructor exists for this ADT. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes, either inherited or specified in this statement.

**[NOT] PERSISTABLE**

Use **[NOT] PERSISTABLE** clause to indicate whether or not instances of the object type are persistable.

Only **PERSISTABLE** types can be stored in a table.

• **(Default)** You can specify **PERSISTABLE** if all the object type attributes are persistable. Creating a persistable object type with non-persistable attributes is not allowed.

• You can specify **NOT PERSISTABLE** if the object type attributes are persistable or non-persistable.

• Specify **NOT PERSISTABLE** if the ADT has a unique PL/SQL predefined types, such as BOOLEAN and PLS_INTEGER.

**Restrictions on [NOT] PERSISTABLE ADT**

You cannot specify the **[NOT] PERSISTABLE** clause in a subtype definition. The persistance property of a subtype is inherited from its supertype.
Non-persistable ADTs with PL/SQL unique attributes are only allowed in the PL/SQL context.

See: Example 15-28, "Creating a Non-Persistent Object Type"

varray_type_spec

Creates the type as an ordered set of elements, each of which has the same data type.

Restrictions on varray_type_spec

You can create a VARRAY type of XMLType or of a LOB type for procedural purposes, for example, in PL/SQL or in view queries. However, database storage of such a varray is not supported, so you cannot create an object table or an column of such a VARRAY type.

See Also:

"Example 15-26, Creating a Varray Type"

[NOT] PERSISTABLE
( datatype [NOT NULL] )

The parentheses before and after the datatype [NOT NULL] clause are required when PERSISTABLE is specified. The parentheses are optional if PERSISTABLE is not specified.

Use [NOT] PERSISTABLE clause to indicate whether or not instances of the collection type (VARRAY or nested table) are persistable.

- (Default) A collection can be PERSISTABLE only if the collection element type is persistable. Creating a persistable collection type with non-persistable element type is not allowed.
- Specify NOT PERSISTABLE if any element type of the collection is not persistable. You can specify NOT PERSISTABLE for any collection, whether the element type is persistable or not.
- Specify NOT PERSISTABLE if the collection has a unique PL/SQL predefined types, such as BOOLEAN and PLS_INTEGER.

Restrictions on [NOT] PERSISTABLE Varray and Nested Array

Non-persistable types with PL/SQL unique attributes are only allowed in the PL/SQL context.

See Example 15-27, "Creating a Non-Persistent Nested Array" and Example 15-29, "Creating a Non-Persistent Varray"

nested_table_type_spec

Creates a named nested table of type datatype.

[NOT] PERSISTABLE

Same as for VARRAY, see "[NOT] PERSISTABLE"
Examples

Example 15-23    ADT Examples

This example shows how the sample type `customer_typ` was created for the sample Order Entry (`oe`) schema. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```sql
CREATE TYPE customer_typ_demo AS OBJECT
  ( customer_id        NUMBER(6)
  , cust_first_name    VARCHAR2(20)
  , cust_last_name     VARCHAR2(20)
  , cust_address       CUST_ADDRESS_TYP
  , phone_numbers      PHONE_LIST_TYP
  , nls_language       VARCHAR2(3)
  , nls_territory      VARCHAR2(30)
  , credit_limit       NUMBER(9,2)
  , cust_email         VARCHAR2(30)
  , cust_orders        ORDER_LIST_TYP
 ) ;
```

In this example, the `data_typ1` ADT is created with one member function `prod`, which is implemented in the `CREATE TYPE BODY` statement:

```sql
CREATE TYPE data_typ1 AS OBJECT
  ( year NUMBER,
    MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
  ) ;
/

CREATE TYPE BODY data_typ1 IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
    BEGIN
      RETURN (year + invent);
    END;
  END;
```

Example 15-24    Creating a Subtype

This statement shows how the subtype `corporate_customer_typ` in the sample `oe` schema was created.

It is based on the `customer_typ` supertype created in the preceding example and adds the `account_mgr_id` attribute. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```sql
CREATE TYPE corporate_customer_typ_demo UNDER customer_typ
  ( account_mgr_id     NUMBER(6)
 ) ;
```
Example 15-25  Creating a Type Hierarchy

These statements create a type hierarchy.

Type `employee_t` inherits the `name` and `ssn` attributes from type `person_t` and in addition has `department_id` and `salary` attributes. Type `part_time_emp_t` inherits all of the attributes from `employee_t` and, through `employee_t`, those of `person_t` and in addition has a `num_hrs` attribute. Type `part_time_emp_t` is final by default, so no further subtypes can be created under it.

```sql
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER) NOT FINAL;
CREATE TYPE employee_t UNDER person_t (department_id NUMBER, salary NUMBER) NOT FINAL;
CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
```

You can use type hierarchies to create substitutable tables and tables with substitutable columns.

Example 15-26  Creating a Varray Type

This statement shows how the `phone_list_typ` VARRAY type with five elements in the sample `oe` schema was created.

A hypothetical name is given to the table so that you can duplicate this example in your test database:

```sql
CREATE TYPE phone_list_typ_demo AS VARRAY(5) OF VARCHAR2(25);
```

Example 15-27  Creating a Non-Persistable Nested Array

This example shows how to create a PL/SQL nested array with unique PL/SQL predefined type `PLS_INTEGER` that is not persistable and can only be used in your PL/SQL programs.

```sql
CREATE TYPE varr_int AS VARRAY(10) OF (PLS_INTEGER) NOT PERSISTABLE;
```

Example 15-28  Creating a Non-Persistable Object Type

This example shows how to create a PL/SQL object type with unique PL/SQL predefined type `PLS_INTEGER` that is not persistable and can only be used in your PL/SQL programs.

```sql
CREATE TYPE plsint AS OBJECT (I PLS_INTEGER) NOT PERSISTABLE;
```

Example 15-29  Creating a Non-Persistable Varray

This example shows how to create a PL/SQL varray with unique PL/SQL predefined type `BOOLEAN` that is not persistable and can only be used in your PL/SQL programs.

```sql
CREATE TYPE tab_bool AS TABLE OF (BOOLEAN) NOT PERSISTABLE;
```

Example 15-30  Creating a Nested Table Type

This example from the sample schema `pm` creates the table type `textdoc_tab` of type `textdoc_typ`:

```sql
CREATE TYPE textdoc_typ AS OBJECT
    ( document_typ      VARCHAR2(32)
    , formatted_doc     BLOB
    );
```
Example 15-31  Creating a Nested Table Type Containing a Varray

This example of multilevel collections is a variation of the sample table `oe.customers`.

In this example, the `cust_address` object column becomes a nested table column with the `phone_list_typ` varray column embedded in it. The `phone_list_typ_demo` type was created in "Example 15-26".

```sql
CREATE TYPE cust_address_typ2 AS OBJECT
  ( street_address     VARCHAR2(40)
  , postal_code        VARCHAR2(10)
  , city               VARCHAR2(30)
  , state_province     VARCHAR2(10)
  , country_id         CHAR(2)
  , phone              phone_list_typ_demo
  );

CREATE TYPE cust_nt_address_typ
  AS TABLE OF cust_address_typ2;
```

Example 15-32  Constructor Example

This example invokes the system-defined constructor to construct the `demo_typ` object and insert it into the `demo_tab` table.

```sql
CREATE TYPE demo_typ1 AS OBJECT (a1 NUMBER, a2 NUMBER);
CREATE TABLE demo_tab1 (b1 NUMBER, b2 demo_typ1);
INSERT INTO demo_tab1 VALUES (1, demo_typ1(2,3));
```

Example 15-33  Creating a Member Method

This example invokes method constructor `col.get_square`.

First the type is created:

```sql
CREATE TYPE demo_typ2 AS OBJECT (a1 NUMBER,
  MEMBER FUNCTION get_square RETURN NUMBER);
```

Next a table is created with an ADT column and some data is inserted into the table:

```sql
CREATE TABLE demo_tab2(col demo_typ2);
INSERT INTO demo_tab2 VALUES (demo_typ2(2));
```

The type body is created to define the member function, and the member method is invoked:

```sql
CREATE TYPE BODY demo_typ2 IS
  MEMBER FUNCTION get_square
  RETURN NUMBER
  IS x NUMBER;
  BEGIN
    SELECT c.col.a1*c.col.a1 INTO x
    FROM demo_tab2 c;
    RETURN (x);
  END;
```
Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

**Example 15-34 Creating a Static Method**

This example changes the definition of the `employee_t` type to associate it with the `construct_emp` function.

The example first creates an ADT `department_t` and then an ADT `employee_t` containing an attribute of type `department_t`:

```sql
CREATE OR REPLACE TYPE department_t AS OBJECT (
    deptno number(10),
    dname CHAR(30));
CREATE OR REPLACE TYPE employee_t AS OBJECT(
    empid RAW(16),
    ename CHAR(31),
    dept REF department_t,
    STATIC function construct_emp
        (name VARCHAR2, dept REF department_t)
        RETURN employee_t
);
```

This statement requires this type body statement.

```sql
CREATE OR REPLACE TYPE BODY employee_t IS
    STATIC FUNCTION construct_emp
        (name varchar2, dept REF department_t)
        RETURN employee_t IS
    BEGIN
        return employee_t(SYS_GUID(),name,dept);
    END;
END;
```

Next create an object table and insert into the table:

```sql
CREATE TABLE emptab OF employee_t;
INSERT INTO emptab
    VALUES (employee_t.construct_emp('John Smith', NULL));
```

**Related Topics**

- ALTER TYPE Statement
- CREATE TYPE BODY Statement
- DROP TYPE Statement
- Abstract Data Types
- Conditional Compilation Directive Restrictions
- Collection Variable Declaration
- Collection Types for information about user-defined standalone collection types
15.14 CREATE TYPE BODY Statement

The CREATE TYPE BODY defines or implements the member methods defined in the type specification that was created with the CREATE TYPE statement.

For each method specified in a type specification for which you did not specify the call_spec, you must specify a corresponding method body in the type body.

Prerequisites

Every member declaration in the CREATE TYPE specification for an ADT must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. To replace a type in another user's schema, you must have the DROP ANY TYPE system privilege.

Syntax

```
create_type_body ::= CREATE OR REPLACE EDITIONABLE NONEDITIONABLE TYPE BODY plsql_type_body_source
```

```
plsql_type_body_source ::= schema . type_name sharing_clause IS AS subprog_decl_in_type map_order_func_declaration , END ;
```
(sharing_clause ::=, map_order_func_declaration ::=, subprog_decl_in_type ::=)

subprog_decl_in_type ::= 

\[
\text{proc\_decl\_in\_type} ::= \\
\text{func\_decl\_in\_type} ::= \\
\text{constructor\_declaration} \\
\]

\[
\text{proc\_decl\_in\_type} ::= \\
\quad \text{PROCEDURE name (parameter\_declaration \') IS}
\]

\[
\text{body ::=, call\_spec ::=, declare\_section ::=, parameter\_declaration ::=} \\
\]

\[
\text{func\_decl\_in\_type} ::= \\
\quad \text{FUNCTION name (parameter\_declaration \') RETURN datatype IS}
\]

\[
\text{body ::=, invoker\_rights\_clause ::=, accessible\_by\_clause ::=, deterministic\_clause ::=, call\_spec ::=, declare\_section ::=, parameter\_declaration ::=, parallel\_enable\_clause ::=, result\_cache\_clause ::=, pipelined\_clause ::=} \\
\]

Chapter 15
CREATE TYPE BODY Statement
constructor_declaration ::= 

\[
\text{FINAL} \quad \text{INSTANTIABLE} \quad \text{CONSTRUCTOR} \quad \text{FUNCTION} \quad \text{datatype} \\
\text{SELF} \quad \text{IN} \quad \text{OUT} \quad \text{datatype} \quad \text{parameter} \quad \text{datatype} \\
\text{RETURN} \quad \text{SELF} \quad \text{AS} \quad \text{RESULT} \quad \text{IS} \quad \text{AS} \quad \text{declare_section} \quad \text{body} \\
\text{call_spec ::=} \\
\text{map_order_func_declaration ::=} \\
\text{MAP} \quad \text{ORDER} \quad \text{MEMBER} \quad \text{func_decl_in_type} \\
\]

Semantics

create_type_body

OR REPLACE

Re-creates the type body if it exists, and recompiles it.

Users who were granted privileges on the type body before it was redefined can still access the type body without being regranted the privileges.

You can use this clause to add member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE statement.

[ EDITIONABLE | NONEDITIONABLE ]

If you do not specify this property, then the type body inherits EDITIONABLE or NONEDITIONABLE from the type specification. If you do specify this property, then it must match that of the type specification.

plsql_type_body_source

schema

Name of the schema containing the type body. Default: your schema.

type_name

Name of an ADT.
**subprog_decl_in_type**

The type of function or procedure subprogram associated with the type specification.

You must define a corresponding method name and optional parameter list in the type specification for each procedure or function declaration. For functions, you also must specify a return type.

**map_order_func_declaration**

You can declare either one **MAP** method or one **ORDER** method, regardless of how many **MEMBER** or **STATIC** methods you declare. If you declare either a **MAP** or **ORDER** method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

**MAP MEMBER**

Declares or implements a **MAP** member function 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.

If the argument to the **MAP** method is null, then the **MAP** method returns null and the method is not invoked.

An 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**

Specifies an **ORDER** member function that takes an instance of an object as an explicit argument and the implicit **SELF** argument and returns either a negative integer, zero, or a positive integer, indicating that the implicit **SELF** argument is less than, equal to, or greater than the explicit argument, respectively.

If either argument to the **ORDER** method is null, then the **ORDER** method returns null and the method is not invoked.

When instances of the same ADT definition are compared in an **ORDER BY** clause, the database invokes the **ORDER MEMBER func_decl_in_type**.

An object specification can contain only one **ORDER** method, which must be a function having the return type **NUMBER**.

**proc_decl_in_type**

A procedure subprogram declaration.

**constructor_declaration**

A user-defined constructor subprogram declaration. The **RETURN** clause of a constructor function must be **RETURN SELF AS RESULT**. This setting indicates that the most specific type of the value returned by the constructor function is the most specific type of the **SELF** argument that was passed in to the constructor function.
See Also:

- "CREATE TYPE Statement" for a list of restrictions on user-defined functions
- "Overloaded Subprograms" for information about overloading subprogram names
- Oracle Database Object-Relational Developer's Guide for information about and examples of user-defined constructors

\[\textit{declare\_section}\]

Declares items that are local to the procedure or function.

\[\textit{body}\]

Procedure or function statements.

\[\textit{func\_decl\_in\_type}\]

A function subprogram declaration.

Examples

Several examples of creating type bodies appear in the Examples section of "CREATE TYPE Statement". For an example of re-creating a type body, see "Example 15-7".

Related Topics

- "CREATE TYPE Statement"
- "DROP TYPE BODY Statement"
- "CREATE FUNCTION Statement"
- "CREATE PROCEDURE Statement"

15.15 DROP FUNCTION Statement

The \texttt{DROP FUNCTION} statement drops a standalone function from the database.

\[\textit{Note:}\]

Do not use this statement to drop a function that is part of a package. Instead, either drop the entire package using the "DROP PACKAGE Statement" or redefine the package without the function using the "CREATE PACKAGE Statement" with the OR REPLACE clause.

Topics

- Prerequisites
- Syntax
• Semantics
• Example
• Related Topics

Prerequisites
The function must be in your schema or you must have the \texttt{DROP ANY PROCEDURE} system privilege.

Syntax

\texttt{drop\_function ::=}

\begin{verbatim}
   DROP FUNCTION schema.function_name;
\end{verbatim}

Semantics

\textit{drop\_function}

\textit{schema}

Name of the schema containing the function. \textbf{Default:} your schema.

\textit{function\_name}

Name of the function to be dropped.

The database invalidates any local objects that depend on, or invoke, the dropped function. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped function.

If any statistics types are associated with the function, then the database disassociates the statistics types with the \texttt{FORCE} option and drops any user-defined statistics collected with the statistics type.

\begin{tcolorbox}[breakable]
\textbf{See Also:}
\begin{itemize}
\item \textit{Oracle Database SQL Language Reference} for information about the \texttt{ASSOCIATE STATISTICS} statement
\item \textit{Oracle Database SQL Language Reference} for information about the \texttt{DISASSOCIATE STATISTICS} statement
\end{itemize}
\end{tcolorbox}

Example

Example 15-35  Dropping a Function

This statement drops the function \texttt{SecondMax} in the sample schema \texttt{oe} and invalidates all objects that depend upon \texttt{SecondMax}:

\begin{verbatim}
   DROP FUNCTION oe.SecondMax;
\end{verbatim}
15.16 DROP LIBRARY Statement

The DROP LIBRARY statement drops an external procedure library from the database.

Topics
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites
You must have the DROP ANY LIBRARY system privilege.

Syntax

\[
drop\_library ::= \text{\textsc{drop library}} \text{ library\_name}
\]

Semantics

\text{library\_name}
Name of the external procedure library being dropped.

Example

Example 15-36 Dropping a Library

The following statement drops the ext_lib library, which was created in "CREATE LIBRARY Statement":

\[
\text{DROP LIBRARY ext_lib;}
\]

Related Topics
- "ALTER LIBRARY Statement"
15.17 DROP PACKAGE Statement

The DROP PACKAGE statement drops a stored package from the database.

This statement drops the body and specification of a package.

**Note:**

Do not use this statement to drop a single object from a package. Instead, re-create the package without the object using the "CREATE PACKAGE Statement" and "CREATE PACKAGE BODY Statement" with the OR REPLACE clause.

**Topics**

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

**Prerequisites**

The package must be in your schema or you must have the DROP ANY PROCEDURE system privilege.

**Syntax**

\[\text{drop\_package} ::= \]

\[\text{DROP PACKAGE}\ \text{BODY}\ \text{schema}\ .\ \text{package}\ ;\]

**Semantics**

\[\text{drop\_package}\]

**BODY**

Drops only the body of the package. If you omit this clause, then the database drops both the body and specification of the package.

When you drop only the body of a package but not its specification, the database does not invalidate dependent objects. However, you cannot invoke a procedure or stored function declared in the package specification until you re-create the package body.

**schema**

Name of the schema containing the package. **Default:** your schema.
**package**

Name of the package to be dropped.

The database invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, then the database disassociates the statistics types with the **FORCE** clause and drops any user-defined statistics collected with the statistics types.

---

**See Also:**

- *Oracle Database SQL Language Reference* for information about the **ASSOCIATE STATISTICS** statement
- *Oracle Database SQL Language Reference* for information about the **DISASSOCIATE STATISTICS** statement

---

**Example**

**Example 15-37   Dropping a Package**

This statement drops the specification and body of the `emp_mgmt` package, which was created in "CREATE PACKAGE BODY Statement", invalidating all objects that depend on the specification:

```
DROP PACKAGE emp_mgmt;
```

**Related Topics**

- "ALTER PACKAGE Statement"
- "CREATE PACKAGE Statement"
- "CREATE PACKAGE BODY Statement"

---

**15.18 DROP PROCEDURE Statement**

The **DROP PROCEDURE** statement drops a standalone procedure from the database.

---

**Note:**

Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the "DROP PACKAGE Statement", or redefine the package without the procedure using the "CREATE PACKAGE Statement" with the **OR REPLACE** clause.

---

**Topics**

- Prerequisites
• Syntax
• Semantics
• Example
• Related Topics

Prerequisites
The procedure must be in your schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax

\[\text{drop\_procedure ::=} \]

\[\text{DROP}\ \text{PROCEDURE}\ \text{schema} \ .\ \text{procedure} ;\]

Semantics

\[\text{schema}\]

Name of the schema containing the procedure. Default: your schema.

\[\text{procedure}\]

Name of the procedure to be dropped.

When you drop a procedure, the database invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

Example

Example 15-38 Dropping a Procedure

This statement drops the procedure remove_emp owned by the user hr and invalidates all objects that depend upon remove_emp:

\[\text{DROP PROCEDURE hr.remove\_emp;}\]

Related Topics

• "ALTER PROCEDURE Statement"
• "CREATE PROCEDURE Statement"

15.19 DROP TRIGGER Statement

The DROP TRIGGER statement drops a database trigger from the database.

Topics

• Prerequisites
Prerequisites

The trigger must be in your schema or you must have the `DROP ANY TRIGGER` system privilege. To drop a trigger on `DATABASE` in another user’s schema, you must also have the `ADMINISTER DATABASE TRIGGER` system privilege.

Syntax

```sql
drop_trigger ::= DROP TRIGGER schema . trigger ;
```

Semantics

`schema`

Name of the schema containing the trigger. **Default:** your schema.

`trigger`

Name of the trigger to be dropped.

Example

**Example 15-39  Dropping a Trigger**

This statement drops the `salary_check` trigger in the schema `hr`:

```sql
DROP TRIGGER hr.salary_check;
```

Related Topics

- "ALTER TRIGGER Statement"
- "CREATE TRIGGER Statement"

15.20 DROP TYPE Statement

The `DROP TYPE` statement drops the specification and body of an ADT, `VARRAY` type, or nested table type.

Topics

- Prerequisites
- Syntax
- Semantics
- Example
• Related Topics

Prerequisites

The ADT, VARRAY type, or nested table type must be in your schema or you must have the DROP ANY TYPE system privilege.

Syntax

\[\text{drop_type ::= }\]

```
DROP TYPE
schema .
type_name
FORCE
VALIDATE
;
```

Semantics

**schema**

Name of the schema containing the type. **Default**: your schema.

**type_name**

Name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If **type_name** is a supertype, then this statement fails unless you also specify **FORCE**. If you specify **FORCE**, then the database invalidates all subtypes depending on this supertype.

If **type_name** is a statistics type, then this statement fails unless you also specify **FORCE**. If you specify **FORCE**, then the database first disassociates all objects that are associated with **type_name** and then drops **type_name**.

See Also:

• *Oracle Database SQL Language Reference* for information about the ASSOCIATE STATISTICS statement
• *Oracle Database SQL Language Reference* for information about the DISASSOCIATE STATISTICS statement

If **type_name** is an ADT that is associated with a statistics type, then the database first tries to disassociate **type_name** from the statistics type and then drops **type_name**. However, if statistics have been collected using the statistics type, then the database cannot disassociate **type_name** from the statistics type, and this statement fails.

If **type_name** is an implementation type for an index type, then the index type is marked INVALID.

If **type_name** has a public synonym defined on it, then the database also drops the synonym.
Unless you specify FORCE, you can drop only types that are standalone schema objects with no dependencies. This is the default behavior.

**See Also:**

*Oracle Database SQL Language Reference* for information about the `CREATE INDEXTYPE` statement

**FORCE**

Drops the type even if it has dependent database objects. The database marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible.

**Note:**

Oracle recommends against specifying FORCE to drop object types with dependencies. This operation is not recoverable and might make the data in the dependent tables or columns inaccessible.

**VALIDATE**

Causes the database to check for stored instances of this type in substitutable columns of any of its supertypes. If no such instances are found, then the database completes the drop operation.

This clause is meaningful only for subtypes. Oracle recommends the use of this option to safely drop subtypes that do not have any explicit type or table dependencies.

**Example**

**Example 15-40  Dropping an ADT**

This statement removes the ADT `person_t`. See "CREATE TYPE Statement" for the example that creates this ADT. Any columns that are dependent on `person_t` are marked UNUSED and become inaccessible.

```
DROP TYPE person_t FORCE;
```

**Related Topics**

- "ALTER TYPE Statement"
- "CREATE TYPE Statement"
- "CREATE TYPE BODY Statement"
15.21 DROP TYPE BODY Statement

The DROP TYPE BODY statement drops the body of an ADT, VARRAY type, or nested table type.

When you drop a type body, the type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the type, although you cannot invoke its member functions.

Topics

• Prerequisites
• Syntax
• Semantics
• Example
• Related Topics

Prerequisites

The type body must be in your schema or you must have the DROP ANY TYPE system privilege.

Syntax

\[
\text{drop\_type\_body} ::= \\
\text{DROP} \text{ TYPE} \text{ BODY} \\
\text{schema} . \text{type\_name} ;
\]

Semantics

\text{schema}

Name of the schema containing the type. Default: your schema.

\text{type\_name}

Name of the type body to be dropped.

Restriction on type\_name

You can drop a type body only if it has no type or table dependencies.

Example

Example 15-41  Dropping an ADT Body

This statement removes the ADT body data\_typ1. See "CREATE TYPE Statement" for the example that creates this ADT.

\[
\text{DROP TYPE BODY data\_typ1;}
\]
Related Topics

- "ALTER TYPE Statement"
- "CREATE TYPE Statement"
- "CREATE TYPE BODY Statement"
PL/SQL Source Text Wrapping

You can wrap the PL/SQL source text for any of these stored PL/SQL units, thereby preventing anyone from displaying that text with the static data dictionary views *_SOURCE:

- Package specification
- Package body
- Type specification
- Type body
- Function
- Procedure

Note:
Wrapping text is low-assurance security. For high-assurance security, use Oracle Database Vault, described in Oracle Database Vault Administrator’s Guide.

A file containing wrapped PL/SQL source text is called a wrapped file. A wrapped file can be moved, backed up, or processed by SQL*Plus or the Import and Export utilities.

To produce a wrapped file, use either the PL/SQL Wrapper utility or a DBMS_DDL subprogram. The PL/SQL Wrapper utility wraps the source text of every wrappable PL/SQL unit created by a specified SQL file. The DBMS_DDL subprograms wrap the source text of single dynamically generated wrappable PL/SQL units.

Both the PL/SQL Wrapper utility and DBMS_DDL subprograms detect tokenization errors (for example, runaway strings), but not syntax or semantic errors (for example, nonexistent tables or views).

By default, the 12.2 PL/SQL compiler can use wrapped packages that were compiled with the 9.2 PL/SQL compiler. To prevent the 12.2 PL/SQL compiler from using wrapped packages that were compiled with the 9.2 PL/SQL compiler, set the PL/SQL compilation parameter PERMIT_92_WRAP_FORMAT to FALSE. For more information about PERMIT_92_WRAP_FORMAT, see Oracle Database Reference. For more information about PL/SQL compilation parameters, see "PL/SQL Units and Compilation Parameters".

Topics

- PL/SQL Source Text Wrapping Limitations
- PL/SQL Source Text Wrapping Guidelines
- Wrapping PL/SQL Source Text with PL/SQL Wrapper Utility
• Wrapping PL/SQL Source Text with DBMS_DDL Subprograms

A.1 PL/SQL Source Text Wrapping Limitations

• Wrapped files are not downward-compatible between Oracle Database releases. For example, you cannot load files produced by the version \( n.1 \) PL/SQL Wrapper utility into a version \( (n-1).2 \) Oracle Database. Nor can you load files produced by the version \( n.2 \) PL/SQL Wrapper utility into a version \( n.1 \) Oracle Database.

• Wrapped files are both upward- and downward-compatible across patch sets.

• Wrapping PL/SQL source text is not a secure way to hide passwords or table names.
  For high-assurance security, use Oracle Database Vault, described in Oracle Database Vault Administrator's Guide.

• You cannot wrap the PL/SQL source text of triggers.
  To hide the implementation details of a trigger, put them in a stored subprogram, wrap the subprogram, and write a one-line trigger that invokes the subprogram.

A.2 PL/SQL Source Text Wrapping Guidelines

• Wrap only the body of a package or type, not the specification.
  Leaving the specification unwrapped allows other developers to see the information needed to use the package or type (see Example A-5). Wrapping the body prevents them from seeing the package or type implementation.

• Wrap files only after you have finished editing them.
  You cannot edit wrapped files. If a wrapped file needs changes, you must edit the original unwrapped file and then wrap it.

• Before distributing a wrapped file, view it in a text editor and ensure that all important parts are wrapped.

A.3 Wrapping PL/SQL Source Text with PL/SQL Wrapper Utility

The PL/SQL Wrapper utility takes a single SQL file (such as a SQL*Plus script) and produces an equivalent text file in which the PL/SQL source text of each wrappable PL/SQL unit is wrapped.

Note:
Oracle recommends using PL/SQL Wrapper Utility version 10 or later.

For the list of wrappable PL/SQL units, see the introduction to "PL/SQL Source Text Wrapping".
The PL/SQL Wrapper utility cannot connect to Oracle Database. To run the PL/SQL Wrapper utility, enter this command at the operating system prompt (with no spaces around the equal signs):

```
wrap iname=input_file [ oname=output_file ] [ keep_comments=yes ]
```

`input_file` is the name of an existing file that contains any combination of SQL statements. `output_file` is the name of the file that the PL/SQL Wrapper utility creates—the wrapped file.

### Note:

`input_file` cannot include substitution variables specified with the SQL*Plus `DEFINE` notation, because `output_file` is parsed by the PL/SQL compiler, not by SQL*Plus.

The PL/SQL Wrapper utility deletes all comments from the wrapped file unless `keep_comments=yes` is specified. When `keep_comments=yes` is specified, only the comments outside the source are kept.

### Note:

If `input_file` is a wrapped file, then `input_file` and `output_file` have identical contents.

The default file extension for `input_file` is `sql`. The default name of `output_file` is `input_file.plb`. Therefore, these commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

This example specifies a different file extension for `input_file` and a different name for `output_file`:

```
wrap iname=/mydir/myfile.src oname=/yourdir/yourfile.out keep_comments=yes
```

You can run `output_file` as a script in SQL*Plus. For example:

```
SQL> @myfile.plb;
```

### Example A-1  SQL File with Two Wrappable PL/SQL Units

This example shows the text of a SQL file, `wraptest2.sql`, that contains two wrappable PL/SQL units—the procedure `wraptest` and the function `fibonacci`. The file also contains comments and a SQL `SELECT` statement.

```
-- The following statement will not change.
SELECT COUNT(*) FROM EMPLOYEES
/

/* The PL/SQL source text of the following two CREATE statements will be wrapped. */
```
CREATE PROCEDURE wraptest AUTHID CURRENT_USER /* C style comment in procedure declaration */ IS
    TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
    all_emps  emp_tab;
BEGIN
    SELECT * BULK COLLECT INTO all_emps FROM employees;
    FOR i IN 1..10 LOOP /* C style in pl/sql source */
        DBMS_OUTPUT.PUT_LINE('Emp Id: ' || all_emps(i).employee_id);
    END LOOP;
END;
/

CREATE OR REPLACE FUNCTION fibonacci (n PLS_INTEGER) RETURN PLS_INTEGER
AUTHID CURRENT_USER -- PL/SQL style comment inside fibonacci function spec IS
    fib_1 PLS_INTEGER := 0;
    fib_2 PLS_INTEGER := 1;
BEGIN
    IF n = 1 THEN -- terminating condition
        RETURN fib_1;
    ELSIF n = 2 THEN
        RETURN fib_2; -- terminating condition
    ELSE
        RETURN fibonacci(n-2) + fibonacci(n-1); -- recursive invocations
    END IF;
END;
/

Example A-2 Wrapping File with PL/SQL Wrapper Utility

This example uses the PL/SQL Wrapper utility to wrap wraptest2.sql and shows the wrapped file, wraptest2.plb. The wrapped file shows that the utility deleted the comments inside the code and wrapped (made unreadable) the PL/SQL source text of the procedure wraptest and the function fibonacci, but kept the comments outside the wrapped source.

Assume that the operating system prompt is >. Wrap the file wraptest.sql:

> wrap keep_comments=yes iname=wraptest2.sql

Result:

Processing wraptest2.sql to wraptest2.plb

Contents of wraptest.plb:

-- The following statement will not change.
SELECT COUNT(*) FROM EMPLOYEES
/

/* The PL/SQL source text of the following two CREATE statements will be wrapped. */
CREATE OR REPLACE PROCEDURE wraptest wrapped
a000000
1
abcd
abcd
abcd
Example A-3    Running Wrapped File and Viewing Wrapped PL/SQL Units

In SQL*Plus, this example runs the wrapped file wraptest.plb, creating the procedure wraptest and the function fibonacci; selects the text of the subprograms (which is wrapped and therefore unreadable), and then invokes the subprograms.

```sql
SQL> -- Run wrapped file:
SQL> @wraptest2.plb
SQL> -- The following statement will not change.
SQL> SELECT COUNT(*) FROM EMPLOYEES
```

COUNT(*)
---------
107
1 row selected.

SQL> /* The PL/SQL source text of the following two CREATE statements will be wrapped. */

SQL> CREATE PROCEDURE wraptest
   2  a000000
   3  1
   4  abcd
   5  abcd
   6  abcd
   7  abcd
   8  abcd
   9  abcd
  10  abcd
  11  abcd
  12  abcd
  13  abcd
  14  abcd
  15  abcd
  16  abcd
  17  abcd
  18  abcd
  19  7
  20  129 138
  21  qf4HggDBeNMPlWAsPn6pGf+2LGwpg+nwJK5qZ3SVWE4+GayD1aL1bF7RwYm2/zzlqj2X3FrN
  22  48M1bKcWMG5aY9Y8+DrtT45JN370Rpq7ck5D0sc1D5sKAwTyX1JHV9RmjwkdXa0v8Z4q/mCU
  23  EQusX23U2z2jxha7Ct1C0Cw8guGw/M/ohXxc6w8X8L8V80osqWMV/Hj7e68gIN170stalRScr
  24  uZ1/1/W1yaaA9Lj8Fbx5/nJw96ZNy1SCIY8VsB/G605f/65+EDxdThpfU4elvrrE91B3/IpI
  25  +7FE1z9f wc+aZq3570
  26
  27 /

Procedure created.

SQL> CREATE OR REPLACE FUNCTION fibonacci
   2  a000000
   3  1
   4  abcd
   5  abcd
   6  abcd
   7  abcd
   8  abcd
   9  abcd
  10  abcd
  11  abcd
  12  abcd
  13  abcd
  14  abcd
  15  abcd
  16  abcd
  17  abcd
  18  abcd
  19  8
  20  150 ff
Function created.

SQL> -- Try to display procedure source text:
SQL> SELECT text FROM USER_SOURCE WHERE name='WRAPTEST';

TEXT
--------------------------------------------------------------------------------
PROCEDURE wraptest wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
5
129 138
qf4HggDBeNMP1WAsPnPn6pGf+2LGowg+rWJK5q23SVWE4+GayDzALbF7RwYm2/zr1qjZY3FrN48M1bRC/mG5aY9Ye+DrtT4SJN370Rqj7ck5Dsc1D5sKAwTyX13HyvRm/wkdxAs0VE34q/mCU
EGusX23UBb2jxha7Ct1CDCx8guGW/M/oHEXc8dHXL8V80sqQMWv/hj7z6Bg1N170stalRScr
uB2/gYIYaaA9Lj8Fbx5/nJw962NyI5CY8VsB/G6O5f/65+EDxdTphpnF4+envrE9iB3/lpi
+7fE1Tv9fwC+aZ3870

1 row selected.

SQL> -- Try to display function source text:
SQL> SELECT text FROM USER_SOURCE WHERE name='FIBONACCI';

TEXT
--------------------------------------------------------------------------------
FUNCTION fibonacci wrapped
a000000
1
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
129 138
qf4HggDBeNMP1WAsPnPn6pGf+2LGowg+rWJK5q23SVWE4+GayDzALbF7RwYm2/zr1qjZY3FrN48M1bRC/mG5aY9Ye+DrtT4SJN370Rqj7ck5Dsc1D5sKAwTyX13HyvRm/wkdxAs0VE34q/mCU
EGusX23UBb2jxha7Ct1CDCx8guGW/M/oHEXc8dHXL8V80sqQMWv/hj7z6Bg1N170stalRScr
uB2/gYIYaaA9Lj8Fbx5/nJw962NyI5CY8VsB/G6O5f/65+EDxdTphpnF4+envrE9iB3/lpi
+7fE1Tv9fwC+aZ3870

1 row selected.
A.4 Wrapping PL/SQL Source Text with DBMS_DDL Subprograms

The DBMS_DDL package provides WRAP functions and CREATE_WRAPPED procedures, each of which wraps the PL/SQL source text of a single dynamically generated wrappable PL/SQL unit. The DBMS_DDL package also provides the exception MALFORMED_WRAP_INPUT (ORA-24230), which is raised if the input to WRAP or CREATE_WRAPPED is not a valid wrappable PL/SQL unit. (For the list of wrappable PL/SQL units, see the introduction to "PL/SQL Source Text Wrapping".)

Each WRAP function takes as input a single CREATE statement that creates a wrappable PL/SQL unit and returns an equivalent CREATE statement in which the PL/SQL source text is wrapped. For more information about the WRAP functions, see Oracle Database PL/SQL Packages and Types Reference.
Caution:

If you pass the statement that `DBMS_DDL.WRAP` returns to the `DBMS_SQL.PARSE` procedure whose formal parameter statement has data type `VARCHAR2A`, then you must set the `ifflg` parameter of `DBMS_SQL.PARSE` to `FALSE`. Otherwise, `DBMS_SQL.PARSE` adds lines to the wrapped PL/SQL unit, corrupting it. (For the syntax of `DBMS_SQL.PARSE`, see Oracle Database PL/SQL Packages and Types Reference.)

Each `CREATE_WRAPPED` procedure does what its corresponding `WRAP` function does and then runs the returned `CREATE` statement, creating the specified PL/SQL unit. For more information about the `CREATE_WRAPPED` procedures, see Oracle Database PL/SQL Packages and Types Reference.

Tip:

When invoking a `DBMS_DDL` subprogram, use the fully qualified package name, `SYS.DBMS_DDL`, to avoid name conflict if someone creates a local package named `DBMS_DDL` or defines the public synonym `DBMS_DDL`.

Note:

The `CREATE` statement that is input to a `WRAP` function or `CREATE_WRAPPED` procedure runs with the privileges of the user who invoked the subprogram.

Example A-4 dynamically creates a package specification (using the `EXECUTE IMMEDIATE` statement) and a wrapped package body, using a `CREATE_WRAPPED` procedure.

Example A-5 selects the text of the package that Example A-4 created, `emp_actions`, and then invokes the procedure `emp_actions.raise_salary`. If the package specification were wrapped, then the information needed to invoke the procedure would be unreadable, like the PL/SQL source text of the package body.

Example A-4 Creating Wrapped Package Body with `CREATE_WRAPPED` Procedure

```sql
DECLARE
  package_text  VARCHAR2(32767); -- text for creating package spec and body

FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS BEGIN
  RETURN 'CREATE PACKAGE ' || pkgname || ' AUTHID CURRENT_USER AS
  PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
  PROCEDURE fire_employee (emp_id NUMBER);
  END ' || pkgname || ';';
END generate_spec;

FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2 AS BEGIN
```

A-9
RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
    BEGIN
        UPDATE employees
            SET salary = salary + amount WHERE employee_id = emp_id;
        END raise_salary;
    END raise_salary;
PROCEDURE fire_employee (emp_id NUMBER) IS
    BEGIN
        DELETE FROM employees WHERE employee_id = emp_id;
        END fire_employee;
    END ' || pkgname || ';'
END generate_body;
BEGIN
    package_text := generate_spec('emp_actions');  -- Generate package spec
    EXECUTE IMMEDIATE package_text;                -- Create package spec
    package_text := generate_body('emp_actions');  -- Generate package body
    SYS.DBMS_DDL.CREATE_WRAPPED(package_text);     -- Create wrapped package body
END;
/

Example A-5  Viewing Package with Wrapped Body and Invoking Package Procedure

Select text of package:

SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';

Result:

TEXT
------------------------------------------------------------------------
PACKAGE emp_actions AUTHID CURRENT_USER AS
    PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
    PROCEDURE fire_employee (emp_id NUMBER);
    END emp_actions;
PACKAGE BODY emp_actions wrapped
a000000
 1f
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
b
180 113
1f0VodeXem7j9dB0mBaiEjQzOBKCGwg/BKoZ4V2y/pTBIYo8Uj1sjpEbEz08Ck3HMjYq/Mf0X2n
u9D0Kd+i189g%06116v2Yjy2Aum8dNLeSs061LH2pFD/7lyDtfJeDY5vmNwLTXrFazGy243
O1HkAzm01wffBYy1kZ2N2UNpms1e6z/BU2nbwbfpqd224p69FwYVXmF2H51MsdZ2/vWkAf9
CDMCD1KEQOnPpbU2yXdpW3G1bGD8JFIbKAfपJLkcLfVxoRVPXQfj0hlk=
Invoke raised_salary and show its effect:

```
DECLARE
    s employees.salary%TYPE;
BEGIN
    SELECT salary INTO s FROM employees WHERE employee_id=130;
    DBMS_OUTPUT.PUT_LINE('Old salary: ' || s);
    emp_actions.raise_salary(130, 100);
    SELECT salary INTO s FROM employees WHERE employee_id=130;
    DBMS_OUTPUT.PUT_LINE('New salary: ' || s);
END;
/
```

Result:

Old salary: 3557.4
New salary: 3657.4

PL/SQL procedure successfully completed.
This appendix explains PL/SQL name resolution; that is, how the PL/SQL compiler resolves ambiguous references to identifiers.

An unambiguous identifier reference can become ambiguous if you change identifiers in its compilation unit (that is, if you add, rename, or delete identifiers).

Note:
The AUTHID property of a stored PL/SQL unit affects the name resolution of SQL statements that the unit issues at run time. For more information, see "Invoker's Rights and Definer's Rights (AUTHID Property)".

Topics

• Qualified Names and Dot Notation
• Column Name Precedence
• Differences Between PL/SQL and SQL Name Resolution Rules
• Resolution of Names in Static SQL Statements
• What is Capture?
• Avoiding Inner Capture in SELECT and DML Statements

B.1 Qualified Names and Dot Notation

When one named item belongs to another named item, you can (and sometimes must) qualify the name of the "child" item with the name of the "parent" item, using dot notation. For example:

<table>
<thead>
<tr>
<th>When referencing ...</th>
<th>You must qualify its name with ...</th>
<th>Using this syntax ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field of a record</td>
<td>Name of the record</td>
<td>record_name.field_name</td>
</tr>
<tr>
<td>Method of a collection</td>
<td>Name of the collection</td>
<td>collection_name.method</td>
</tr>
<tr>
<td>Pseudocolumn CURRVAL</td>
<td>Name of a sequence</td>
<td>sequence_name.CURRVAL</td>
</tr>
<tr>
<td>Pseudocolumn NEXTVAL</td>
<td>Name of a sequence</td>
<td>sequence_name.NEXTVAL</td>
</tr>
</tbody>
</table>

If an identifier is declared in a named PL/SQL unit, you can qualify its simple name (the name in its declaration) with the name of the unit (block, subprogram, or package), using this syntax:

unit_name.simple_identifier_name
If the identifier is not visible, then you must qualify its name (see “Scope and Visibility of Identifiers”).

If an identifier belongs to another schema, then you must qualify its name with the name of the schema, using this syntax:

```
schema_name.package_name
```

A simple name can be qualified with multiple names, as Example B-1 shows.

Some examples of possibly ambiguous qualified names are:

- Field or attribute of a function return value, for example:
  ```
  func_name().field_name
  func_name().attribute_name
  ```

- Schema object owned by another schema, for example:
  ```
  schema_name.table_name
  schema_name.procedure_name()
  schema_name.type_name.member_name()
  ```

- Package object owned by another user, for example:
  ```
  schema_name.package_name.procedure_name()
  schema_name.package_name.record_name.field_name
  ```

- Record containing an ADT, for example:
  ```
  record_name.field_name.attribute_name
  record_name.field_name.member_name()
  ```

Example B-1  Qualified Names

```sql
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  m NUMBER;
  TYPE t1 IS RECORD (a NUMBER);
  v1 t1;
  TYPE t2 IS TABLE OF t1 INDEX BY PLS_INTEGER;
  v2 t2;
  FUNCTION f1 (p1 NUMBER) RETURN t1;
  FUNCTION f2 (q1 NUMBER) RETURN t2;
END pkg1;
/
```

```sql
CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1 (p1 NUMBER) RETURN t1 IS
    n NUMBER;
    BEGIN
      n := m;             -- Unqualified variable name
      n := pkg1.m;        -- Variable name qualified by package name
      n := pkg1.f1.p1;    -- Parameter name qualified by function name, -- which is qualified by package name
      n := v1.a;          -- Variable name qualified by component name
      n := pkg1.v1.a;     -- Variable name qualified by package name -- and followed by component name
      n := v2(10).a;      -- Indexed name followed by component name
      n := f1(10).a;      -- Function invocation followed by component name
      n := f2(10)(10).a;  -- Function invocation followed by indexed name -- and followed by component name
      n := hr.pkg1.f2(10)(10).a;  -- Schema name, package name, -- function invocation, index, component name
    v1.a := p1;
```
RETURN v1;
END f1;

FUNCTION f2 (q1 NUMBER) RETURN t2 IS
  v_t1 t1;
  v_t2 t2;
BEGIN
  v_t1.a := q1;
  v_t2(1) := v_t1;
  RETURN v_t2;
END f2;
END pkg1;
/

B.2 Column Name Precedence

If a SQL statement references a name that belongs to both a column and either a local variable or formal parameter, then the column name takes precedence.

⚠️ Caution:

When a variable or parameter name is interpreted as a column name, data can be deleted, changed, or inserted unintentionally.

In Example B-2, the name `last_name` belongs to both a local variable and a column (names are not case-sensitive). Therefore, in the WHERE clause, both references to `last_name` resolve to the column, and all rows are deleted.

Example B-3 solves the problem in Example B-2 by giving the variable a different name.

Example B-4 solves the problem in Example B-2 by labeling the block and qualifying the variable name with the block name.

In Example B-5, the function `dept_name` has a formal parameter and a local variable whose names are those of columns of the table `DEPARTMENTS`. The parameter and variable name are qualified with the function name to distinguish them from the column names.

Example B-2  Variable Name Interpreted as Column Name Causes Unintended Result

DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT LAST_NAME FROM employees;

DECLARE
  last_name VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/

Result:
Example B-3  Fixing Example B-2 with Different Variable Name

```
DECLARE
  v_last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = v_last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.);
END;
/
```

Result:

Deleted 2 rows.

Example B-4  Fixing Example B-2 with Block Label

```
<<main>>
DECLARE
  last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE last_name = main.last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.);
END;
/
```

Result:

Deleted 2 rows.

Example B-5  Subprogram Name for Name Resolution

```
DECLARE
  FUNCTION dept_name (department_id IN NUMBER)
  RETURN departments.department_name%TYPE
  IS
    department_name  departments.department_name%TYPE;
    BEGIN
      SELECT department_name INTO dept_name.department_name
      -- ^column ^local variable
      FROM departments
      WHERE department_id = dept_name.department_id;
      -- ^column ^formal parameter
      RETURN department_name;
    END dept_name;
BEGIN
  FOR item IN (SELECT department_id
    FROM departments
    ORDER BY department_name) LOOP
    DBMS_OUTPUT.PUT_LINE ('Department: ' || dept_name(item.department_id));
  END LOOP;
END;
/
```

Result:
B.3 Differences Between PL/SQL and SQL Name Resolution Rules

PL/SQL and SQL name resolution rules are very similar. However:

- PL/SQL rules are less permissive than SQL rules.
  Because most SQL rules are context-sensitive, they recognize as legal more situations than PL/SQL rules do.

- PL/SQL and SQL resolve qualified names differently.
  For example, when resolving the table name `HR.JOBS`:
  - PL/SQL searches first for packages, types, tables, and views named `HR` in the current schema, then for public synonyms, and finally for objects named `JOBS` in the `HR` schema.
  - SQL searches first for objects named `JOBS` in the `HR` schema, and then for packages, types, tables, and views named `HR` in the current schema.

To avoid problems caused by the few differences between PL/SQL and SQL name resolution rules, follow the recommendations in "Avoiding Inner Capture in SELECT and DML Statements".
B.4 Resolution of Names in Static SQL Statements

Static SQL is described in PL/SQL Static SQL.

When the PL/SQL compiler finds a static SQL statement:

1. If the statement is a SELECT statement, the PL/SQL compiler removes the INTO clause.
2. The PL/SQL compiler sends the statement to the SQL subsystem.
3. The SQL subsystem checks the syntax of the statement.
   If the syntax is incorrect, the compilation of the PL/SQL unit fails. If the syntax is correct, the SQL subsystem determines the names of the tables and tries to resolve the other names in the scope of the SQL statement.
4. If the SQL subsystem cannot resolve a name in the scope of the SQL statement, then it sends the name back to the PL/SQL compiler. The name is called an escaped identifier.
5. The PL/SQL compiler tries to resolve the escaped identifier.
   First, the compiler tries to resolve the identifier in the scope of the PL/SQL unit. If that fails, the compiler tries to resolve the identifier in the scope of the schema. If that fails, the compilation of the PL/SQL unit fails.
6. If the compilation of the PL/SQL unit succeeds, the PL/SQL compiler generates the text of the regular SQL statement that is equivalent to the static SQL statement and stores that text with the generated computer code.
7. At run time, the PL/SQL runtime system invokes routines that parse, bind, and run the regular SQL statement.
   The bind variables are the escaped identifiers (see step 4).
8. If the statement is a SELECT statement, the PL/SQL runtime system stores the results in the PL/SQL targets specified in the INTO clause that the PL/SQL compiler removed in step 1.

Note:
Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.
B.5 What is Capture?

When a declaration or definition prevents the compiler from correctly resolving a reference in another scope, the declaration or definition is said to capture the reference. Capture is usually the result of migration or schema evolution.

Topics

• Outer Capture
• Same-Scope Capture
• Inner Capture

Note:

Same-scope and inner capture occur only in SQL scope.

B.5.1 Outer Capture

Outer capture occurs when a name in an inner scope, which had resolved to an item in an inner scope, now resolves to an item in an outer scope. Both PL/SQL and SQL are designed to prevent outer capture; you need not be careful to avoid it.

B.5.2 Same-Scope Capture

Same-scope capture occurs when a column is added to one of two tables used in a join, and the new column has the same name as a column in the other table. When only one table had a column with that name, the name could appear in the join unqualified. Now, to avoid same-scope capture, you must qualify the column name with the appropriate table name, everywhere that the column name appears in the join.

B.5.3 Inner Capture

Inner capture occurs when a name in an inner scope, which had resolved to an item in an outer scope, now either resolves to an item in an inner scope or cannot be resolved. In the first case, the result might change. In the second case, an error occurs.

In Example B-6, a new column captures a reference to an old column with the same name. Before new column col2 is added to table tab2, col2 resolves to tab1.col2; afterward, it resolves to tab2.col2.

To avoid inner capture, follow the rules in "Avoiding Inner Capture in SELECT and DML Statements".

Example B-6  Inner Capture of Column Reference

Table tab1 has a column named col2, but table tab2 does not:

```
DROP TABLE tab1;
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER);
INSERT INTO tab1 (col1, col2) VALUES (100, 10);
```
DROP TABLE tab2;
CREATE TABLE tab2 (col1 NUMBER);
INSERT INTO tab2 (col1) VALUES (100);

Therefore, in the inner SELECT statement, the reference to col2 resolves to column tab1.col2:

CREATE OR REPLACE PROCEDURE proc AUTHID DEFINER AS
CURSOR c1 IS
  SELECT * FROM tab1
  WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
  OPEN c1;
  CLOSE c1;
END;
/

Add a column named col2 to table tab2:

ALTER TABLE tab2 ADD (col2 NUMBER);

Now procedure proc is invalid. At its next invocation, the database automatically recompiles it, and the reference to col2 in the inner SELECT statement resolves to column tab2.col2.

B.6 Avoiding Inner Capture in SELECT and DML Statements

Avoid inner capture of references in SELECT, SELECT INTO, and DML statements by following these recommendations:

- Specify a unique alias for each table in the statement.
- Do not specify a table alias that is the name of a schema that owns an item referenced in the statement.
- Qualify each column reference in the statement with the appropriate table alias.

In Example B-7, schema hr owns tables tab1 and tab2. Table tab1 has a column named tab2, whose Abstract Data Type (ADT) has attribute a. Table tab2 does not have a column named a. Against recommendation, the query specifies alias hr for table tab1 and references table tab2. Therefore, in the query, the reference hr.tab2.a resolves to table tab1, column tab2, attribute a. Then the example adds column a to table tab2. Now the reference hr.tab2.a in the query resolves to schema hr, table tab2, column a. Column a of table tab2 captures the reference to attribute a in column tab2 of table tab1.

Topics

- Qualifying References to Attributes and Methods
- Qualifying References to Row Expressions

Example B-7    Inner Capture of Attribute Reference

CREATE OR REPLACE TYPE type1 AS OBJECT (a NUMBER);
/
DROP TABLE tab1;
CREATE TABLE tab1 (tab2 type1);
INSERT INTO tab1 (tab2) VALUES (type1(10));

DROP TABLE tab2;
CREATE TABLE tab2 (x NUMBER);
INSERT INTO tab2 (x) VALUES (10);

/* Alias tab1 with same name as schema name, a bad practice used here for illustration purpose. Note lack of alias in second SELECT statement. */
SELECT * FROM hr.tab1
WHERE EXISTS (SELECT * FROM hr.tab2 WHERE x = hr.tab2.a);

Result:
TAB2(A)
 --------------
TYPE1(10)
1 row selected.

Add a column named a to table tab2 (which belongs to schema hr):
ALTER TABLE tab2 ADD (a NUMBER);

Now, when the query runs, hr.tab2.a resolves to schema hr, table tab2, column a. To avoid this inner capture, apply the recommendations to the query:
SELECT * FROM hr.tab1 p1
WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);

B.6.1 Qualifying References to Attributes and Methods

To reference an attribute or method of a table element, you must give the table an alias and use the alias to qualify the reference to the attribute or method.

In Example B-8, table tbl1 has column col1 of data type t1, an ADT with attribute x. The example shows several correct and incorrect references to tbl1.col1.x.

Example B-8 Qualifying ADT Attribute References

CREATE OR REPLACE TYPE t1 AS OBJECT (x NUMBER);
/
DROP TABLE tbl1;
CREATE TABLE tbl1 (col1 t1);

The references in the following INSERT statements do not need aliases, because they have no column lists:

BEGIN
  INSERT INTO tbl1 VALUES ( t1(10) );
  INSERT INTO tbl1 VALUES ( t1(20) );
  INSERT INTO tbl1 VALUES ( t1(30) );
END;
/

The following references to the attribute x cause error ORA-00904:
UPDATE tbl SET **coll.x** = 10 WHERE **coll.x** = 20;
UPDATE tbl SET **tbl.coll.x** = 10 WHERE **tbl.coll.x** = 20;
UPDATE hr.tbl SET **hr.tbl.coll.x** = 10 WHERE **hr.tbl.coll.x** = 20;
DELETE FROM tbl WHERE **tbl.coll.x** = 10;

The following references to the attribute **x**, with table aliases, are correct:
UPDATE hr.tbl **t** SET **t.coll.x** = 10 WHERE **t.coll.x** = 20;

DECLARE
    y NUMBER;
BEGIN
    SELECT **t.coll.x** INTO y FROM tbl **t** WHERE **t.coll.x** = 30;
END;
/

DELETE FROM tbl **t** WHERE **t.coll.x** = 10;

**B.6.2 Qualifying References to Row Expressions**

Row expressions must resolve as references to table aliases. A row expression can appear in the `SET` clause of an `UPDATE` statement or be the parameter of the SQL function `REF` or `VALUE`.

In **Example B-9**, table `ot1` is a standalone nested table of elements of data type `t1`, an ADT with attribute **x**. The example shows several correct and incorrect references to row expressions.

**Example B-9   Qualifying References to Row Expressions**

CREATE OR REPLACE TYPE t1 AS OBJECT (**x** NUMBER);
/
DROP TABLE `ot1`;
CREATE TABLE `ot1` OF T1;

BEGIN
    INSERT INTO `ot1` VALUES (t1(10));
    INSERT INTO `ot1` VALUES (20);
    INSERT INTO `ot1` VALUES (30);
END;
/

The following references cause error ORA-00904:
UPDATE `ot1` SET **VALUE**(ot1.**x**) = t1(20) WHERE **VALUE**(ot1.**x**) = t1(10);
DELETE FROM `ot1` WHERE **VALUE**(ot1) = (t1(10));

The following references, with table aliases, are correct:

UPDATE `ot1` **o** SET **o.x** = 10;

DECLARE
    n_ref  REF t1;
BEGIN
    SELECT REF(**o**) INTO n_ref FROM `ot1` **o** WHERE **VALUE**(o) = t1(30);
END;
DECLARE
  n t1;
BEGIN
  SELECT VALUE(o) INTO n FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/

DECLARE
  n NUMBER;
BEGIN
  SELECT o.x INTO n FROM ot1 o WHERE o.x = 30;
END;
/

DELETE FROM ot1 o WHERE VALUE(o) = t1(20);
This appendix describes the program limits that are imposed by the PL/SQL language. PL/SQL is based on the programming language Ada. As a result, PL/SQL uses a variant of Descriptive Intermediate Attributed Notation for Ada (DIANA), a tree-structured intermediate language. It is defined using a metanotation called Interface Definition Language (IDL). DIANA is used internally by compilers and other tools.

At compile time, PL/SQL source text is translated into system code. Both the DIANA and system code for a subprogram or package are stored in the database. At run time, they are loaded into the shared memory pool. The DIANA is used to compile dependent subprograms; the system code simply runs.

In the shared memory pool, a package specification, ADT specification, standalone subprogram, or anonymous block is limited to 67108864 (2**26) DIANA nodes which correspond to tokens such as identifiers, keywords, operators, and so on. This allows for ~6,000,000 lines of code unless you exceed limits imposed by the PL/SQL compiler, some of which are given in Table C-1.

### Table C-1  PL/SQL Compiler Limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind variables passed to a program unit</td>
<td>32768</td>
</tr>
<tr>
<td>exception handlers in a program unit</td>
<td>65536</td>
</tr>
<tr>
<td>fields in a record</td>
<td>65536</td>
</tr>
<tr>
<td>levels of block nesting</td>
<td>255</td>
</tr>
<tr>
<td>levels of record nesting</td>
<td>32</td>
</tr>
<tr>
<td>levels of subquery nesting</td>
<td>254</td>
</tr>
<tr>
<td>levels of label nesting</td>
<td>98</td>
</tr>
<tr>
<td>levels of nested collections</td>
<td>no predefined limit</td>
</tr>
<tr>
<td>magnitude of a PLS_INTEGER or BINARY_INTEGER value</td>
<td>-2147483648..2147483647</td>
</tr>
<tr>
<td>number of formal parameters in an explicit cursor, function, or procedure</td>
<td>65536</td>
</tr>
<tr>
<td>objects referenced by a program unit</td>
<td>65536</td>
</tr>
<tr>
<td>precision of a FLOAT value (binary digits)</td>
<td>126</td>
</tr>
<tr>
<td>precision of a NUMBER value (decimal digits)</td>
<td>38</td>
</tr>
<tr>
<td>precision of a REAL value (binary digits)</td>
<td>63</td>
</tr>
<tr>
<td>size of an identifier (bytes)</td>
<td>128</td>
</tr>
<tr>
<td>size of a string literal (bytes)</td>
<td>32767</td>
</tr>
<tr>
<td>size of a CHAR value (bytes)</td>
<td>32767</td>
</tr>
<tr>
<td>size of a LONG value (bytes)</td>
<td>32760</td>
</tr>
</tbody>
</table>
Table C-1  (Cont.) PL/SQL Compiler Limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of a LONG RAW value (bytes)</td>
<td>32760</td>
</tr>
<tr>
<td>size of a RAW value (bytes)</td>
<td>32767</td>
</tr>
<tr>
<td>size of a VARCHAR2 value (bytes)</td>
<td>32767</td>
</tr>
<tr>
<td>size of an NCHAR value (bytes)</td>
<td>32767</td>
</tr>
<tr>
<td>size of an NVARCHAR2 value (bytes)</td>
<td>32767</td>
</tr>
<tr>
<td>size of a BFILE value (bytes)</td>
<td>4G * value of DB_BLOCK_SIZE parameter</td>
</tr>
<tr>
<td>size of a BLOB value (bytes)</td>
<td>4G * value of DB_BLOCK_SIZE parameter</td>
</tr>
<tr>
<td>size of a CLOB value (bytes)</td>
<td>4G * value of DB_BLOCK_SIZE parameter</td>
</tr>
<tr>
<td>size of an NCLOB value (bytes)</td>
<td>4G * value of DB_BLOCK_SIZE parameter</td>
</tr>
<tr>
<td>size of a trigger</td>
<td>32 K</td>
</tr>
</tbody>
</table>

To estimate how much memory a program unit requires, you can query the static data dictionary view USER_OBJECT_SIZE. The column PARSED_SIZE returns the size (in bytes) of the “flattened” DIANA. For example:

```sql
CREATE OR REPLACE PACKAGE pkg1 AS
    TYPE numset_t IS TABLE OF NUMBER;
    FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/

CREATE PACKAGE BODY pkg1 AS
    -- FUNCTION f1 returns a collection of elements (1,2,3,... x)
    FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
    BEGIN
        FOR i IN 1..x LOOP
            PIPE ROW(i);
        END LOOP;
        RETURN;
    END f1;
END pkg1;
/

SQL*Plus commands for formatting results of next query:

COLUMN name FORMAT A4
COLUMN type FORMAT A12
COLUMN source_size FORMAT 999
COLUMN parsed_size FORMAT 999
COLUMN code_size FORMAT 999
COLUMN error_size FORMAT 999

Query:

SELECT * FROM user_object_size WHERE name = 'PKG1' ORDER BY type;

Result:

NAME  TYPE  SOURCE_SIZE  PARSED_SIZE  CODE_SIZE  ERROR_SIZE
-----  ----  -----------  ----------  --------  ---------
Unfortunately, you cannot estimate the number of DIANA nodes from the parsed size. Two program units with the same parsed size might require 1500 and 2000 DIANA nodes, respectively because, for example, the second unit contains more complex SQL statements.

When a PL/SQL block, subprogram, package, or schema-level user-defined type exceeds a size limit, you get an error such as PLS-00123: program too large. Typically, this problem occurs with packages or anonymous blocks. With a package, the best solution is to divide it into smaller packages. With an anonymous block, the best solution is to redefine it as a group of subprograms, which can be stored in the database.

For more information about the limits on data types, see PL/SQL Data Types.
**PL/SQL Reserved Words and Keywords**

Reserved words (listed in Table D-1) and keywords (listed in Table D-2) are identifiers that have special meaning in PL/SQL. They are case-insensitive. For more information about them, see "Reserved Words and Keywords".

---

**Note:**

Some of the words in this appendix are also reserved by SQL. You can display them with the dynamic performance view `V$RESERVED_WORDS`. For information about this view, see *Oracle Database Reference*.

---

**Table D-1  PL/SQL Reserved Words**

<table>
<thead>
<tr>
<th>Begins with</th>
<th>Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ALL, ALTER, AND, ANY, AS, ASC, AT</td>
</tr>
<tr>
<td>B</td>
<td>BEGIN, BETWEEN, BY</td>
</tr>
<tr>
<td>C</td>
<td>CASE, CHECK, CLUSTERS, CLUSTER, COLAUTH, COLUMNS, COMPRESS, CONNECT, CRASH, CREATE, CURSOR</td>
</tr>
<tr>
<td>D</td>
<td>DECLARE, DEFAULT, DESC, DISTINCT, DROP</td>
</tr>
<tr>
<td>E</td>
<td>ELSE, END, EXCEPTION, EXCLUSIVE</td>
</tr>
<tr>
<td>F</td>
<td>FETCH, FOR, FROM, FUNCTION</td>
</tr>
<tr>
<td>G</td>
<td>GOTO, GRANT, GROUP</td>
</tr>
<tr>
<td>H</td>
<td>HAVING</td>
</tr>
<tr>
<td>I</td>
<td>IDENTIFIED, IF, IN, INDEX, INDEXES, INSERT, INTERSECT, INTO, IS</td>
</tr>
<tr>
<td>L</td>
<td>LIKE, LOCK</td>
</tr>
<tr>
<td>M</td>
<td>MINUS, MODE</td>
</tr>
<tr>
<td>N</td>
<td>NOCOMPRESS, NOT, NOWAIT, NULL</td>
</tr>
<tr>
<td>O</td>
<td>OF, ON, OPTION, OR, ORDER, OVERLAPS</td>
</tr>
<tr>
<td>P</td>
<td>PROCEDURE, PUBLIC</td>
</tr>
<tr>
<td>R</td>
<td>RESOURCE, REVOKE</td>
</tr>
<tr>
<td>S</td>
<td>SELECT, SHARE, SIZE, SQL, START, SUBTYPE</td>
</tr>
<tr>
<td>T</td>
<td>TABAUTH, TABLE, THEN, TO, TYPE</td>
</tr>
<tr>
<td>U</td>
<td>UNION, UNIQUE, UPDATE</td>
</tr>
<tr>
<td>V</td>
<td>VALUES, VIEW, VIEWS</td>
</tr>
<tr>
<td>W</td>
<td>WHEN, WHERE, WITH</td>
</tr>
</tbody>
</table>
## Table D-2  PL/SQL Keywords

<table>
<thead>
<tr>
<th>Begins with</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A, ADD, ACCESSIBLE, AGENT, AGGREGATE, ARRAY, ATTRIBUTE, AUTHID, AVG</td>
</tr>
<tr>
<td>B</td>
<td>BFILE_BASE, BINARY, BLOB_BASE, BLOCK, BODY, BOTH, BOUND, BULK, BYTE</td>
</tr>
<tr>
<td>C</td>
<td>C, CALL, CALLING, CASCADE, CHAR, CHAR_BASE, CHARACTER, CHARSET, CHARSETFORM, CHARSETID, CLOB_BASE, CLONE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONTINUE, CONV, COUNT, CREDENTIAL, CURRENT, CUSTOMDATUM</td>
</tr>
<tr>
<td>D</td>
<td>DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DELETE, DETERMINISTIC, DIRECTORY, DOUBLE, DURATION</td>
</tr>
<tr>
<td>E</td>
<td>ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPTIONS, EXECUTE, EXISTS, EXIT, EXTERNAL</td>
</tr>
<tr>
<td>F</td>
<td>FINAL, FIRST, FIXED, FLOAT, FORALL, FORCE</td>
</tr>
<tr>
<td>G</td>
<td>GENERAL</td>
</tr>
<tr>
<td>H</td>
<td>HASH, HEAP, HIDDEN, HOUR</td>
</tr>
<tr>
<td>I</td>
<td>IMMEDIATE, IMMUTABLE, INCLUDING, INDICATOR, INDICES, INFINITE, INSTANTIABLE, INT, INTERFACE, INTERVAL, INVALIDATE, ISOLATION</td>
</tr>
<tr>
<td>J</td>
<td>JAVA</td>
</tr>
<tr>
<td>L</td>
<td>LANGUAGE, LARGE, LEADING, LENGTH, LEVEL, LIBRARY, LIKE2, LIKE4, LIKEC, LIMIT, LIMITED, LOCAL, LONG, LOOP</td>
</tr>
<tr>
<td>M</td>
<td>MAP, MAX, MAXLEN, MEMBER, MERGE, MIN, MINUTE, MOD, MODIFY, MONTH, MULTISET, MUTABLE</td>
</tr>
<tr>
<td>N</td>
<td>NAME, NAN, NATIONAL, NATIVE, NCHAR, NEW, NOCOPY, NUMBER_BASE</td>
</tr>
<tr>
<td>O</td>
<td>OBJECT, OCICOLL, OCIDATE, OCIDATETIME, OCIDURATION, OCIINTERVAL, OCILOBLOCATOR, OCINUMBER, OCIRAW, OCIREF, OCIREFCURSOR, OCIROWID, OCISTRING, OCTYPE, OLD, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLVAR, OTHERS, OUT, OVERRIDING</td>
</tr>
<tr>
<td>P</td>
<td>PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARENT, PARTITION, PASCAL, PERSISTABLE, PIPE, PIPELINED, PLUGGABLE, POLYMORPHIC, PRAGMA, PRECISION, PRIOR, PRIVATE</td>
</tr>
<tr>
<td>R</td>
<td>RAISE, RANGE, RAW, READ, RECORD, REF, REFERENCE, RELIES_ON, REM, REMAINDER, RENAME, RESULT, RESULT_CACHE, RETURN, RETURNING, REVERSE, ROLLBACK, ROW</td>
</tr>
<tr>
<td>S</td>
<td>SAMPLE, SAVE, SAVEPOINT, SB1, SB2, SB4, SECOND, SEGMENT, SELF, SEPARATE, SEQUENCE, SERIALIZABLE, SET, SHORT, SIZE_T, SOME, SPARSE, SQLCODE, SQLDATA, SQLNAME, SQLSTATE, STANDARD, STATIC, STDDEV, STORED, STRING, STRUCT, STYLE, SUBMULTISET, SUBPARTITION, SUBSTITUTE, SUM, SYNONYM</td>
</tr>
<tr>
<td>T</td>
<td>TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABR, TIMEZONE_HOUR, TIMEZONE_MIN, TIMEZONE_REGION, TRAILING, TRANSACTION, TRANSACTIONAL, TRUSTED</td>
</tr>
<tr>
<td>U</td>
<td>UB1, UB2, UB4, UNDER, UNPLUG, UNSIGNED, UNTRUSTED, USE, USING</td>
</tr>
<tr>
<td>V</td>
<td>VALIST, VALUE, VARIABLE, VARIANCE, VARRAY, VARYING, VOID</td>
</tr>
<tr>
<td>W</td>
<td>WHILE, WORK, WRAPPED, WRITE</td>
</tr>
<tr>
<td>Y</td>
<td>YEAR</td>
</tr>
<tr>
<td>Z</td>
<td>ZONE</td>
</tr>
</tbody>
</table>
PL/SQL Predefined Data Types

This appendix groups by data type family the data types and subtypes that the package STANDARD predefines.

Constants
This constant defines the maximum name length possible.

```plsql
ORA_MAX_NAME_LEN CONSTANT PLS_INTEGER := 128;
```

BFILE Data Type Family

type BFILE is BFILE_BASE;

BLOB Data Type Family

type BLOB is BLOB_BASE;

subtype "BINARY LARGE OBJECT" is BLOB;

BOOLEAN Data Type Family

type BOOLEAN is (FALSE, TRUE);

CHAR Data Type Family

type VARCHAR2 is new CHAR_BASE;

type MLSLABEL is new CHAR_BASE;

type UROWID is new CHAR_BASE;

DBMS_ID and DBMS_QUOTED_ID define the length of identifiers in objects for SQL, PL/SQL and users.

- subtype DBMS_ID is VARCHAR2(ORA_MAX_NAME_LEN);
- subtype DBMS_QUOTED_ID is VARCHAR2(ORA_MAX_NAME_LEN+2);

DBMS_ID_30 and DBMS_QUOTED_ID_30 define the length of SQL objects whose limits is 30 bytes.

- subtype DBMS_ID_30 is VARCHAR2(30);
- subtype DBMS_QUOTED_ID_30 is VARCHAR2(32);

- subtype VARCHAR is VARCHAR2;
- subtype STRING is VARCHAR2;
- subtype LONG is VARCHAR2(32760);
- subtype RAW is VARCHAR2;
- subtype "LONG RAW" is RAW(32760);
- subtype ROWID is VARCHAR2(256);
- subtype CHAR is VARCHAR2;
- subtype CHARACTER is CHAR;
- subtype "CHARACTER VARYING" is VARCHAR;
- subtype "CHAR VARYING" is VARCHAR;
- subtype "NATIONAL CHARACTER" is CHAR CHARACTER SET NCHAR_CS;
- subtype "NATIONAL CHAR" is CHAR CHARACTER SET NCHAR_CS;
subtype "NCHAR" is CHAR CHARACTER SET NCHAR_CS;
subtype "NVARCHAR2" is VARCHAR2 CHARACTER SET NCHAR_CS;

CLOB Data Type Family

type CLOB is CLOB_BASE;
subtype "CHARACTER LARGE OBJECT" is CLOB;
subtype "CHAR LARGE OBJECT" is CLOB;
subtype "NATIONAL CHARACTER LARGE OBJECT" is CLOB CHARACTER SET NCHAR_CS;
subtype "NCHAR LARGE OBJECT" is CLOB CHARACTER SET NCHAR_CS;
subtype "NCLOB" is CLOB CHARACTER SET NCHAR_CS;

DATE Data Type Family

type DATE is DATE_BASE;
type TIMESTAMP is new DATE_BASE;
type "TIMESTAMP WITH TIME ZONE" is new DATE_BASE;
type "INTERVAL YEAR TO MONTH" is new DATE_BASE;
type "INTERVAL DAY TO SECOND" is new DATE_BASE;
type "TIMESTAMP WITH LOCAL TIME ZONE" is new DATE_BASE;
subtype TIME_UNCONSTRAINED is TIME(9);
subtype TIME_TZ_UNCONSTRAINED is TIME(9) WITH TIME ZONE;
subtype TIMESTAMP_UNCONSTRAINED is TIMESTAMP(9);
subtype TIMESTAMP_TZ_UNCONSTRAINED is TIMESTAMP(9) WITH TIME ZONE;
subtype YMINTERVAL_UNCONSTRAINED is INTERVAL YEAR (9) TO MONTH;
subtype DSINTERVAL_UNCONSTRAINED is INTERVAL DAY (9) TO SECOND (9);
subtype TIMESTAMP_LTZ_UNCONSTRAINED is TIMESTAMP(9) WITH LOCAL TIME ZONE;

JSON Data Type Family

type JSON is BLOB_BASE;

NUMBER Data Type Family

type NUMBER is NUMBER_BASE;
subtype FLOAT is NUMBER; -- NUMBER(126)
subtype REAL is FLOAT; -- FLOAT(63)
subtype "DOUBLE PRECISION" is FLOAT;

subtype INTEGER is NUMBER(38,0);
subtype INT is INTEGER;
subtype SMALLINT is NUMBER(38,0);

subtype DECIMAL is NUMBER(38,0);
subtype NUMERIC is DECIMAL;
subtype DEC is DECIMAL;

subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
subtype NATURAL is BINARY_INTEGER range 0..2147483647;
subtype NATURALN is NATURAL not null;
subtype POSITIVE is BINARY_INTEGER range 1..2147483647;
subtype POSITIVEN is POSITIVE not null;
subtype SIGNTYPE is BINARY_INTEGER range '-1'..1; -- for SIGN functions
subtype PLS_INTEGER is BINARY_INTEGER;

type BINARY_FLOAT is NUMBER;
type BINARY_DOUBLE is NUMBER;

subtype SIMPLE_INTEGER is BINARY_INTEGER NOT NULL;
subtype SIMPLE_FLOAT   is BINARY_FLOAT   NOT NULL;
subtype SIMPLE_DOUBLE  is BINARY_DOUBLE  NOT NULL;

See Also:

- PL/SQL Data Types for more information about PL/SQL data types
- “User-Defined PL/SQL Subtypes” for information that also applies to predefined subtypes
Index

Symbols

_ wildcard character, 3-40
% wildcard character, 3-40
%BULK_EXCEPTIONS cursor attribute, 13-21
%BULK_ROWCOUNT cursor attribute, 13-24
%FOUND cursor attribute
  for implicit cursor, 7-7
  for named cursor, 7-22
%ISOPEN cursor attribute
  for implicit cursor, 7-7
  for named cursor, 7-21
%NOTFOUND cursor attribute
  for implicit cursor, 7-8
  for named cursor, 7-22
%ROWCOUNT cursor attribute
  for implicit cursor, 7-8
  for named cursor, 7-23
%ROWTYPE attribute, 6-55
  column alias and, 7-15
  explicit cursor and, 7-15
  invisible columns and, 6-60
  syntax diagram, 14-171
  virtual columns and, 6-59
%TYPE attribute, 3-18
  initial value and, 3-18
  NOT NULL constraint and, 3-18
  syntax diagram, 14-189
$$PLSQL_LINE inquiry directive, 3-57
$$PLSQL_UNIT inquiry directive, 3-57
$$PLSQL_UNIT_OWNER inquiry directive, 3-57
$$PLSQL_UNIT_TYPE inquiry directive, 3-57

ACCESSIBLE BY clause (continued)
  in package specification, 11-1
  in subprogram, 9-2
accessor, 14-22
ADT
  See Abstract Data Type (ADT)
aggregate, 6-17
AGGREGATE clause, 14-27
  in CREATE FUNCTION statement, 15-25
aggregate function, 15-25
  in PL/SQL expression, 3-47
  pipelined table function as, 13-48
SQL%NOTFOUND attribute and, 7-8
alias
  column
    in cursor FOR LOOP, 7-26
    in explicit cursor, 7-15
  table
    for avoiding inner capture, B-8
    for row expression, B-10
    for table element attribute or method, B-9
aliasing (problem)
  SELECT BULK COLLECT INTO statement
    and, 13-28
subprogram parameter, 9-20
ALTER FUNCTION statement, 15-2
ALTER LIBRARY statement, 15-4
ALTER PACKAGE statement, 15-6
ALTER PROCEDURE statement, 15-8
ALTER TRIGGER statement, 15-10
ALTER TYPE statement, 15-12
AND operator, 3-32
anonymous block, 2-4
  AUTHID property and, 9-50
ANYDATA data type, 13-43
ANYDATASET data type, 13-43
ANYTYPE data type, 13-43
application common object
  metadata-linked, 14-177
architecture of PL/SQL, 2-10
array
  associative
    See associative array, 6-2
  non-PL/SQL, 6-2

Abstract Data Type (ADT), 2-8, 15-12
  creating, 15-66
accent-insensitive comparison, 3-39
ACCESS INTO NULL exception, 12-11
ACCESSIBLE BY clause, 11-1, 14-22
  in CREATE FUNCTION statement, 15-25
  in CREATE PACKAGE statement, 15-35
  in CREATE PROCEDURE statement, 15-42
  in CREATE TYPE BODY statement, 15-77
  in CREATE TYPE statement, 15-66
assignment of value
to composite variable
collection, 6-22
record, 6-61
to scalar variable, 3-25
assignment statement, 3-25
syntax diagram, 14-28
associative array, 6-4
characteristics of, 6-2
comparisons, 6-28
declaring constant, 6-7
FIRST and LAST methods for, 6-40
in FORALL statement, 14-124
NLS parameters and, 6-8
See also collection
atomic (lexical) unit, 3-4
atomically null collection
See null collection
attribute
%ROWTYPE
See %ROWTYPE attribute, 6-55
%TYPE
See %TYPE attribute, 3-18
cursor
See cursor attribute, 7-5
AUTHID property, 9-50
syntax diagram, 14-44
autonomous routine, 7-55
declaring, 7-57
autonomous transaction, 7-55
controlling, 7-58
pipelined table function in, 14-153
autonomous trigger, 7-60
AUTONOMOUS_TRANSACTION pragma, 7-57
for pipelined table function, 13-43
syntax diagram, 14-31

bind variable
avoiding SQL injection with, 8-24
placeholder for
See placeholder for bind variable, 8-2
blank-padding
in assignment, 4-6
in comparison, 4-7
block, 2-4
syntax diagram, 14-35
BOOLEAN data type, 4-8
BOOLEAN expression, 3-43
BOOLEAN static expression, 3-52
BOOLEAN variable, 3-28
built-in function
See SQL function
bulk binding, 13-12
BULK COLLECT clause, 13-26, 13-38
aliasing and, 13-28
of FETCH statement, 13-34
of RETURNING INTO clause
FORALL statement and, 13-39
of SELECT INTO statement, 13-26
query result set processing and, 7-26
that returns no rows, 13-26
bulk SQL, 13-12
in compound DML trigger, 10-12

C
declaration, 14-44
c procedure, invoking, 9-54
cache, function result, 9-37
calculated column
See virtual column
call specification, 9-54
in CREATE FUNCTION statement, 15-25
in CREATE PROCEDURE statement, 15-42
in CREATE TYPE statement, 15-66
in package, 11-1
syntax diagram, 14-44
call stack, AUTHID property and, 9-50
capture, B-7
cascading triggers, 10-45
CASE expression
searched, 3-46
simple, 3-44
case sensitivity
character comparison and, 3-39
character literal and, 3-10
identifier and, 3-6
quoted user-defined identifier, 3-8
keyword and, D-1
LIKE operator and, 3-40
reserved word and, D-1

B
data structure, 6-2
type, 4-1
basic LOOP statement, 5-10
syntax diagram, 14-32
BETWEEN operator, 3-42
BINARY_DOUBLE data type
predefined constants for, 4-3
subtype of, 4-3
tuning code and, 13-9
BINARY_FLOAT data type
predefined constants for, 4-3
subtype of, 4-3
tuning code and, 13-9
BINARY_INTEGER data type
See PLS_INTEGER data type
CASE statement, 5-1
searched, 5-7
  syntax diagram, 14-47
simple, 5-6
  IF THEN ELSIF statement and, 5-5
  syntax diagram, 14-47
CASE_NOT_FOUND exception, 12-11
case-insensitive comparison, 3-39
CHAR data type, 4-4
CHAR data type family, E-1
code, 3-1
code literal, 3-10
  See also string
code set, 3-1
CLOB data type and comparison operator, 3-37
CLOB data type family, E-1
CLOSE statement, 14-50
collating sequence, 14-50
collation, 14-75
collection, 6-1, 6-4, 6-10, 6-12
  as public package item, 6-49
  assigning one to another, 6-22
  comparing one to another, 6-28
  cursor variable and, 14-72
  declaration syntax, 14-54
  empty, 6-2
  creating with constructor, 6-16
  index
    See index collection, 13-16
  internal size of
    DELETE method and, 6-33
    EXTEND method and, 6-37
    TRIM method and, 6-36
  multidimensional, 6-26
null, 6-2
  assigning to collection variable, 6-24
  pipelined table function and, 13-42
  querying
    with dynamic SQL, 8-8
    with static SQL, 7-38
  retrieving query results into, 13-26
types of, 6-2
  collection constructor, 6-16
collection method, 6-32
  as subprogram parameter, 6-32
  invocation syntax, 14-51
null collection and, 6-32
  COLLECTION_IS_NULL exception, 12-11
column alias
  in cursor FOR LOOP, 7-26
  in explicit cursor, 7-15
COLUMNS pseudo-operator, 13-56
comment, 3-13
  nested, 3-14
  PL/SQL Wrapper utility and, A-2
comment (continued)
  syntax diagram, 14-59
  COMMIT statement, 7-45
  FOR UPDATE cursor and, 7-53
  in autonomous transaction, 7-59
  comparison operator, 3-37
  cursor variable and, 14-72
compatible data type
  for collection variables, 14-72
  for scalar variables, 3-25
compilation
  conditional, 3-55
  for native execution, 13-70
  interpreted, 13-72
  compile clause
    in ALTER FUNCTION statement, 15-2
  compile_clause syntax diagram, 14-60
  compile-time warning, 12-2
  compiler directive
    See pragma
  composite data type, 6-1
  composite variable, 6-1
  compound trigger, 10-10
  computation-intensive code, 13-9
  concatenation operator (||), 3-29
  concurrent transactions, 7-59
  condition, SQL multiset, 6-31
  conditional compilation, 3-55
  conditional compilation directive, 3-55
    error, 3-57
    inquiry, 3-57
    restrictions on, 3-63
    selection, 3-66
  conditional predicate, 10-5
  conditional selection statement, 5-1
  conditional trigger, 10-1
constant
  declaring, 3-17
    associative array, 6-7
    record, 6-51
    syntax diagram, 14-63
  initial value of, 3-17
  predefined, 4-3
  static, 3-53
    in DBMS_DB_VERSION package, 3-61
  constrained subtype, 4-15
  in performance-critical code, 13-10
  subprogram parameter and, 9-12
constraint
  cursor parameter and, 14-105
constraint (continued)
  NOT NULL
    See NOT NULL constraint, 3-15
  trigger compared to, 10-3
constructor
  See collection constructor
context of transaction, 7-56
CONTINUE statement
  syntax diagram, 14-64
CONTINUE WHEN statement
  syntax diagram, 14-64
control statement, 5-1
control token, 3-56
correlated subquery, 7-29
correlation name, 10-28
  with LONG or LONG RAW column, 10-41
    See also pseudorecord
COUNT collection method, 6-44
COVERAGE pragma, 14-66
CREATE FUNCTION statement, 15-25
CREATE LIBRARY statement, 15-31
CREATE PACKAGE statement, 15-35
CREATE TRIGGER statement, 15-45
CREATE TYPE BODY statement, 15-77
CREATE TYPE statement, 15-66
CREATE_WRAPPED procedure, A-8
crossedition trigger, 15-1
CURRENT OF clause, 7-52
  FOR UPDATE cursor and, 7-52
  ROWID pseudocolumn instead of, 7-53
CURRVAL pseudocolumn, 7-3
cursor
  explicit
    See explicit cursor, 7-9
  FOR UPDATE, 7-52
    after COMMIT or ROLLBACK, 7-53
  implicit
    See implicit cursor, 7-6
    in SERIALLY_REUSABLE package, 11-10
      named, 7-9
        pipelined table function and, 13-47
          See also explicit cursor and cursor variable, 7-9
  nested, 7-43
cursor attribute
  for cursor variable, 7-39
  for explicit cursor, 7-20
    %FOUND, 7-22
    %ISOPEN, 7-21
    %NOTFOUND, 7-22
    %ROWCOUNT, 7-23
  for implicit cursor, 7-6
    DBMS_SQL package and, 8-11
    native dynamic SQL and, 8-2
    SQL%BULK_EXCEPTIONS, 13-21
cursor attribute (continued)
  for implicit cursor (continued)
    SQL%BULK_ROWCOUNT, 13-24
    SQL%FOUND, 7-7
    SQL%ISOPEN, 7-7
    SQL%NOTFOUND, 7-8
    SQL%ROWCOUNT, 7-8
where you can use, 7-5
cURSOR expression, 7-43
cursor FOR LOOP statement, 5-10
  query result set processing with, 7-26
    recursive invocation in, 9-35
      syntax diagram, 14-70
cursor number
  converting cursor variable to, 8-16
  converting to cursor variable, 8-15
  DBMS_SQL.GET_NEXT_RESULT procedure and, 8-13
    DBMS_SQL.RETURN_RESULT procedure and, 8-12
cursor parameter, 7-16
cursor specification, 14-105
cursor variable, 7-30, 7-31
  converting cursor number to, 8-15
  converting to cursor number, 8-16
  DBMS_SQL.GET_NEXT_RESULT procedure and, 8-13
    DBMS_SQL.RETURN_RESULT procedure and, 8-12
    declaration syntax diagram, 14-72
CURSOR_ALREADY_OPEN exception, 12-11
cursor_control, 5-21

D
data abstraction, 2-7
data definition language statement
  See DDL statement
Data Pump Import and triggers, 10-47
data type, 4-1
collection
  See collection, 6-1
compatible
    for collection variables, 6-23
    for scalar variables, 3-25
composite, 6-1
object
  See Abstract Data Type (ADT), 2-8
of expression, 3-28
predefined, E-1
RECORD
  See record, 6-1
scalar, 4-1
SQL, 4-2
data type (continued)
user-defined
See Abstract Data Type (ADT), 2-8
what it determines, 4-1
See also subtype
data type conversion, 4-2
implicit
See implicit data type conversion, 13-10
SQL injection and, 8-22
data type family, 4-1
overloaded subprogram and, 9-30
predefined data types grouped by, E-1
subtypes with base types in same, 4-17
Data-bound collation, 14-75
database character set, 3-1
database links
DR units, 9-53
DATABASE trigger, 10-35
datatype
in CREATE TYPE statement, 15-66
Datatypes, 14-74
DATE data type family, E-1
DBMS_ASSERT package, 8-25
DBMS_DB_VERSION package, 3-61
DBMS_DDL package, A-8
DBMS_PARALLEL_EXECUTE package, 13-67
DBMS_PREPROCESSOR package, 3-63
DBMS_SQL package, 8-11
switching to native dynamic SQL from, 8-11
DBMS_SQL.GET_NEXT_RESULT procedure, 8-13
DBMS_SQL.RETURN_RESULT procedure, 8-12
DBMS_SQL.TO_NUMBER function, 8-16
DBMS_SQL.TO_REFCURSOR function, 8-15
DBMS_STANDARD package, 3-20
DBMS_WARNING package, 12-4
dbmsupgin.sql script, 13-72
dbmsupgnv.sql script, 13-72
DDL statement, 7-60
dynamic SQL for, 8-1
in trigger, 7-60
subprogram side effects and, 9-49
deadlock
autonomous transaction and, 7-59
implicit rollback and, 7-50
declaration, 3-15
exception raised in, 12-23
DEFAULT COLLATION clause, 14-75
in CREATE FUNCTION statement, 15-25
in CREATE PACKAGE statement, 15-35
in CREATE PROCEDURE statement, 15-42
in CREATE TRIGGER statement, 15-45
in CREATE TYPE statement, 15-66
default value, 9-23
of cursor parameter, 7-18
default value (continued)
of subprogram parameter, 9-23
See also initial value
DEFINE
binding category, 13-12
definer’s rights clause, 14-140
definer’s rights unit
See DR unit
DELETE collection method, 6-33
COUNT method and, 6-44
EXISTS method and, 6-39
EXTEND method and, 6-37
FIRST method and, 6-39
LAST method and, 6-39
NEXT method and, 6-46
PRIOR method and, 6-46
TRIM method and, 6-36
DELETE statement, 14-78
BEFORE statement trigger and, 10-39
PL/SQL extension to, 14-78
See also DML statement
DELETING conditional predicate, 10-5
delimiter, 3-5
dense collection, 6-2
DEPRECATE pragma, 14-78
Descriptive Intermediate Attributed Notation for Ada (DIANA), C-1
DETERMINISTIC
in CREATE FUNCTION statement, 15-25
DETERMINISTIC clause, 14-88
DETERMINISTIC option, 14-130
for pipelined table function, 13-43
DIAGNOSTIC_DEST initialization parameter, 12-1
direct-key partitioning, 15-25, 15-77
directive, 3-55
compiler
See pragma, 3-12
error, 3-57
inquiry, 3-57
selection, 3-56
See also conditional compilation directive
DML statement, 14-78, 14-142, 14-191
avoiding inner capture in, B-8
in FORALL statement, 13-13
inside pipelined table function, 13-43
on pipelined table function result, 13-51
PL/SQL syntax of, 7-1
repeating efficiently, 13-13
DML trigger, 10-4
dot notation, 3-20
for collection method, 6-32
for identifier in named PL/SQL unit, 3-20
for pseudocolumn, 7-3
for record field, 6-1
dot notation (continued)
name resolution and, 8-1
double quotation mark ("), 3-1
DR unit, 9-50
call stack and, 9-50
database links, 9-53
dynamic SQL and, 9-50
INHERIT REMOTE PRIVILEGES privilege, 9-53
name resolution and, 9-50
privilege checking and, 9-50
SCHEMA trigger and, 10-34
static SQL and, 9-50
See also AUTHID property
DROP FUNCTION statement, 15-81
DROP LIBRARY statement, 15-83
DROP PACKAGE statement, 15-84
DROP PROCEDURE statement, 15-85
DROP TRIGGER statement, 15-86
DROP TYPE BODY statement, 15-90
DUP_VAL_ON_INDEX exception, 12-11
dynamic SQL, 8-1
AUTHID property and, 9-50
native, 8-2
switching to DBMS_SQL package from, 8-11
placeholder for bind variable in EXECUTE IMMEDIATE statement and, 8-2
repeated, 8-10
tuning, 13-4

E
editioned, 15-12
editioned Abstract Data Type (ADT), 15-12
element of collection, 6-1
element specification
in CREATE TYPE statement, 15-66
syntax diagram, 14-90
embedded SQL
See static SQL
empty collection, 6-2
creating with constructor, 6-16
error directive, 3-57
error handling, 12-1
error-reporting function
SQLCODE, 14-180
SQLERRM, 14-181
SQL%BULK_EXCEPTIONS and, 13-21
escape character, 3-40
escaped identifier, 8-6
evaluation order, 3-29
events publication, 10-49
evolution of type, 15-12
exception, 12-1
handling, 12-5
in FORALL statement, 13-19
in trigger, 10-37
See also exception handler, 12-5
internally defined
See internally defined exception, 12-10
predefined
See predefined exception, 12-11
raised in cursor FOR LOOP statement, 7-26
raised in declaration, 12-23
raised in exception handler, 12-24
raising explicitly, 12-16
re-raising, 12-17
unhandled, 12-27
in FORALL statement, 13-19
user-defined
See user-defined exception, 12-14
exception handler, 12-5
continuing execution after, 12-29
exception raised in, 12-24
for NO_DATA_NEEDED, 13-51
GOTO statement and, 14-132
locator variables for, 12-7
retrieving error code and message in, 12-27
retrying transaction after, 12-30
syntax diagram, 14-98
EXCEPTION_INIT pragma, 14-95
for giving error code to user-defined exception, 12-18
for giving name to internally defined exception, 12-10
EXECUTE IMMEDIATE statement, 8-2
syntax diagram, 14-100
EXISTS collection method, 6-39
EXIT statement
syntax diagram, 14-103
EXIT WHEN statement
syntax diagram, 14-103
exiting a loop, 5-9
explicit cursor, 7-9
declaration syntax diagram, 14-105
in package
declaring, 11-12
opening and closing, 11-11
query result processing with
in FOR LOOP statement, 7-26
with OPEN, FETCH, and CLOSE statements, 7-29
explicit format model, 8-27
expression, 3-28
CURSOR, 7-43
passing to pipelined table function, 13-48
data type of, 3-28
in explicit cursor, 7-15
expression (continued)
  SQL function in PL/SQL, 3-47
  static, 3-48
  syntax diagram, 14-109
EXTEND collection method, 6-37
external subprogram, 9-54
  call specification, 14-44

F

FETCH FIRST clause, 13-33
FETCH statement
  across COMMIT, 7-53
  record variable and, 6-65
  syntax diagram, 14-119
  that returns no row, 7-11
  with BULK COLLECT clause, 13-34
  with cursor variable, 7-33
  with explicit cursor, 7-11
field of record, 6-1
FIRST collection method, 6-39
FOR LOOP index, 5-11
FOR LOOP iterand, 5-11
FOR LOOP statement, 5-10
  FOR LOOP statement
    bounds of, 14-121
    FORALL statement and, 13-13
  STEP clause and, 5-10
  syntax diagram, 14-121
    See also cursor FOR LOOP statement
FOR UPDATE cursor, 7-52
  after COMMIT or ROLLBACK, 7-53
FORALL statement, 13-13
  associative array in, 14-124
  bulk binding and, 13-12
  BULK COLLECT clause and, 13-39
  for sparse collection, 13-16
    SQL%BULK_EXCEPTIONS and, 13-23
  handling exception raised in
    after FORALL completes, 13-21
    immediately, 13-19
  number of rows affected by, 13-24
  syntax diagram, 14-124
  unhandled exception in, 13-19
format model, 8-27
forward declaration of subprogram, 9-9
function, 9-1
  aggregate
    See aggregate function, 13-48
  built-in
    See SQL function, 13-11
  declaration syntax diagram, 14-130
  error-reporting
    SQLCODE, 14-180
    SQLERRM, 13-21
  function (continued)
    invoking, 9-2
      in SQL statement, 9-49
    options for, 9-5
SQL
  See SQL function, 13-11
  structure of, 9-5
  table
    See table function, 13-42
    See also subprogram
function result cache, 9-37
function specification, 14-130

G

generated column
  See virtual column
GET_NEXT_RESULT procedure, 8-13
global identifier, 3-21
goto statement, 5-24
  restrictions on, 14-132
  syntax diagram, 14-132
granting roles to PL/SQL units, 9-52

H

hardware arithmetic, 13-9
hash table, 6-2
hiding PL/SQL source text
  See wrapping PL/SQL source text
host variable
  bulk-binding, 13-41
  cursor variable as, 7-41
  packages and, 11-2

I

identifier, 3-6, 3-20
  ambiguous reference to, B-1
  escaped, B-6
  global, 3-21
  in static SQL, 7-1
  local, 3-21
  reference to, 3-20
  scope of, 3-21
  user-defined, 3-7
    collecting data about, 13-68
    visibility of, 3-21
      See also name
IDL, C-1
IF statement, 5-1
  IF THEN ELSE form, 5-3
  IF THEN ELSIF form, 5-5
    nested IF THEN ELSE statement and,
IF statement (continued)
  IF THEN ELSIF form (continued)
  simple CASE statement and, 5-5
  IF THEN form, 5-2
  nested, 5-3
  syntax diagram, 14-135
immutable iterand, 5-14
imp and triggers, 10-47
implicit cursor, 7-6
  CURSOR expression with, 7-43
declaration syntax, 14-137
dynamic SQL and, 8-11
query result processing with
  with cursor FOR LOOP statement, 7-26
  with SELECT INTO statement, 7-25
implicit data type conversion
minimizing, 13-10
of subprogram parameter, 9-14
  causing overload error, 9-33
of subtypes
  constrained, 4-15
  unconstrained, 4-15
  with base types in same family, 4-17
implicit ROLLBACK statement, 7-50
implicitly returning query results, 8-12
Import and triggers, 10-47
IN operator, 3-42
IN OUT parameter mode, 9-15
IN parameter mode, 9-15
in-bind, 13-12
independent transaction
  See autonomous transaction
index collection, 13-16
  representing subset with, 13-16
index iterator choice, 6-17
index of collection, 6-1
index-by table
  See associative array
indices_of_control, 5-18
infinite loop, 5-10
INFORMATIONAL compile-time warning, 12-2
INHERIT ANY PRIVILEGES privilege, 9-50
INHERIT PRIVILEGES privilege, 9-50
INHERIT REMOTE PRIVILEGES privilege, 9-53
initial value, 9-23
%TYPE attribute and, 3-18
NOT NULL constraint and, 3-15
of constant, 3-17
of variable
  nested table, 6-12
  record, 6-51
  scalar, 3-17
  varray, 6-10
  See also default value
initialization parameter, 2-11
INLINE pragma, 13-2
  syntax diagram, 14-139
inner capture, 8-7
  avoiding, 8-8
input, 2-6
inquiry directive, 3-57
INSERT statement, 14-142
  inserting record with, 6-68
  restrictions on, 6-70
  PL/SQL extension to, 14-142
See also DML statement
INSERTING conditional predicate, 10-5
INSTEAD OF trigger, 10-1
  for CREATE statement, 10-36
  on DML statement, 10-6
  compound, 10-11
  for pipelined table function result, 13-51
  on nested table column, 10-6
Interface Definition Language (IDL), C-1
internally defined exception, 12-10
  giving name to, 12-10
  raising explicitly, 12-17
interpreted compilation, 13-72
INVALID_CURSOR exception, 12-11
INVALID_NUMBER exception, 12-11
invisible column, 6-60
invoker’s rights clause, 14-140
invoker’s rights unit
  See IR unit
IR unit, 9-50
  call stack and, 9-50
  dynamic SQL and, 9-50
  granting roles to, 9-52
  name resolution and, 9-50
  privilege checking and, 9-50
  static SQL and, 9-50, 9-53
  template objects for, 9-53
  See also AUTHID property
IS [NOT] NULL operator, 3-38
isolation level of transaction, 7-56
iterand, 14-3
iterand declaration, 14-3
iterand mutability property, 5-14
iteration control, 14-3
iterator, 14-3
  FOR LOOP statement
    bounds of, 14-3
    PAIRS OF, 14-3
    INDICES OF, 14-3
    syntax diagram, 14-3
    VALUES OF, 14-3
J
Java class method invocation, 9-54
Java method declaration, 14-44
JSON data type family, E-1

K
key-value pair
  See associative array
keywords, 3-6
    list of, D-1

L
LAST collection method, 6-39
LEVEL pseudocolumn, 7-3
lexical unit, 3-4
library
  creating, 15-31
  dropping, 15-83
  explicitly recompiling, 15-4
library arithmetic, 13-9
LIKE operator, 3-40
LIMIT clause, 13-37
LIMIT collection method, 6-45
line-continuation character, 3-10
literal, 3-10
local identifier, 3-21
locator variable, 12-7
lock mode, 7-52
LOCK TABLE statement, 7-52
locking
  overriding default, 7-51
  result set row, 7-52
  table, 7-52
logical operator, 3-32
logical value, 4-8
LOGIN_DENIED exception, 12-11
LONG data type, 4-7
  in trigger, 10-41
LONG RAW data type, 4-7
  in trigger, 10-41
LOOP statement
  exiting, 5-9
  kinds of, 5-9
  labeled, 5-9
  optimizing, 13-9
LOOP UNTIL structure, 5-23

M
MALFORMED_WRAP_INPUT exception, A-8
manageability, 2-3
MapReduce workloads, 15-25, 15-77
materialized view, trigger and, 15-45
member test, 3-42
Method 4, 8-11
method, collection
  See collection method
mixed parameter notation, 9-26
mode
  lock, 7-52
  subprogram parameter, 9-15
multibyte character set
  as database character set, 3-1
  variables for values from, 4-5
multidimensional collection, 6-26
multiline comment, 3-14
multiple data transformations, 13-41
multiset condition, 6-31
mutable iterand, 5-14
mutating table, 10-42
mutating-table error
  for function, 9-49
  for trigger, 10-42

N
name, 3-20
  qualified
    See dot notation, 3-20
  qualified remote, 3-20
  remote, 3-20
  simple, 3-20
  See also identifier
name resolution, B-1
  AUTHID property and, 9-50
  in static SQL, B-6
  PL/SQL and SQL differences, B-5
named choice list, 6-17
named cursor, 7-9
  pipelined table function and, 13-47
  See also explicit cursor and cursor variable
named parameter notation, 9-26
national character set, 3-3
native dynamic SQL, 8-2
  switching to DBMS_SQL package from, 8-11
native execution, compilation for, 13-70
NATURAL subtype, 4-11
NATURALN subtype, 4-11
nested comment, 3-14
nested cursor, 7-43
nested IF statement, 5-3
  IF THEN ELSIF form and, 5-5
nested record
  assignment example, 6-62
  declaration example, 6-52
nested subprogram, 9-2
  declaration and definition of, 9-2
  forward declaration for, 9-9
nested table, 6-12
  assigning null value to, 6-24

ORACLE
nested table (continued)
  assigning set operation result to, 6-25
  characteristics of, 6-2
  column in view, trigger on, 10-6
  comparing to NULL, 6-29
  comparing two, 6-29
  correlation names and, 10-28
  COUNT method for, 6-44
  FIRST and LAST methods for, 6-42
  returned by function, 13-42
  SQL multiset conditions and, 6-31
  See also collection
nested transaction, 7-55
NEW correlation name, 10-28
  with LONG or LONG RAW column, 10-41
NEXT collection method, 6-46
NEXTVAL pseudocolumn, 7-3
NLS parameters
  associative array and, 6-8
  character comparison and, 3-39
  SQL injection and, 8-22
NO_DATA_FOUND exception, 12-11
NO_DATA_NEEDED exception, 13-51
  SQLCODE for, 12-11
no-op (no operation) statement, 5-24
NOCOPY hint, 14-127
  subprogram parameter aliasing and, 9-20
  tuning subprogram invocation with, 13-7
  nonpadded comparison semantics, 4-7
IS [NOT] NULL operator
  collections and, 6-29
  NOT NULL constraint, 3-15
  %TYPE attribute and, 3-18
  EXTEND method and, 6-37
  NOT operator, 3-32
  NOT_LOGGED_ON exception, 12-11
null collection, 6-2
  assigning to collection variable, 6-24
  collection method and, 6-32
NULL statement
  syntax diagram, 14-145
  uses for, 5-24
null string, 3-10
NULL value
  assigning to record variable, 6-67
  comparing to collection
    associative array, 6-28
    nested table, 6-29
    varray, 6-29
  comparison operator and, 3-37
  concatenation operator and, 3-29
  for $SQL_UNIT inquiry directive, 3-57
  for $SQL_UNIT_OWNER inquiry directive, 3-57
  for collection variable, 6-24
NULL value (continued)
  for subprogram parameter, 9-23
  for unresolvable inquiry directive, 3-61
  in control statement, 3-32
  IN operator and, 3-42
  in set, 3-42
  in USING clause, 8-2
  simple CASE expression and, 3-44
  simple CASE statement and, 5-6
NUMBER data type family
  inefficiency of, 13-9
  members of, E-1

O

obfuscating PL/SQL source text
  See wrapping PL/SQL source text
object type
  See Abstract Data Type (ADT)
OBJECT_VALUE pseudocolumn, 10-32
OCI
  associative array and, 6-9
  cursor variable and, 7-41
  of RETURNING INTO clause, 13-38
  OLD correlation name, 10-28
OPEN FOR statement, 14-147
  recursive invocation and, 9-35
OPEN statement, 14-146
  recursive invocation and, 9-35
operation, 3-29
operator
  comparison, 3-37
    cursor variable and, 14-72
    logical, 3-32
    relational, 3-38
  collection and, 6-28
  operator precedence, 3-29
optimizer
  PL/SQL, 13-1
  SQL, 13-47
OR operator, 3-32
ORA-n error
  See internally defined exception
Oracle Call Interface (OCI)
  associative array and, 6-9
  cursor variable and, 7-41
Oracle RAC environment, result caches in, 9-47
ordinary user-defined identifier, 3-7
Original Import and triggers, 10-47
OUT parameter mode, 9-15
out-bind, 13-12
outer capture, 8-7
output, 2-6
overloaded subprogram, 9-30
  INLINE pragma and, 13-2
package, 11-1
   body of
      See package body, 11-6
   DBMS_STANDARD, 3-20
   explicitly recompiling, 15-6
   features of, 11-2
   granting roles to, 9-52
   guidelines for writing, 11-12
   initialization of, 11-7
   of static constants, 3-53
   private items in, 11-6
   product-specific, 11-1
   public items in
      See public package item, 11-3
   reasons to use, 11-2
   SERIALLY_REUSABLE, 11-8
   specification of
      See package specification, 11-3
   STANDARD
      See STANDARD package, 11-18
   state of, 11-7
   supplied by Oracle, 11-1
   wrapping, A-1
      guideline for, A-2
package body, 11-1
   creating, 15-38
   dropping, 15-84
   initialization part of, 11-6
      assigning initial values in, 11-12
   replacing, 15-38
package specification, 11-1, 11-3
   creating, 15-35
   cursor variable in, 14-72
   dropping, 15-84
   replacing, 15-35
   See also public package item
package subprogram, 9-2
   pairs_of_control, 5-18
parallel DML
   bulk binding and, 13-12
   for large table, 13-67
PARALLEL_ENABLE clause, 14-150
   in CREATE FUNCTION statement, 15-25
PARALLEL_ENABLE option, 14-130
   for pipelined table function, 13-43
   for table function, 13-42
parameter
   compilation
      See compilation parameter, 2-11
   explicit cursor, 7-16
   initialization, 2-11
   subprogram
      See subprogram parameter, 9-9
parameter mode, 9-15
   PARENT correlation name, 10-28
   with LONG or LONG RAW column, 10-41
parentheses
   nested, 3-29
   to control evaluation order, 3-29
   to improve readability, 3-29
pattern matching, 3-40
   percent sign (%) wildcard character, 3-40
PERFORMANCE compile-time warning, 12-2
PIPE ROW statement, 13-43
   in autonomous routine, 7-59
PIPELINED Function
   syntax diagram, 14-154
   PIPELINED option, 13-42
   where to specify, 13-43
   pipelined table function, 13-41, 13-42
      as aggregate function, 13-48
      in autonomous transaction, 14-153
   See also table function
PL/Scope tool, 13-68
PL/SQL architecture, 2-10
PL/SQL block
   See block
PL/SQL engine, 2-10
PL/SQL function result cache, 9-37
PL/SQL language
   advantages of, 2-1
   high performance of, 2-2
   high productivity with, 2-2
   lexical units of, 3-4
   limits of, C-1
   main features of, 2-3
   manageability and, 2-3
   portability of, 2-2
   program limits of, C-1
   scalability of, 2-3
   SQL integration in, 2-1
   syntax and semantics, 14-1
PL/SQL optimizer, 13-1
PL/SQL table
   See associative array
PL/SQL unit, 2-11
   stored
      See stored PL/SQL unit, 2-11
PL/SQL Wrapper utility, A-2
   placeholder for bind variable
      in conditional compilation directive, 3-63
      in dynamic SQL
         EXECUTE IMMEDIATE statement and,
            8-2
         repeated, 8-10
      in static SQL, 7-1
         OPEN FOR statement and, 7-33
      in trigger body, 10-28
PLS_INTEGER data type, 4-10
tuning code and, 13-9
PLS_INTEGER static expression, 3-51
PLSQL_CCFLAGS compilation parameter, 3-60
PLSQL_OPTIMIZE_LEVEL compilation parameter, 13-1
PLSQL_WARNINGS compilation parameter
displaying value of
with ALL_PLSQL_OBJECT_SETTINGS view, 12-2
with DBMS_WARNING subprogram, 12-4
setting value of
with ALTER statements, 12-2
with PLSQL_WARNINGS subprogram, 12-4
polymorphic table function, 13-53
compilation and execution, 13-57
definition, 13-53
implementation, 13-54
invocation, 13-54
    COLUMNS pseudo-operator, 13-56
optimization, 13-57
portability, 2-2
positional choice list, 6-17
positional parameter notation, 9-26
POSITIVE subtype, 4-11
POSITIVEN subtype, 4-11
post-processed source text, 3-63
pragma, 3-12
    AUTONOMOUS_TRANSACTION, 14-31
        for pipelined table function, 13-43
    COVERAGE, 14-66
    DEPRECATE, 14-78
    EXCEPTION_INIT, 14-95
    INLINE, 13-2
        syntax diagram, 14-139
    RESTRICT_REFERENCES, 14-163
    SERIALLY_REUSABLE, 14-176
    SUPPRESSES_WARNING_6009, 14-183
    UDF, 14-191
precedence, operator, 3-29
predefined constant, 4-3
predefined data type, E-1
predefined exception, 12-11
    raising explicitly, 12-17
    redeclared, 12-14
predefined inquiry directive, 3-57
predefined subtype, E-1
preprocessor control token, 3-56
PRIOR collection method, 6-46
privilege checking and AUTHID property, 9-50
procedure, 9-1
    declaration syntax, 14-157
    invoking, 9-2
    structure of, 9-5
procedure (continued)
    See also subprogram
    procedure specification, 14-157
    product-specific package, 11-1
    profiling and tracing programs, 13-68
    program limits, C-1
    PROGRAM_ERROR exception, 12-11
    pseudocolumn, 7-3
    OBJECT_VALUE, 10-32
pseudoinstruction
See pragma
pseudorecord, 10-28
    See also correlation name
public package item, 11-3
    appropriate, 11-4
    collection type as, 6-49
    cursor variable as, 14-72
    declaring, 11-3
    RECORD type as, 6-52
    referencing, 11-3
    remote variable, 11-4
    scope of, 11-3
    visibility of, 11-3
publishing events, 10-49
purity rules for subprograms, 9-49
Q
qualified expression, 6-17
    syntax diagram, 14-10
qualified name
    See dot notation
qualified remote name, 3-20
query, 7-1
    implicitly returning results of, 8-12
    invoking function in, 13-6
    processing result set of, 7-24
    multiple-row dynamic query, 8-8
        See also SELECT INTO statement
quotation mark, single or double, 3-1
quoted user-defined identifier, 3-8
R
RAISE statement, 12-16
    syntax diagram, 14-159
RAISEAPPLICATION_ERROR procedure, 12-18
raising exception explicitly, 12-16
range test, 3-42
read-only transaction, 7-50
read-write transaction, 7-50
recompiling stored PL/SQL unit, 15-1
record, 6-1
    as public package item, 6-52
record (continued)
assigning value to, 6-61
comparing one to another, 6-67
creating, 6-50
syntax diagram, 14-160
declaring constant, 6-51
nested
See nested record, 6-52
representing row, 6-55
types of, 6-52
recursive subprogram, 9-35
result-cached, 9-42
recursive trigger, 10-39
REF CURSOR
See cursor variable
REF CURSOR type, 7-31
relational operator, 3-38
collection and, 6-29
RELEASE constant, 3-61
remote exception handling
subprograms and, 12-20
triggers and, 10-37
remote name, 3-20
remote public package variable, 11-4
remote subprogram
exceptions in, 12-20
invoked by trigger, 10-36
with composite parameter, 6-1
REPEAT UNTIL structure, 5-23
replacing stored PL/SQL unit, 15-1
reraising exception, 12-17
reserved preprocessor control token, 3-56
reserved words
information about, 3-6
list of, D-1
RESTRICT_REFERENCES pragma, 14-163
result cache, 9-37
RESULT_CACHE clause, 9-38, 14-169
RESULT_CACHE option for function, 14-130, 15-25
RETURN clause of function, 9-5
RETURN INTO clause
See RETURNING INTO clause
RETURN statement, 9-6
RETURN_RESULT procedure, 8-12
RETURNING INTO clause, 14-166
BULK COLLECT clause of, 13-38
FORALL statement and, 13-39
returning query results implicitly, 8-12
REUSE SETTINGS clause, 2-11
ROLLBACK statement, 7-47
FOR UPDATE cursor and, 7-53
implicit, 7-50
in autonomous transaction, 7-59
transparent, 10-39
row-level trigger, 10-4
rowid, 4-7
ROWID data type, 4-7
ROWID pseudocolumn, 7-3
instead of CURRENT OF clause, 7-53
ROWNUM pseudocolumn
bulk SELECT operation and, 13-33
single-row result set and, 7-25
ROWTYPE_MISMATCH exception
error code for, 12-11
example of, 12-11
runtime error
See exception
S
same-scope capture, 8-7
SAMPLE clause, 13-33
SAVEPOINT statement, 7-48
in autonomous transaction, 7-59
scalability
SERIALLY_REUSABLE packages and, 11-8
subprograms and, 2-3
scalar data type, 4-1
scalar variable
assigning value to, 3-25
declaration, 3-16
syntax diagram, 14-172
initial value of, 3-17
schema object
See stored PL/SQL unit
SCHEMA trigger, 10-34
scope of identifier, 3-21
searched CASE expression, 3-46
searched CASE statement, 5-7
syntax diagram, 14-47
security mechanism
against SQL injection, 8-18
PL/SQL source text wrapping
benefit of, A-1
limitations of, A-2
trigger as, 10-3
SELECT FOR UPDATE statement, 7-52
SELECT INTO statement, 7-1
assigning values with
to record variable, 6-64
to scalar variables, 3-26
avoiding inner capture in, B-8
query result set processing with, 7-25
SQL%NOTFOUND attribute and, 7-8
SQL%ROWCOUNT attribute and, 7-8
syntax diagram, 14-174
with BULK COLLECT clause, 13-26
See also query
selection directive, 3-56
selector
  in simple CASE expression, 3-44
  in simple CASE statement, 5-6
SELF_IS_NULL exception, 12-11
sequence, 7-3
sequence iterator choice, 6-17
sequential control statement, 5-24
SERIALLY_REUSABLE package, 11-8
SERIALLY_REUSABLE pragma, 14-176
session cursor, 7-5
set data structure, 6-2
set membership test, 3-42
SET TRANSACTION statement, 7-50
SEVERE compile-time warning, 12-2
SHARING clause
  in CREATE FUNCTION statement, 15-25
  in CREATE LIBRARY statement, 15-31
  in CREATE PACKAGE statement, 15-35
  in CREATE PROCEDURE statement, 15-42
  in CREATE TRIGGER statement, 15-45
  in CREATE TYPE statement, 15-66
  sharing_clause syntax diagram, 14-177
short-circuit evaluation
  how it works, 3-37
  tuning code and, 13-11
side effects of subprogram, 9-37
SIGNTYPE subtype, 4-11
simple CASE expression, 3-44
simple CASE statement, 5-6
  IF THEN ELSIF statement and, 5-5
  syntax diagram, 14-47
simple DML trigger, 10-4
simple name, 3-20
SIMPLE_DOUBLE subtype, 4-3
  tuning code and, 13-9
SIMPLE_FLOAT subtype, 4-3
  tuning code and, 13-9
SIMPLE_INTEGER subtype, 4-13
  tuning code and, 13-9
single quotation mark ("), 3-1
single-line comment, 3-13
sort ordering, 14-75
sparse collection, 6-2
  FORALL statement for, 13-16
  SQL%BULK_EXCEPTIONS and, 13-23
  traversing, 6-46
specification
  cursor, 14-105
  function, 14-130
  package
    See package specification, 11-1
  procedure, 14-157
SQL
  bulk, 13-12
    in compound DML trigger, 10-12
SQL (continued)
  dynamic
    See dynamic SQL, 8-1
  static
    See static SQL, 7-1
SQL cursor
  See implicit cursor
SQL data type, 4-2
SQL function, 13-11
  in PL/SQL expression, 3-47
  tuning and, 13-11
SQL injection, 8-18
SQL integration in PL/SQL, 2-1
SQL multiset condition, 6-31
SQL MULTISET operator, 6-25
SQL optimizer, 13-47
SQL statement, 2-4
  for stored PL/SQL unit, 15-1
  in trigger, 10-1
  invoking collection method in, 6-32
  invoking PL/SQL function in, 9-49
  tuning, 13-5
  See also anonymous block
SQL_MACRO, 14-14
SQL_MACRO clause
  in CREATE FUNCTION statement, 15-25
SQL*Loader and triggers, 10-47
SQL%BULK_EXCEPTIONS cursor attribute, 13-21
SQL%BULK_ROWCOUNT cursor attribute, 13-24
SQL%FOUND cursor attribute, 7-7
SQL%NOTFOUND cursor attribute, 7-8
SQL%ROWCOUNT cursor attribute, 7-8
SQLCODE function, 14-180
SQLERRM function, 14-181
  SQL%BULK_EXCEPTIONS and, 13-21
standalone subprogram, 9-2
  function
    creating, 15-25
    dropping, 15-81
    explicitly recompiling, 15-2
    replacing, 15-25
  procedure
    creating, 15-42
    dropping, 15-85
    explicitly recompiling, 15-8
    replacing, 15-42
STANDARD package
  data type defined in
    See predefined data type, E-1
  exception defined in
    See predefined exception, 12-11
  how it defines PL/SQL environment, 11-18
  listing identifiers defined in, 3-7
STANDARD package (continued)
  referencing item defined in, 3-20
statement injection, 8-20
statement modification, 8-19
statement-level trigger, 10-4
static constant, 3-53
  in DBMS_DB_VERSION package, 3-61
static expression, 3-48
static SQL, 7-1
  AUTHID property and, 9-50
  name resolution in, 8-6
  PL/SQL identifier in, 7-1
  placeholder for bind variable in, 7-1
    OPEN FOR statement and, 7-33
STORAGE_ERROR exception, 12-11
  recursive invocation and, 9-35
store table, 6-16
stored PL/SQL unit, 2-11, 15-1
  creating, 15-1
  recompiling, 15-1
  replacing, 15-1
  wrapping, A-1
stored subprogram, 9-2
  unhandled exception in, 12-27
  wrapping, A-1
string, 3-10
  null, 3-10
  zero-length, 3-10
  See also character literal
STRING subtype, 4-6
strong REF CURSOR type
  creating, 7-31
  FETCH statement and, 7-33
subprogram, 9-1
  inlining, 13-2
  invoked by trigger, 10-36
  remote
    See remote subprogram, 6-1
  unhandled exception in, 12-27
subprogram invocation
  optimization of, 13-2
  resolution of, 9-28
  syntax of, 9-2
  tuning, 13-7
subprogram parameter, 9-9
  collection as, 6-32
  composite variable as, 6-1
  CURSOR expression as actual, 7-43
  cursor variable as, 7-39
  optional, 9-23
  query result as, 7-39
  required, 9-23
subprogram property, 9-3
subquery
  correlated, 7-29
subquery (continued)
  result set processing with, 7-29
SUBSCRIPT_BEYOND_COUNT exception, 12-11
SUBSCRIPT_OUTSIDE_LIMIT exception, 12-11
subtype, 4-1
  constrained, 4-15
  subprogram parameter and, 9-12
  of BINARY_DOUBLE data type, 4-3
  of BINARY_FLOAT data type, 4-3
  of PLS_INTEGER data type, 4-11
  predefined, 4-1
  unconstrained, 4-15
  user-defined, 4-14
  See also data type
SUPPRESSES_WARNING_6009 pragma, 14-183
synonym, 3-20
SYS_INVALID_ROWID exception, 12-11
SYS_REFCURSOR type, 7-31
system trigger, 10-34

T

table
  hash, 6-2
  index-by
    See associative array, 6-4
    mutating, 10-42
    nested, 6-12
    characteristics of, 6-2
    parallel DML for large, 13-67
    PL/SQL
      See associative array, 6-4
      store, 6-16
      unordered, 6-2
      updating large in parallel, 13-67
  table alias
    for avoiding inner capture, B-8
    for row expression, B-10
    for table element attribute or method, 8-9
  table function, 13-41, 13-42
    pipelined
      See pipelined table function, 13-41
      weak cursor variable argument to, 7-31
  TABLE operator, 7-38
  TCL statement, 7-1
    in subprogram invoked by trigger, 10-36
    in trigger, 7-60
template object, 9-53
TIMEOUT_ON_RESOURCE exception, 12-11
timing point
  of DML trigger
    compound, 10-11
    simple, 10-4
timning point (continued)
of system trigger, 10-34
trigger firing order and, 10-45
TO_NUMBER function, 8-16
TO_REF_CURSOR function, 8-15
TOO_MANY_ROWS exception, 12-11
trace file, 12-1
tracing and profiling programs, 13-68
transaction
autonomous, 7-55
pipelined table function in, 14-153
context of, 7-56
ending
with COMMIT statement, 7-45
with ROLLBACK statement, 7-47
isolation level of, 7-56
nested, 7-55
read-only, 7-50
read-write, 7-50
retrieving after handling exception, 12-30
SQL%ROWCOUNT cursor attribute and, 7-8
visibility of, 7-56
Transaction Control Language
See TCL statement
TRANSACTIONS initialization parameter, 7-59
tri-state logic, 3-32
trigger, 10-1
as security mechanism, 10-3
AUTHID property and, 9-50
autonomous, 7-60
cascading, 10-45
DDL statement in, 7-60
hiding implementation details of, A-2
materialized view and, 15-45
recursive, 10-39
TCL statement in, 7-60
TRIM collection method, 6-36
tuning PL/SQL code, 13-1
type
See data type
type-compatible data type
for collection variables, 6-23
for scalar variables, 3-25
typemark, 6-17

UPDATE statement (continued)
PL/SQL extensions to, 14-191
with values in record, 6-69
restrictions on, 6-70
See also DML statement
UPDATING conditional predicate, 10-5
UROWID data type, 4-7
user-defined exception, 12-14
giving error code to, 12-18
raising
with RAISE statement, 12-16
with RAISE_APPLICATION_ERROR procedure, 12-18
user-defined identifier, 3-7
collecting data about, 13-68
user-defined subtype, 4-14
collecting data about, 13-68
See Abstract Data Type (ADT)
USING_NLS_COMP, 14-75
utlrp.sql script, 13-72

V
V$RESERVED_WORDS view, D-1
validation check for avoiding SQL injection, 8-25
VALUE_ERROR exception, 12-11
values_of_control, 5-18
VARCHAR subtype, 4-6
VARCHAR2 data type, 4-4
VARCHAR2 static expression, 3-53
variable
binding of, 13-12
BOOLEAN, 3-28
collection
See collection, 6-1
cursor
See cursor variable, 7-30
host
cursor variable as, 7-41
packages and, 11-2
in cursor variable query, 7-36
in explicit cursor query, 7-13
locator, 12-7
record
See record, 6-1
remote public package, 11-4
scalar
See scalar variable, 3-16
with undefined value, 7-1
variable-size array
See varray
variadic pseudo-operator, 13-56
varray, 6-10
assigning null value to, 6-24
varray (continued)
   characteristics of, 6-2
   comparing to NULL, 6-29
   COUNT method for, 6-44
   FIRST and LAST methods for, 6-41
   returned by function, 13-42
   See also collection
VERSION constant, 3-61
view
   AUTHID property and, 9-50
   INSTEAD OF trigger and, 15-45
   materialized, trigger and, 15-45
virtual column, 7-15
   %ROWTYPE attribute and, 6-59
   explicit cursors and, 7-15
visibility
   of identifier, 3-21
   of transaction, 7-56

W
   warning, compile-time, 12-2

weak REF CURSOR type
   creating, 7-31
   FETCH statement and, 7-33
   WHILE LOOP statement, 5-23
   syntax diagram, 14-192
   white list, 11-1, 14-22
   See also ACCESSIBLE BY clause
whitespace character
   between lexical units, 3-15
   in character literal, 3-10
   in database character set, 3-1
wildcard character, 3-40
WRAP function, A-8
wrap utility
   See PL/SQL Wrapper utility
         wrapping PL/SQL source text, A-1
   inquiry directives and, 3-61

Z

ZERO_DIVIDE exception, 12-11
zero-length string, 3-10