14

Coding PL/SQL Subprograms and Packages

PL/SQL subprograms and packages are the building blocks of Oracle Database applications. Oracle recommends that you implement your application as a package, for the reasons given in *Oracle Database PL/SQL Language Reference*.

Topics:

See Also:

- *Oracle Database PL/SQL Language Reference* for information about handling errors in PL/SQL subprograms and packages
- *Oracle Database Data Cartridge Developer's Guide* for information about creating aggregate functions for complex data types such as multimedia data stored using object types, opaque types, and LOBs
- *Oracle Database SQL Tuning Guide* for information about application tracing tools, which can help you find problems in PL/SQL code

14.1 Overview of PL/SQL Subprograms

The basic unit of a PL/SQL source program is the **block**, which groups related declarations and statements. A block has an optional declarative part, a required executable part, and an optional exception-handling part. A block can be either anonymous or named.

A PL/SQL **subprogram** is a named block that can be invoked repeatedly. If the subprogram has parameters, then 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.

A subprogram is also either a **nested subprogram** (created inside a PL/SQL block, which can be another subprogram), a **package subprogram** (declared in a package specification and defined in the package body), or a **standalone subprogram** (created at schema level). Package subprograms and standalone programs are **stored subprograms**. A stored subprogram is compiled and stored in the database, where many applications can invoke it.

Stored subprograms are affected by the `AUTHID` and `ACCESSIBLE BY` clauses. The **`AUTHID` clause** affects the name resolution and privilege checking of SQL statements that the subprogram issues at runtime. The **`ACCESSIBLE BY` clause** specifies a white list of PL/SQL units that can access the subprogram.

A PL/SQL subprogram running on an Oracle Database instance can invoke an **external subprogram** written in a third-generation language (3GL). The 3GL subprogram runs in a separate address space from that of the database.

PL/SQL lets you overload nested subprograms, package subprograms, and type methods. **Overloaded subprograms** have the same name but their formal parameters differ in either name, number, order, or data type family.

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.

A `BEFORE UPDATE` trigger can fire multiple times due to internal retries. Consider this while designing your applications. If you use a normal trigger, any work done by the trigger is rolled back when there is a retry. However if you define a trigger using autonomous transactions, then any work done by the trigger is not rolled back.

See Also:

- *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL subprograms
- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL blocks
- *Oracle Database PL/SQL Language Reference* for more information about subprograms
- Overview of PL/SQL Units for information about PL/SQL units
- Developing Applications with Multiple Programming Languages for information about external subprograms
- *Oracle Database PL/SQL Language Reference* for more information about overloaded subprograms
- *Oracle Database PL/SQL Language Reference* for more information about triggers

14.2 Overview of PL/SQL Packages

A PL/SQL **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. Public items can be used or invoked by external users who have the `EXECUTE` privilege for the package or the `EXECUTE ANY PROCEDURE` privilege.

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 the accessor 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.

> **Note:**
>
> Before you create your own package, check *Oracle Database PL/SQL Packages and Types Reference* to see if Oracle supplies a package with the functionality that you need.

> **See Also:**
>
> - Overview of PL/SQL Units for information about PL/SQL units
> - *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL packages
> - *Oracle Database PL/SQL Language Reference* for the reasons to use packages

14.3 Overview of PL/SQL Units

A PL/SQL unit is one of these:

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

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.

The `AUTHID` property of a PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at runtime.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about PL/SQL units and compilation parameters
> - *Oracle Database PL/SQL Language Reference* for more information about `AUTHID` property

14.3.1 PLSQL_OPTIMIZE_LEVEL Compilation Parameter

The PL/SQL optimize level determines how much the PL/SQL optimizer can rearrange code for better performance. This level is set with the compilation parameter `PLSQL_OPTIMIZE_LEVEL` (whose default value is 2).

To change the PL/SQL optimize level for your session, use the SQL command `ALTER SESSION`. Changing the level for your session affects only subsequently created PL/SQL units. To change the level for an existing PL/SQL unit, use an `ALTER` command with the `COMPILE` clause.

To display the current value of `PLSQL_OPTIMIZE_LEVEL` for one or more PL/SQL units, use the static data dictionary view `ALL_PLSQL_OBJECT_SETTINGS`.

Example 14-1 creates two procedures, displays their optimize levels, changes the optimize level for the session, creates a third procedure, and displays the optimize levels of all three procedures. Only the third procedure has the new optimize level. Then the example changes the optimize level for only one procedure and displays the optimize levels of all three procedures again.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for information about the PL/SQL optimizer
> - *Oracle Database Reference* for more information about `PLSQL_OPTIMIZE_LEVEL`
> - *Oracle Database SQL Language Reference* for more information about `ALTER SESSION`
> - *Oracle Database PL/SQL Language Reference* for information about the `ALTER` commands for PL/SQL units
> - *Oracle Database Reference* for more information about `ALL_PLSQL_OBJECT_SETTINGS`

Example 14-1 Changing PLSQL_OPTIMIZE_LEVEL

Create two procedures:

```
CREATE OR REPLACE PROCEDURE p1 AUTHID DEFINER AS
BEGIN
  NULL;
END;
/
CREATE OR REPLACE PROCEDURE p2 AUTHID DEFINER AS
BEGIN
  NULL;
END;
/
```

Display the optimization levels of the two procedures:

```
SELECT NAME, PLSQL_OPTIMIZE_LEVEL
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME LIKE 'P%' AND TYPE='PROCEDURE'
ORDER BY NAME;
```

Result:

```
NAME                           PLSQL_OPTIMIZE_LEVEL
------------------------------ --------------------
P1                                                2
P2                                                2

2 rows selected.
```

Change the optimization level for the session and create a third procedure:

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=1;

CREATE OR REPLACE PROCEDURE p3 AUTHID DEFINER AS
BEGIN
  NULL;
END;
/
```

Display the optimization levels of the three procedures:

```
SELECT NAME, PLSQL_OPTIMIZE_LEVEL
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME LIKE 'P%' AND TYPE='PROCEDURE'
ORDER BY NAME;
```

Result:

```
NAME                           PLSQL_OPTIMIZE_LEVEL
------------------------------ --------------------
P1                                                2
P2                                                2
P3                                                1

3 rows selected.
```

Change the optimization level of procedure `p1` to 3:

```
ALTER PROCEDURE p1 COMPILE PLSQL_OPTIMIZE_LEVEL=3;
```

Display the optimization levels of the three procedures:

```
SELECT NAME, PLSQL_OPTIMIZE_LEVEL
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME LIKE 'P%' AND TYPE='PROCEDURE'
ORDER BY NAME;
```

Result:

```
NAME                           PLSQL_OPTIMIZE_LEVEL
------------------------------ --------------------
P1                                                3
P2                                                2
P3                                                1

3 rows selected.
```

14.4 Creating PL/SQL Subprograms and Packages

Topics:

- Returning Large Amounts of Data from a Function
- PL/SQL Function Result Cache
- Overview of Bulk Binding
- PL/SQL Dynamic SQL

14.4.1 Privileges Needed to Create Subprograms and Packages

To create a standalone subprogram or package in your own schema, you must have the `CREATE PROCEDURE` system privilege. To create a standalone subprogram or package in another schema, you must have the `CREATE ANY PROCEDURE` system privilege.

If the subprogram or package that you create references schema objects, then you must have the necessary object privileges for those objects. These privileges must be granted to you explicitly, not through roles.

If the privileges of the owner of a subprogram or package change, then the subprogram or package must be reauthenticated before it is run. If a necessary object privilege for a referenced object is revoked from the owner of the subprogram or package, then the subprogram cannot run.

Granting the `EXECUTE` privilege on a subprogram lets users run that subprogram under the security domain of the subprogram owner, so that the user need not be granted privileges to the objects that the subprogram references. The `EXECUTE` privilege allows more disciplined and efficient security strategies for database applications and their users. Furthermore, it allows subprograms and packages to be stored in the data dictionary (in the `SYSTEM` tablespace), where no quota controls the amount of space available to a user who creates subprograms and packages.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about system and object privileges
> - Invoking Stored PL/SQL Subprograms

14.4.2 Creating Subprograms and Packages

This topic explains how to create standalone subprograms and packages, using SQL Data Definition Language (DDL) statements.

The DDL statements for creating standalone subprograms and packages are:

- `CREATE FUNCTION`
- `CREATE PROCEDURE`
- `CREATE PACKAGE`
- `CREATE PACKAGE BODY`

  The name of a package and the names of its public objects must be unique within the package schema. The package specification and body must have the same name. Package constructs must have unique names within the scope of the package, except for overloaded subprograms.

Each of the preceding `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 regranting 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.

Using any text editor, create a text file that contains DDL statements for creating any number of subprograms and packages.

To run the DDL statements, use an interactive tool such as SQL*Plus. The SQL*Plus command `START` or @ runs a script. For example, this SQL*Plus command runs the script `my_app.sql`:

```
@my_app
```

Alternatively, you can create and run the DDL statements using SQL Developer.

> **See Also:**
>
> - *SQL*Plus User's Guide and Reference* for information about runnings scripts in SQL*Plus
> - *Oracle SQL Developer User's Guide* for information about SQL Developer
> - *Oracle Database PL/SQL Language Reference* for more information about the following functions
>   - `CREATE FUNCTION`
>   - `CREATE PROCEDURE`
>   - `CREATE PACKAGE`
>   - `CREATE PACKAGE BODY`

14.4.3 PL/SQL Object Size Limits

The size limit for PL/SQL stored database objects such as subprograms, triggers, and packages is the size of the Descriptive Intermediate Attributed Notation for Ada (DIANA) code in the shared pool in bytes. The Linux and UNIX limit on the size of the flattened DIANA/code size is 64K but the limit might be 32K on desktop platforms.

The most closely related number that a user can access is `PARSED_SIZE`, a column in the static data dictionary view `*_OBJECT_SIZE`. The column `PARSED_SIZE` gives the size of the DIANA in bytes as stored in the `SYS.IDL_xxx$` tables. This is not the size in the shared pool. The size of the DIANA part of PL/SQL code (used during compilation) is significantly larger in the shared pool than it is in the system table.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about PL/SQL program limits and `PARSED_SIZE`
> - *Oracle Database Reference* for information about `*_OBJECT_SIZE`

14.4.4 PL/SQL Data Types

This topic introduces the PL/SQL data types and refers to other chapters or documents for more information.

Use the correct and most specific PL/SQL data type for each PL/SQL variable in your database application.

> **See Also:**
>
> Using the Correct and Most Specific Data Type

Topics:

- PL/SQL Scalar Data Types
- PL/SQL Composite Data Types
- Abstract Data Types

14.4.4.1 PL/SQL Scalar Data Types

Scalar data types store values that have 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.

Topics:

- SQL Data Types
- BOOLEAN Data Type
- PLS_INTEGER and BINARY_INTEGER Data Types
- REF CURSOR Data Type
- User-Defined PL/SQL Subtypes

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for a complete description of scalar PL/SQL data types
> - *Oracle Database PL/SQL Language Reference* for the predefined PL/SQL data types and subtypes, grouped by data type family

14.4.4.1.1 SQL Data Types

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

> **See Also:**
>
> - Using SQL Data Types in Database Applications for information about how to use the SQL data types in database applications
> - *Oracle Database PL/SQL Language Reference* for more information about SQL data types

14.4.4.1.2 BOOLEAN Data Type

The `BOOLEAN` data type stores **logical values**, which are the Boolean values `TRUE` and `FALSE` and the value `NULL`. `NULL` represents an unknown value.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `BOOLEAN` data type

14.4.4.1.3 PLS_INTEGER and BINARY_INTEGER Data Types

The PL/SQL data types `PLS_INTEGER` and `BINARY_INTEGER` are identical. For simplicity, this guide 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.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `PLS_INTEGER` data type

14.4.4.1.4 REF CURSOR Data Type

`REF CURSOR` is the data type of a cursor variable.

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.

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.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `REF CURSOR` data type and cursor variables

14.4.4.1.5 User-Defined PL/SQL Subtypes

PL/SQL lets you define your own subtypes. The base type can be any scalar PL/SQL type, including a previously defined user-defined subtype.

Subtypes can:

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

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about user-defined PL/SQL subtypes

14.4.4.2 PL/SQL Composite Data Types

Composite data types have internal components. The PL/SQL composite data types are collections and records.

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. PL/SQL has three collection types—associative array, `VARRAY` (variable-size array), and nested table.

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.

You can create a collection of records, and a record that contains collections.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about PL/SQL composite data types

14.4.4.3 Abstract Data Types

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

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about ADTs

14.4.5 Returning Result Sets to Clients

In PL/SQL, as in traditional database programming, you use cursors to process query result sets. 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 section discusses are session cursors. A **session cursor** lives in session memory until the session ends, when it ceases to exist. Session cursors are different from the cursors in the private SQL area of the program global area (PGA).

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**. The only advantage of an explicit cursor over an implicit cursor is that with an explicit cursor, you can limit the number of fetched rows.

A **cursor variable** is a pointer to a cursor. That is, its value is the address of a cursor, not the cursor itself. Therefore, a cursor variable has more flexibility than an explicit cursor. However, a cursor variable also has costs that an explicit cursor does not.

Topics:

See Also:

- *Oracle Database PL/SQL Language Reference* for general information about cursor variables, and query set result processing with implicit and explicit cursors
- *Oracle Database PL/SQL Language Reference* for more information about how to limit the number of rows and the collection size in bulk collect statements
- *Oracle Call Interface Programmer's Guide* for information about using cursor variables in OCI
- *Pro*C/C++ Programmer's Guide* for information about using cursor variables in Pro*C/C++
- *Pro*COBOL Programmer's Guide* for information about using cursor variables in Pro*COBOL
- *Oracle Database JDBC Developer's Guide* for information about using cursor variables in JDBC
- Returning Large Amounts of Data from a Function
- *Oracle Database Concepts* for more information about PGA

14.4.5.1 Advantages of 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.

The preceding characteristics give cursor variables these advantages:

- Encapsulation

  Queries are centralized in the stored subprogram that opens the cursor variable.

- Easy maintenance

  If you must change the cursor, then you must change only the stored subprogram, not every application that invokes the stored subprogram.

- Convenient security

  The application connects to the server with the user name of the application user. The application user must have `EXECUTE` permission on the stored subprogram that opens the cursor, but need not have `READ` permission on the queried tables.

14.4.5.2 Disadvantages of Cursor Variables

If you need not use a cursor variable, then use an implicit or explicit cursor, for both better performance and ease of programming.

Topics:

- Parsing Penalty for Cursor Variable
- Multiple-Row-Fetching Penalty for Cursor Variable

> **Note:**
>
> The examples in these topics include `TKPROF` reports.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for instructions for producing `TKPROF` reports

14.4.5.2.1 Parsing Penalty for Cursor Variable

When you close an explicit cursor, the cursor closes from your perspective—that is, you cannot use it where an open cursor is required—but PL/SQL caches the explicit cursor in an open state. If you reexecute the statement associated with the cursor, then PL/SQL uses the cached cursor, thereby avoiding a parse.

Avoiding a parse can significantly reduce CPU use, and the caching of explicit cursors is transparent to you; it does not affect your programming. PL/SQL does not reduce your supply of available open cursors. If your program must open another cursor but doing so would exceed the init.ora setting of `OPEN_CURSORS`, then PL/SQL closes cached cursors.

PL/SQL cannot cache a cursor variable in an open state. Therefore, a cursor variable has a parsing penalty.

In Example 14-2, the procedure opens, fetches from, and closes an explicit cursor and then does the same with a cursor variable. The anonymous block calls the procedure 10 times. The TKPROF report shows that both queries were run 10 times, but the query associated with the explicit cursor was parsed only once, while the query associated with the cursor variable was parsed 10 times.

Example 14-2 Parsing Penalty for Cursor Variable

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER IS
  CURSOR e_c IS SELECT * FROM DUAL d1;  -- explicit cursor
  c_v SYS_REFCURSOR;                    -- cursor variable
  rec DUAL%ROWTYPE;
BEGIN
  OPEN e_c;                             -- explicit cursor
  FETCH e_c INTO rec;
  CLOSE e_c;

  OPEN c_v FOR SELECT * FROM DUAL d2;   -- cursor variable
  FETCH c_v INTO rec;
  CLOSE c_v;
END;
/
BEGIN
  FOR i IN 1..10 LOOP                   -- execute p 10 times
     p;
  END LOOP;
```

TKPROF report is similar to:

```
SELECT * FROM DUAL D1;

call     count
------- ------
Parse        1
Execute     10
Fetch       10
------- ------
total       21
****************
SELECT * FROM DUAL D2;


call     count
------- ------
Parse       10
Execute     10
Fetch       10
------- ------
total       30
```

14.4.5.2.2 Multiple-Row-Fetching Penalty for Cursor Variable

Example 14-3 creates a table that has more than 7,000 rows and fetches all of those rows twice, first with an implicit cursor (fetching arrays) and then with a cursor variable (fetching individual rows). The code for the implicit cursor is simpler than the code for the cursor variable, and the TKPROF report shows that it also performs better.

Although you could use the cursor variable to fetch arrays, you would need much more code. Specifically, you would need code to do the following:

- Define the types of the collections into which you will fetch the arrays

- Explicitly bulk collect into the collections
- Loop through the collections to process the fetched data
- Close the explicitly opened cursor variable

Example 14-3 Array Fetching Penalty for Cursor Variable

Create table to query and display its number of rows:

```
CREATE TABLE t AS
  SELECT * FROM ALL_OBJECTS;

SELECT COUNT(*) FROM t;
```

Result is similar to:

```
  COUNT(*)
----------
     70788
```

Perform equivalent operations with an implicit cursor and a cursor variable:

```
DECLARE
  c_v SYS_REFCURSOR;
  rec t%ROWTYPE;
BEGIN
  FOR x IN (SELECT * FROM t exp_cur) LOOP  -- implicit cursor
    NULL;
  END LOOP;

  OPEN c_v FOR SELECT * FROM t cur_var;    -- cursor variable

  LOOP
    FETCH c_v INTO rec;
    EXIT WHEN c_v%NOTFOUND;
  END LOOP;

  CLOSE c_v;
END;
/
```

`TKPROF` report is similar to:

```
SELECT * FROM T EXP_CUR

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      722      0.23       0.23          0       1748          0       72198
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      724      0.23       0.23          0       1748          0       72198
********************************************************************************
SELECT * FROM T CUR_VAR

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch    72199      0.40       0.42          0      72203          0       72198
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    72201      0.40       0.42          0      72203          0       72198
```

14.4.5.3 Returning Query Results Implicitly

A stored subprogram can return a query result implicitly to either the client program or the subprogram's immediate caller by invoking the `DBMS_SQL.RETURN_RESULT` procedure. After `DBMS_SQL.RETURN_RESULT` returns the result, only the recipient can access it.

> **Note:**
>
> To return implicitly the result of a query executed with dynamic SQL, the subprogram must execute the query with `DBMS_SQL` procedures, not the `EXECUTE IMMEDIATE` statement. The reason is that the cursors that the `EXECUTE IMMEDIATE` statement returns to the subprogram are closed when the `EXECUTE IMMEDIATE` statement completes.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about `DBMS_SQL.RETURN_RESULT` procedure
> - *Oracle Database PL/SQL Language Reference* for information about using `DBMS_SQL` procedures for dynamic SQL

14.4.6 Returning Large Amounts of Data from a Function

In a data warehousing environment, you might use PL/SQL functions to transform large amounts of data. You might pass the data through a series of transformations, each performed by a different function. PL/SQL table functions let you perform such transformations without significant memory overhead or the need to store the data in tables between each transformation stage. These functions can accept and return multiple rows, can return rows as they are ready rather than all at once, and can be parallelized.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about performing multiple transformations with pipelined table functions

14.4.7 PL/SQL Function Result Cache

Using the PL/SQL function result cache can save significant space and time. Each time a result-cached PL/SQL 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, the result is retrieved from the cache, instead of being recomputed. Because the cache is stored in a shared global area (SGA), it is available to any session that runs your application.

If a database object that was used to compute a cached result is updated, the cached result becomes invalid and must be recomputed.

The best candidates for result-caching are functions that are invoked frequently but depend on information that changes infrequently or never.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the PL/SQL function result cache

14.4.8 Overview of Bulk Binding

Oracle Database uses two engines to run PL/SQL units. The PL/SQL engine runs the procedural statements and the SQL engine runs the SQL statements. Every SQL statement causes a context switch between the two engines. You can greatly improve the performance of your database application by minimizing the number of context switches for each PL/SQL unit.

When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required can cause poor performance. Collections include:

- Associative arrays
- Variable-size arrays
- Nested tables
- Host arrays

Binding is the assignment of values to PL/SQL variables in SQL statements. **Bulk binding** is binding an entire collection at once. Bulk binds pass the entire collection between the two engines in a single operation.

Typically, bulk binding improves performance for SQL statements that affect four or more database rows. The more rows affected by a SQL statement, the greater the performance gain from bulk binding. Consider using bulk binding to improve the performance of DML and `SELECT INTO` statements that reference collections and `FOR` loops that reference collections and return DML.

> **Note:**
>
> Parallel DML statements are disabled with bulk binding.

Topics:

- DML Statements that Reference Collections
- SELECT Statements that Reference Collections
- FOR Loops that Reference Collections and Return DML

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about bulk binding, including how to handle exceptions that occur during bulk binding operations
> - *Oracle Database PL/SQL Language Reference* for more information about parallel DML statements

14.4.8.1 DML Statements that Reference Collections

A bulk bind, which uses the `FORALL` keyword, can improve the performance of `INSERT`, `UPDATE`, or `DELETE` statements that reference collection elements.

The PL/SQL block in Example 14-4 increases the salary for employees whose manager's ID number is 7902, 7698, or 7839, with and without bulk binds. Without bulk bind, PL/SQL sends a SQL statement to the SQL engine for each updated employee, leading to context switches that slow performance.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `FORALL` statement

Example 14-4 DML Statements that Reference Collections

```
DECLARE
  TYPE numlist IS VARRAY (100) OF NUMBER;
  id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN
  -- Efficient method, using bulk bind:

  FORALL i IN id.FIRST..id.LAST
  UPDATE EMPLOYEES
  SET SALARY = 1.1 * SALARY
  WHERE MANAGER_ID = id(i);

 -- Slower method:

 FOR i IN id.FIRST..id.LAST LOOP
    UPDATE EMPLOYEES
    SET SALARY = 1.1 * SALARY
    WHERE MANAGER_ID = id(i);
 END LOOP;
END;
/
```

14.4.8.2 SELECT Statements that Reference Collections

The `BULK COLLECT` clause can improve the performance of queries that reference collections. You can use `BULK COLLECT` with tables of scalar values, or tables of `%TYPE` values.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `BULK COLLECT` clause

The PL/SQL block in Example 14-5 queries multiple values into PL/SQL tables, with and without bulk binds. Without bulk bind, PL/SQL sends a SQL statement to the SQL engine for each selected employee, leading to context switches that slow performance.

Example 14-5 SELECT Statements that Reference Collections

```
DECLARE
  TYPE var_tab IS TABLE OF VARCHAR2(20)
  INDEX BY PLS_INTEGER;

  empno    VAR_TAB;
  ename    VAR_TAB;
  counter  NUMBER;

  CURSOR c IS
    SELECT EMPLOYEE_ID, LAST_NAME
    FROM EMPLOYEES
    WHERE MANAGER_ID = 7698;
BEGIN
 -- Efficient method, using bulk bind:

 SELECT EMPLOYEE_ID, LAST_NAME BULK COLLECT
 INTO empno, ename
 FROM EMPLOYEES
 WHERE MANAGER_ID = 7698;

 -- Slower method:

 counter := 1;

 FOR rec IN c LOOP
    empno(counter) := rec.EMPLOYEE_ID;
    ename(counter) := rec.LAST_NAME;
    counter := counter + 1;
 END LOOP;
END;
/
```

14.4.8.3 FOR Loops that Reference Collections and Return DML

You can use the `FORALL` keyword with the `BULK COLLECT` keywords to improve the performance of `FOR` loops that reference collections and return DML.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about use the `BULK COLLECT` clause with the `RETURNING INTO` clause

The PL/SQL block in Example 14-6 updates the `EMPLOYEES` table by computing bonuses for a collection of employees. Then it returns the bonuses in a column called `bonus_list_inst`. The actions are performed with and without bulk binds. Without bulk bind, PL/SQL sends a SQL statement to the SQL engine for each updated employee, leading to context switches that slow performance.

Example 14-6 FOR Loops that Reference Collections and Return DML

```
DECLARE
  TYPE emp_list IS VARRAY(100) OF EMPLOYEES.EMPLOYEE_ID%TYPE;
  empids emp_list := emp_list(182, 187, 193, 200, 204, 206);

  TYPE bonus_list IS TABLE OF EMPLOYEES.SALARY%TYPE;
  bonus_list_inst  bonus_list;

BEGIN
  -- Efficient method, using bulk bind:

 FORALL i IN empids.FIRST..empids.LAST
 UPDATE EMPLOYEES
 SET SALARY = 0.1 * SALARY
 WHERE EMPLOYEE_ID = empids(i)
 RETURNING SALARY BULK COLLECT INTO bonus_list_inst;

 -- Slower method:

 FOR i IN empids.FIRST..empids.LAST LOOP
   UPDATE EMPLOYEES
   SET SALARY = 0.1 * SALARY
   WHERE EMPLOYEE_ID = empids(i)
   RETURNING SALARY INTO bonus_list_inst(i);
 END LOOP;
END;
/
```

14.4.9 PL/SQL Dynamic SQL

Dynamic SQL is a programming methodology for generating and running SQL statements at runtime. It is useful when writing general-purpose and flexible programs like dynamic 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.

If you do not need dynamic SQL, then 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.

See Also:

- Understanding Schema Object Dependency for information about schema object dependency
- *Oracle Database PL/SQL Language Reference* for more information about dynamic SQL
- *Oracle Database PL/SQL Language Reference* for more information about static SQL

14.5 Altering PL/SQL Subprograms and Packages

To alter the name of a stored standalone subprogram or package, you must drop it and then create it with the new name. For example:

```
CREATE PROCEDURE p IS BEGIN NULL; END;
/
DROP PROCEDURE p
/
CREATE PROCEDURE p1 IS BEGIN NULL; END;
/
```

To alter a stored standalone subprogram or package without changing its name, you can replace it with a new version with the same name by including `OR REPLACE` in the `CREATE` statement. For example:

```
CREATE OR REPLACE PROCEDURE p1 IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, world!');
END;
/
```

Note:

`ALTER` statements (such as `ALTER FUNCTION`, `ALTER PROCEDURE`, and `ALTER PACKAGE`) do not alter the declarations or definitions of existing PL/SQL units, they recompile only the units.

See Also:

- Dropping PL/SQL Subprograms and Packages
- *Oracle Database PL/SQL Language Reference* for information about `ALTER` statements

14.6 Deprecating Packages, Subprograms, and Types

As of Oracle Database 12*c* Release 2 (12.2), you can use the `DEPRECATE` pragma to communicate that a package, subprogram, or type has been deprecated or superseded by a new interface. When a unit is compiled that makes a reference to a deprecated element, a compilation warning is issued. To mark a PL/SQL unit as deprecated, use the `DEPRECATE` pragma.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about `DEPRECATE` pragma

14.7 Dropping PL/SQL Subprograms and Packages

To drop stored standalone subprograms, use these statements:

- `DROP FUNCTION`
- `DROP PROCEDURE`

To drop a package (specification and body) or only its body, use the `DROP PACKAGE` statement.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about `DROP FUNCTION`
> - *Oracle Database PL/SQL Language Reference* for more information about `DROP PROCEDURE`
> - *Oracle Database PL/SQL Language Reference* `DROP PACKAGE`

14.8 Compiling PL/SQL Units for Native Execution

You can usually speed up PL/SQL units—your own and those that Oracle supplies—by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace.

PL/SQL units compiled into native code run in all server environments, including the shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

Whether to compile a PL/SQL unit into native code depends on where you are in the development cycle and what the PL/SQL unit does.

> **Note:**
>
> To compile Java packages and classes for native execution, use the `ncomp` tool.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for more information about compiling PL/SQL units for native execution
> - *Oracle Database Java Developer's Guide*

14.9 Invoking Stored PL/SQL Subprograms

Stored PL/SQL subprograms can be invoked from many different environments. For example:

- Interactively, using an Oracle Database tool
- From the body of another subprogram
- From the body of a trigger
- From within an application (such as a SQL*Forms or a precompiler)

Stored PL/SQL functions (but not procedures) can also be invoked from within SQL statements.

When you invoke a subprogram owned by another user:

- You must include the name of the owner in the invocation. For example:

```
EXECUTE jdoe.Fire_emp (1043);
EXECUTE jdoe.Hire_fire.Fire_emp (1043);
```

- The `AUTHID` property of the subprogram affects the name resolution and privilege checking of SQL statements that the subprogram issues at runtime.

Topics:

See Also:

- *Oracle Database PL/SQL Language Reference* for information about `AUTHID` property, subprogram invocation, parameters, and definer's and invoker's rights
- *Oracle Database PL/SQL Language Reference* for information about coding the body of a trigger
- Invoking Stored PL/SQL Functions from SQL Statements
- *Oracle Call Interface Programmer's Guide* for information about invoking PL/SQL subprograms from OCI applications
- *Pro*C/C++ Programmer's Guide* for information about invoking PL/SQL subprograms from Pro*C/C++
- *Pro*COBOL Programmer's Guide* for information about invoking PL/SQL subprograms from Pro*COBOL
- *Oracle Database JDBC Developer's Guide* for information about invoking PL/SQL subprograms from JDBC applications

14.9.1 Privileges Required to Invoke a Stored Subprogram

You do not need privileges to invoke:

- Standalone subprograms that you own
- Subprograms in packages that you own
- Public standalone subprograms
- Subprograms in public packages

To invoke a stored subprogram owned by another user, you must have the `EXECUTE` privilege for the standalone subprogram or for the package containing the package subprogram, or you must have the `EXECUTE ANY PROCEDURE` system privilege. If the subprogram is remote, then you must be granted the `EXECUTE` privilege or `EXECUTE ANY PROCEDURE` system privilege directly, not through a role.

See Also:

Oracle Database SQL Language Reference for information about system and object privileges

14.9.2 Invoking a Subprogram Interactively from Oracle Tools

You can invoke a subprogram interactively from an Oracle Database tool, such as SQL*Plus.

> **See Also:**
>
> - *SQL*Plus User's Guide and Reference* for information about the `EXECUTE` command
> - Your tools documentation for information about performing similar operations using your development tool

Example 14-7 uses SQL*Plus to create a procedure and then invokes it in two different ways.

Some interactive tools allow you to create session variables, which you can use for the duration of the session. Using SQL*Plus, Example 14-8 creates, uses, and prints a session variable.

Example 14-7 Invoking a Subprogram Interactively with SQL*Plus

```
CREATE OR REPLACE PROCEDURE salary_raise (
  employee  EMPLOYEES.EMPLOYEE_ID%TYPE,
  increase  EMPLOYEES.SALARY%TYPE
)
IS
BEGIN
  UPDATE EMPLOYEES
  SET SALARY = SALARY + increase
  WHERE EMPLOYEE_ID = employee;
END;
/
```

Invoke procedure from within anonymous block:

```
BEGIN
  salary_raise(205, 200);
END;
/
```

Result:

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

Invoke procedure with `EXECUTE` statement:

```
EXECUTE salary_raise(205, 200);
```

Result:

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

Example 14-8 Creating and Using a Session Variable with SQL*Plus

```
-- Create function for later use:

CREATE OR REPLACE FUNCTION get_job_id (
  emp_id  EMPLOYEES.EMPLOYEE_ID%TYPE
) RETURN EMPLOYEES.JOB_ID%TYPE
IS
  job_id  EMPLOYEES.JOB_ID%TYPE;
```

```
BEGIN
  SELECT JOB_ID INTO job_id
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  RETURN job_id;
END;
/
-- Create session variable:

VARIABLE job VARCHAR2(10);

-- Run function and store returned value in session variable:

EXECUTE :job := get_job_id(204);

PL/SQL procedure successfully completed.
```

SQL*Plus command:

```
PRINT job;
```

Result:

```
JOB
--------------------------------
PR_REP
```

14.9.3 Invoking a Subprogram from Another Subprogram

A subprogram or a trigger can invoke another stored subprogram. In Example 14-9, the procedure `print_mgr_name` invokes the procedure `print_emp_name`.

Recursive subprogram invocations are allowed (that is, a subprogram can invoke itself).

Example 14-9 Invoking a Subprogram from Within Another Subprogram

```
-- Create procedure that takes employee's ID and prints employee's name:

CREATE OR REPLACE PROCEDURE print_emp_name (
  emp_id  EMPLOYEES.EMPLOYEE_ID%TYPE
)
IS
  fname  EMPLOYEES.FIRST_NAME%TYPE;
  lname  EMPLOYEES.LAST_NAME%TYPE;
BEGIN
  SELECT FIRST_NAME, LAST_NAME
  INTO fname, lname
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  DBMS_OUTPUT.PUT_LINE (
    'Employee #' || emp_id || ':  ' || fname || ' ' || lname
  );
END;
/

-- Create procedure that takes employee's ID and prints manager's name:

CREATE OR REPLACE PROCEDURE print_mgr_name (
```

```
  emp_id  EMPLOYEES.EMPLOYEE_ID%TYPE
)
IS
  mgr_id  EMPLOYEES.MANAGER_ID%TYPE;
BEGIN
  SELECT MANAGER_ID
  INTO mgr_id
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

 DBMS_OUTPUT.PUT_LINE (
   'Manager of employee #' || emp_id || ' is:  '
 );

 print_emp_name(mgr_id);
END;
/
```

Invoke procedures:

```
BEGIN
  print_emp_name(200);
  print_mgr_name(200);
END;
/
```

Result:

```
Employee #200:  Jennifer Whalen
Manager of employee #200 is:
Employee #101:  Neena Kochhar
```

14.9.4 Invoking a Remote Subprogram

A **remote subprogram** is stored on a different database from its invoker. A remote subprogram invocation must include the subprogram name, a database link to the database on which the subprogram is stored, and an actual parameter for every formal parameter (even if the formal parameter has a default value).

For example, this SQL*Plus statement invokes the stored standalone procedure `fire_emp1`, which is referenced by the local database link named `boston_server`:

```
EXECUTE fire_emp1@boston_server(1043);
```

> **Note:**
>
> Although you can invoke remote package subprograms, you cannot directly access remote package variables and constants.

> **Caution:**
>
> - Remote subprogram invocations use runtime binding. The user account to which you connect depends on the database link. (Stored subprograms use compile-time binding.)
> - If a local subprogram invokes a remote subprogram, and a time-stamp mismatch is found during execution of the local subprogram, then the remote subprogram is not run, and the local subprogram is invalidated. For more information, see Dependencies Among Local and Remote Database Procedures.

Topics:

- Synonyms for Remote Subprograms
- Transactions That Invoke Remote Subprograms

> **See Also:**
>
> - Dependencies Among Local and Remote Database Procedures
> - *Oracle Database PL/SQL Language Reference* for information about handling errors in subprograms

14.9.4.1 Synonyms for Remote Subprograms

A **synonym** is an alias for a schema object. You can create a synonym for a remote subprogram name and database link, and then use the synonym to invoke the subprogram. For example:

```
CREATE SYNONYM synonym1 for fire_emp1@boston_server;
EXECUTE synonym1(1043);
```

> **Note:**
>
> You cannot create a synonym for a package subprogram, because it is not a schema object (its package is a schema object).

Synonyms provide both data independence and location transparency. Using the synonym, a user can invoke the subprogram without knowing who owns it or where it is. However, a synonym is not a substitute for privileges—to use the synonym to invoke the subprogram, the user still needs the necessary privileges for the subprogram.

Granting a privilege on a synonym is equivalent to granting the privilege on the base object. Similarly, granting a privilege on a base object is equivalent to granting the privilege on all synonyms for the object.

You can create both private and public synonyms. A private synonym is in your schema and you control its availability to others. A public synonym belongs to the user group `PUBLIC` and is available to every database user.

Use public synonyms sparingly because they make database consolidation more difficult.

If you do not want to use a synonym, you can create a local subprogram to invoke the remote subprogram. For example:

```
CREATE OR REPLACE PROCEDURE local_procedure
  (arg IN NUMBER)
AS
BEGIN
  fire_emp1@boston_server(arg);
END;
/
DECLARE
  arg NUMBER;
BEGIN
  local_procedure(arg);
END;
/
```

> **See Also:**
>
> - *Oracle Database Concepts* for general information about synonyms
> - *Oracle Database Concepts* for examples of public synonym
> - *Oracle Database SQL Language Reference* for information about the `CREATE SYNONYM` statement
> - *Oracle Database SQL Language Reference* for information about the `GRANT` statement

14.9.4.2 Transactions That Invoke Remote Subprograms

A remote subprogram invocation is assumed to update a database. Therefore, a transaction that invokes a remote subprogram requires a two-phase commit (even if the remote subprogram does not update a database). If the transaction is rolled back, then the work done by the remote subprogram is also rolled back.

With respect to the statements `COMMIT`, `ROLLBACK`, and `SAVEPOINT`, a remote subprogram differs from a local subprogram in these ways:

- If the transaction starts on a database that is not an Oracle database, then the remote subprogram cannot run these statements.

  This situation can occur in Oracle XA applications, which are not recommended. For details, see Developing Applications with Oracle XA.

- After running one of these statements, the remote subprogram cannot start its own distributed transactions.

  A **distributed transaction** updates two or more databases. Statements in the transaction are sent to the different databases, and the transaction succeeds or fails as a unit. If the transaction fails on any database, then it must be rolled back (either to a savepoint or completely) on all databases. Consider this when creating subprograms that perform distributed updates.

- If the remote subprogram does not commit or roll back its work, then the work is implicitly committed when the database link is closed. Until then, the remote subprogram is

considered to be performing a transaction. Therefore, further invocations to the remote subprogram are not allowed.

14.10 Invoking Stored PL/SQL Functions from SQL Statements

> **Caution:**
>
> Because SQL is a declarative language, rather than an imperative (or procedural) one, you cannot know how many times a function invoked by a SQL statement will run—even if the function is written in PL/SQL, an imperative language.
>
> If your application requires that a function be executed a certain number of times, do not invoke that function from a SQL statement. Use a cursor instead.
>
> For example, if your application requires that a function be called for each selected row, then open a cursor, select rows from the cursor, and call the function for each row. This technique guarantees that the number of calls to the function is the number of rows fetched from the cursor.
>
> For general information about cursors, see *Oracle Database PL/SQL Language Reference*.

These SQL statements can invoke PL/SQL stored functions:

- `INSERT`
- `UPDATE`
- `DELETE`
- `SELECT`

  (`SELECT` can also invoke a PL/SQL function declared and defined in its `WITH` clause.
- `CALL`

  (`CALL` can also invoke a PL/SQL stored procedure.)

To invoke a PL/SQL function from a SQL statement, you must either own or have the `EXECUTE` privilege on the function. To select from a view defined with a PL/SQL function, you must have `READ` or `SELECT` privilege on the view. No separate `EXECUTE` privileges are needed to select from the view.

> **Note:**
>
> The `AUTHID` property of the PL/SQL function can also affect the privileges that you need to invoke the function from a SQL statement, because `AUTHID` affects the name resolution and privilege checking of SQL statements that the unit issues at runtime. For details, see *Oracle Database PL/SQL Language Reference*.

Topics:

- When PL/SQL Functions Can Appear in SQL Expressions
- Controlling Side Effects of PL/SQL Functions Invoked from SQL Statements

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for more information about `SELECT` statement
> - *Oracle Database PL/SQL Language Reference* for general information about invoking subprograms, including passing parameters

14.10.1 Why Invoke PL/SQL Functions from SQL Statements?

Invoking PL/SQL functions in SQL statements can:

- Increase user productivity by extending SQL

  Expressiveness of the SQL statement increases where activities are too complex, too awkward, or unavailable with SQL.
- Increase query efficiency

  Functions in the `WHERE` clause of a query can filter data using criteria that must otherwise be evaluated by the application.
- Manipulate character strings to represent special data types (for example, latitude, longitude, or temperature)
- Provide parallel query execution

  If the query is parallelized, then SQL statements in your PL/SQL subprogram might also run in parallel (using the parallel query option).

14.10.2 Where PL/SQL Functions Can Appear in SQL Statements

A PL/SQL function can appear in a SQL statement wherever a SQL function or an expression can appear in a SQL statement. For example:

- Select list of the `SELECT` statement
- Condition of the `WHERE` or `HAVING` clause
- `CONNECT BY`, `START WITH`, `ORDER BY`, or `GROUP BY` clause
- `VALUES` clause of the `INSERT` statement
- `SET` clause of the `UPDATE` statement

A PL/SQL table function (which returns a collection of rows) can appear in a `SELECT` statement instead of:

- Column name in the `SELECT` list
- Table name in the `FROM` clause

A PL/SQL function cannot appear in these contexts, which require unchanging definitions:

- `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` statement

- Default value specification for a column

14.10.3 When PL/SQL Functions Can Appear in SQL Expressions

To be invoked from a SQL expression, a PL/SQL function must satisfy these requirements:

- It must be either a user-defined aggregate function or a row function.
- Its formal parameters must be `IN` parameters, not `OUT` or `IN OUT` parameters.

The function in Example 14-10 satisfies the preceding requirements.

Example 14-10 PL/SQL Function in SQL Expression (Follows Rules)

```
DROP TABLE payroll;  -- in case it exists
CREATE TABLE payroll (
  srate  NUMBER,
  orate  NUMBER,
  acctno NUMBER
);

CREATE OR REPLACE FUNCTION gross_pay (
  emp_id  IN NUMBER,
  st_hrs  IN NUMBER := 40,
  ot_hrs  IN NUMBER := 0
) RETURN NUMBER
IS
  st_rate  NUMBER;
  ot_rate  NUMBER;
BEGIN
  SELECT srate, orate
  INTO st_rate, ot_rate
  FROM payroll
  WHERE acctno = emp_id;

 RETURN st_hrs * st_rate + ot_hrs * ot_rate;
END gross_pay;
/
```

14.10.4 Controlling Side Effects of PL/SQL Functions Invoked from SQL Statements

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.

Some side effects are not allowed in a function invoked from a SQL query or DML statement.

Before Oracle Database 8*g* Release 1 (8.1), application developers used `PRAGMA RESTRICT_REFERENCES` to assert the **purity** (freedom from side effects) of a function. This pragma remains available for backward compatibility, but do not use it in new applications. Instead, specify the optimizer hints `DETERMINISTIC` and `PARALLEL_ENABLE` when you create the function.

Topics:

- Restrictions on Functions Invoked from SQL Statements
- PL/SQL Functions Invoked from Parallelized SQL Statements
- PRAGMA RESTRICT_REFERENCES (deprecated)

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about `DETERMINISTIC` and `PARALLEL_ENABLE` optimizer hints

14.10.4.1 Restrictions on Functions Invoked from SQL Statements

> **Note:**
>
> The restrictions on functions invoked from SQL statements also apply to triggers fired by SQL statements.

If a SQL statement invokes a function, and the function runs a new SQL statement, then the execution of the new statement is logically embedded in the context of the statement that invoked the function. To ensure that the new statement is safe in this context, Oracle Database enforces these restrictions on the function:

- If the SQL statement that invokes the function is a query or DML statement, then the function cannot end the current transaction, create or rollback to a savepoint, or `ALTER` the system or session.
- If the SQL statement that invokes the function is a query or parallelized DML statement, then the function cannot run a DML statement or otherwise modify the database.
- If the SQL statement that invokes the function is a DML statement, then the function can neither read nor modify the table being modified by the SQL statement that invoked the function.

The restrictions apply regardless of how the function runs the new SQL statement. For example, they apply to new SQL statements that the function:

- Invokes from PL/SQL, whether embedded directly in the function body, run using the `EXECUTE IMMEDIATE` statement, or run using the `DBMS_SQL` package
- Runs using JDBC
- Runs with OCI using the callback context from within an external C function

To avoid these restrictions, ensure that the execution of the new SQL statement is not logically embedded in the context of the SQL statement that invokes the function. For example, put the

new SQL statement in an autonomous transaction or, in OCI, create a new connection for the external C function rather than using the handle provided by the `OCIExtProcContext` argument.

> **See Also:**
>
> Autonomous Transactions

14.10.4.2 PL/SQL Functions Invoked from Parallelized SQL Statements

When Oracle Database runs a **parallelized** SQL statement, multiple processes work simultaneously to run the single SQL statement. When a parallelized SQL statement invokes a function, each process might invoke its own copy of the function, for only the subset of rows that the process handles.

Each process has its own copy of package variables. When parallel execution begins, the package variables are initialized for each process as if a user were logging into the system; the package variable values are not copied from the original login session. Changes that one process makes to package variables do not automatically propagate to the other processes or to the original login session. Java `STATIC` class attributes are similarly initialized and modified independently in each process. A function can use package and Java `STATIC` variables to accumulate a value across the various rows that it encounters. Therefore, Oracle Database does not parallelize the execution of user-defined functions by default.

Before Oracle Database 8*g* Release 1 (8.1):

- If a parallelized query invoked a user-defined function, then the execution of the function could be parallelized if `PRAGMA RESTRICT_REFERENCES` asserted both `RNPS` and `WNPS` for the function—that is, that the function neither referenced package variables nor changed their values.

  Without this assertion, the execution of a standalone PL/SQL function (but not a C or Java function) could be parallelized if Oracle Database determined that the function neither referenced package variables nor changed their values.
- If a parallelized DML statement invoked a user-defined function, then the execution of the function could be parallelized if `PRAGMA RESTRICT_REFERENCES` asserted `RNDS`, `WNDS`, `RNPS` and `WNPS` for the function—that is, that the function neither referenced nor changed the values of either package variables or database tables.

  Without this assertion, the execution of a standalone PL/SQL function (but not a C or Java function) could be parallelized if Oracle Database determined that the function neither referenced nor changed the values of either package variables or database tables.

As of Oracle Database 8*g* Release 1 (8.1), if a parallelized SQL statement invokes a user-defined function, then the execution of a function can be parallelized in these situations:

- The function was created with `PARALLEL_ENABLE`.
- Before Oracle Database 8*g* Release 1 (8.1), the database recognized the function as parallelizable.

14.10.4.3 PRAGMA RESTRICT_REFERENCES

> **Note:**
>
> `PRAGMA RESTRICT_REFERENCES` is deprecated. In new applications, Oracle recommends using `DETERMINISTIC` and `PARALLEL_ENABLE` (explained in *Oracle Database SQL Language Reference*) instead of `RESTRICT_REFERENCES`.

In existing PL/SQL applications, you can either remove `PRAGMA RESTRICT_REFERENCES` or continue to use it, even with new functionality, to ease integration with the existing code. For example:

- When it is impossible or impractical to completely remove `PRAGMA RESTRICT_REFERENCES` from existing code.

  For example, if subprogram S1 depends on subprogram S2, and you do not remove the pragma from S1, then you might need the pragma in S2 to compile S1.
- When replacing `PRAGMA RESTRICT_REFERENCES` with `PARALLEL_ENABLE` and `DETERMINISTIC` in existing code would negatively affect the action of new, dependent code.

To used `PRAGMA RESTRICT_REFERENCES` to assert the purity of a function: In the package specification (not the package body), anywhere after the function declaration, use this syntax:

```
PRAGMA RESTRICT_REFERENCES (function_name, assertion [, assertion]... );
```

Where *`assertion`* is one of the following:

Assertion	Meaning
`RNPS`	The function reads no package state (does not reference the values of package variables)
`WNPS`	The function writes no package state (does not change the values of package variables).
`RNDS`	The function reads no database state (does not query database tables).
`WNDS`	The function writes no database state (does not modify database tables).
`TRUST`	Trust that no SQL statement in the function body violates any assertion made for the function. For more information, see Specifying the Assertion TRUST.

If you do not specify `TRUST`, and a SQL statement in the function body violates an assertion that you do specify, then the PL/SQL compiler issues an error message when it parses a violating statement.

Assert the highest purity level (the most assertions) that the function allows, so that the PL/SQL compiler never rejects the function unnecessarily.

> **Note:**
>
> If the function invokes subprograms, then either specify `PRAGMA RESTRICT_REFERENCES` for those subprograms also or specify `TRUST` in either the invoking function or the invoked subprograms.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about `PRAGMA RESTRICT_REFERENCES`

Topics:

- Specifying the Assertion TRUST
- Differences between Static and Dynamic SQL Statements

14.10.4.3.1 Example: PRAGMA RESTRICT_REFERENCES

You can use the `PRAGMA RESTRICT_REFERENCES` clause when you create a PL/SQL package.

Example 14-11 creates a function that neither reads nor writes database or package state, and asserts that is has the maximum purity level.

Example 14-11 PRAGMA RESTRICT_REFERENCES

```
DROP TABLE accounts; -- in case it exists
CREATE TABLE accounts (
  acctno   INTEGER,
  balance  NUMBER
);

INSERT INTO accounts (acctno, balance)
VALUES (12345, 1000.00);

CREATE OR REPLACE PACKAGE finance AS
  FUNCTION compound_ (
    years  IN NUMBER,
    amount IN NUMBER,
    rate   IN NUMBER
   ) RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES (compound_, WNDS, WNPS, RNDS, RNPS);
END finance;
/
CREATE PACKAGE BODY finance AS
  FUNCTION compound_ (
    years  IN NUMBER,
    amount IN NUMBER,
    rate   IN NUMBER
   ) RETURN NUMBER
   IS
   BEGIN
     RETURN amount * POWER((rate / 100) + 1, years);
   END compound_;
  -- No pragma in package body
END finance;
/
DECLARE
  interest NUMBER;
BEGIN
  SELECT finance.compound_(5, 1000, 6)
  INTO interest
  FROM accounts
  WHERE acctno = 12345;
```

```
END;
/
```

14.10.4.3.2 Specifying the Assertion TRUST

When `PRAGMA RESTRICT REFERENCES` specifies `TRUST`, the PL/SQL compiler does not check the subprogram body for violations.

`TRUST` makes it easier for a subprogram that uses `PRAGMA RESTRICT REFERENCES` to invoke subprograms that do not use it.

If your PL/SQL subprogram invokes a C or Java subprogram, then you must specify `TRUST` for either the PL/SQL subprogram (as in Example 14-12) or the C or Java subprogram (as in Example 14-13), because the PL/SQL compiler cannot check a C or Java subprogram for violations at runtime.

Example 14-12 PRAGMA RESTRICT REFERENCES with TRUST on Invoker

```
CREATE OR REPLACE PACKAGE p IS
  PROCEDURE java_sleep (milli_seconds IN NUMBER)
  AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';

  FUNCTION f (n NUMBER) RETURN NUMBER;
  PRAGMA RESTRICT_REFERENCES(f,WNDS,TRUST);
END p;
/
CREATE OR REPLACE PACKAGE BODY p IS
  FUNCTION f (
    n NUMBER
  ) RETURN NUMBER
  IS
  BEGIN
    java_sleep(n);
     RETURN n;
  END f;
END p;
/
```

Example 14-13 PRAGMA RESTRICT REFERENCES with TRUST on Invokee

```
CREATE OR REPLACE PACKAGE p IS
  PROCEDURE java_sleep (milli_seconds IN NUMBER)
  AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
  PRAGMA RESTRICT_REFERENCES(java_sleep,WNDS,TRUST);

  FUNCTION f (n NUMBER) RETURN NUMBER;
END p;
/
CREATE OR REPLACE PACKAGE BODY p IS
  FUNCTION f (
    n NUMBER
   ) RETURN NUMBER
   IS
   BEGIN
     java_sleep(n);
     RETURN n;
   END f;
END p;
/
```

14.10.4.3.3 Differences Between Static and Dynamic SQL Statements

A static `INSERT`, `UPDATE`, or `DELETE` statement does not violate `RNDS` if it does not explicitly read a database state (such as a table column). A dynamic `INSERT`, `UPDATE`, or `DELETE` statement always violate `RNDS`, regardless of whether it explicitly reads a database state.

The following `INSERT` statement violates `RNDS` if it is executed dynamically, but not if it is executed statically:

```
INSERT INTO my_table values(3, 'BOB');
```

The following `UPDATE` statement always violates `RNDS`, whether it is executed statically or dynamically, because it explicitly reads the column `name` of `my_table`:

```
UPDATE my_table SET id=777 WHERE name='BOB';
```

14.11 Analyzing and Debugging Stored Subprograms

To compile a stored subprogram, you must fix any syntax errors in the code. To ensure that the subprogram works correctly, performs well, and recovers from errors, you might need to do additional debugging. Such debugging might involve:

- Adding extra output statements to verify execution progress and check data values at certain points within the subprogram.

  To output the value of variables and expressions, use the `PUT` and `PUT_LINE` subprograms in the Oracle package `DBMS_OUTPUT`.
- Analyzing the program and its execution in greater detail by running PL/Scope, the PL/SQL hierarchical profiler, or a debugger

Topics:

- PL/Scope
- PL/SQL Hierarchical Profiler
- Debugging PL/SQL and Java

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for information about handling errors in PL/SQL subprograms and packages
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_OUTPUT` package

14.11.1 PL/Scope

PL/Scope lets you develop powerful and effective PL/Scope source code tools that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

For more information about PL/Scope, see Using PL/Scope.

14.11.2 PL/SQL Hierarchical Profiler

The PL/SQL hierarchical profiler reports the dynamic execution profile of your PL/SQL program, organized by subprogram calls. It accounts for SQL and PL/SQL execution times separately. Each subprogram-level summary in the dynamic execution profile includes information such as number of calls to the subprogram, time spent in the subprogram itself, time spent in the subprogram subtree (that is, in its descendent subprograms), and detailed parent-children information.

You can browse the generated HTML reports in any browser. The browser's navigational capabilities, combined with well chosen links, provide a powerful way to analyze performance of large applications, improve application performance, and lower development costs.

> **See Also:**
>
> Using the PL/SQL Hierarchical Profiler for more information about PL/SQL hierarchical profiler

14.11.3 Debugging PL/SQL and Java

PL/SQL and Java code in the database can be debugged using Oracle SQL Developer, Oracle JDeveloper, and various third-party tools. The DBMS_DEBUG_JDWP package is used to establish connections between a database session and these debugger programs.

It is possible to investigate a problem occurring in a long running test, or in a production environment by connecting to the session to debug from another session. While debugging a session, it is possible to inspect the state of in-scope variables and to examine the database state as the session being debugged sees it during an uncommitted transaction. When stopped at a breakpoint, it is possible for the debugging user to issue SQL commands, and run PL/SQL code invoking stored PL/SQL subprograms in an anonymous block if necessary.

> **See Also:**
>
> - *Oracle SQL Developer User's Guide* for more information about running and debugging functions and procedures
> - *Oracle Database Java Developer's Guide* for information about using the Java Debug Wire Protocol (JDWP) PL/SQL Debugger
> - *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_DEBUG_JDWP` package
> - *Oracle Database Reference* for information about the `V$PLSQL_DEBUGGABLE_SESSIONS` view

14.11.3.1 Compiling Code for Debugging

A debugger can stop on individual code lines and access variables only in code compiled with debug information generated.

To compile a PL/SQL unit with debug information generated, set the compilation parameter `PLSQL_OPTIMIZE_LEVEL` to 1 (the default value is 2).

> **Note:**
>
> The PL/SQL compiler never generates debug information for code hidden with the PL/SQL `wrap` utility.

> **See Also:**
>
> - *Oracle Database PL/SQL Language Reference* for information about the `wrap` utility
> - Overview of PL/SQL Units for information about PL/SQL units
> - *Oracle Database Reference* for more information about `PLSQL_OPTIMIZE_LEVEL`

14.11.3.2 Privileges for Debugging PL/SQL and Java Stored Subprograms

For a session to connect to a debugger, the effective user at the time of the connect operation must have the `DEBUG CONNECT SESSION` , `DEBUG CONNECT ANY`, or appropriate `DEBUG CONNECT ON USER` privilege. The effective user might be the owner of a DR subprogram involved in making the connect call.

When a session connects to a debugger, the session login user and the enabled session-level roles are fixed as the privilege environment for that debugging connection. The privileges needed for debugging must be granted to that combination of user and roles on the relevant code. The privileges are:

- To display and change variables declared in a PL/SQL package specification or Java public variables: either `EXECUTE` or `DEBUG`.
- To display and change private variables, or to breakpoint and run code lines step by step: `DEBUG`

> **Caution:**
>
> The `DEBUG` privilege allows a debugging session to do anything that the subprogram being debugged could have done if that action had been included in its code.

Granting the `DEBUG ANY PROCEDURE` system privilege is equivalent to granting the `DEBUG` privilege on all objects in the database. Objects owned by `SYS` are not included.

> **Caution:**
>
> Granting the `DEBUG ANY PROCEDURE` privilege, or granting the `DEBUG` privilege on any object owned by `SYS`, grants complete rights to the database.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about system and object privileges
> - *Oracle Database Java Developer's Guide* for information about privileges for debugging Java subprograms
> - *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_DEBUG_JDWP package security model

14.12 Package Invalidations and Session State

Each session that references a package object has its own instance of the corresponding package, including persistent state for any public and private variables, cursors, and constants. If any of the session's instantiated packages (specification or body) are invalidated, then all package instances in the session are invalidated and recompiled. Therefore, the session state is lost for all package instances in the session.

When a package in a given session is invalidated, the session receives ORA-04068 the first time it tries to use any object of the invalid package instance. The second time a session makes such a package call, the package is reinstantiated for the session without error. However, if you handle this error in your application, be aware of the following:

- For optimal performance, Oracle Database returns this error message only when the package state is discarded. When a subprogram in one package invokes a subprogram in another package, the session state is lost for both packages.
- If a server session traps ORA-04068, then ORA-04068 is not raised for the client session. Therefore, when the client session tries to use an object in the package, the package is not reinstantiated. To reinstantiate the package, the client session must either reconnect to the database or recompile the package.

In most production environments, DDL operations that can cause invalidations are usually performed during inactive working hours; therefore, this situation might not be a problem for end-user applications. However, if package invalidations are common in your system during working hours, then you might want to code your applications to handle this error when package calls are made.

14.13 Example: Raising an ORA-04068 Error

You can use the `RAISE` clause to raise exceptions.

In Example 14-14, the `RAISE` statement raises the current exception, ORA-04068, which is the cause of the exception being handled, ORA-06508. ORA-04068 is not trapped.

Example 14-14 Raising ORA-04068

```
PROCEDURE p IS
  package_exception EXCEPTION;
  PRAGMA EXCEPTION_INIT (package_exception, -6508);
BEGIN
 ...
EXCEPTION
  WHEN package_exception THEN
    RAISE;
```

```
END;
/
```

14.14 Example: Trapping ORA-04068

You can use the `RAISE` statement in a package definition to trap errors.

In Example 14-15, the `RAISE` statement raises the exception ORA-20001 in response to ORA-06508, instead of the current exception, ORA-04068. ORA-04068 is trapped. When this happens, the ORA-04068 error is masked, which stops the package from being reinstantiated.

Example 14-15 Trapping ORA-04068

```
PROCEDURE p IS
  package_exception EXCEPTION;
  other_exception   EXCEPTION;
  PRAGMA EXCEPTION_INIT (package_exception, -6508);
  PRAGMA EXCEPTION_INIT (other_exception, -20001);
BEGIN
 ...
EXCEPTION
  WHEN package_exception THEN
    ...
    RAISE other_exception;
END;
/
```