7 Coding PL/SQL Procedures and Packages

This chapter describes some of the procedural capabilities of Oracle Database for application development, including:

- Overview of PL/SQL Program Units
- Compiling PL/SQL Procedures for Native Execution
- Remote Dependencies
- Cursor Variables
- Handling PL/SQL Compile-Time Errors
- Handling Run-Time PL/SQL Errors
- Debugging Stored Procedures
- Calling Stored Procedures
- Calling Remote Procedures
- Calling Stored Functions from SQL Expressions
- Returning Large Amounts of Data from a Function
- Coding Your Own Aggregate Functions

Overview of PL/SQL Program Units

PL/SQL is a modern, block-structured programming language. It provides several features that make developing powerful database applications very convenient. For example, PL/SQL provides procedural constructs, such as loops and conditional statements, that are not available in standard SQL.

You can directly enter SQL data manipulation language (DML) statements inside PL/SQL blocks, and you can use procedures supplied by Oracle to perform data definition language (DDL) statements.

PL/SQL code runs on the server, so using PL/SQL lets you centralize significant parts of your database applications for increased maintainability and security. It also enables you to achieve a significant reduction of network overhead in client/server applications.

Note:
Some Oracle tools, such as Oracle Forms, contain a PL/SQL engine that lets you run PL/SQL locally.

You can even use PL/SQL for some database applications in place of 3GL programs that use embedded SQL or Oracle Call Interface (OCI).

PL/SQL program units include:

- Anonymous Blocks
- Stored Program Units (Procedures, Functions, and Packages)
- Triggers

See Also:

- *Oracle Database PL/SQL User's Guide and Reference* for syntax and examples of operations on PL/SQL packages
- *Oracle Database PL/SQL Packages and Types Reference* for information about the PL/SQL packages that come with Oracle Database

Anonymous Blocks

An anonymous block is a PL/SQL program unit that has no name. An anonymous block consists of an optional *declarative* part, an *executable* part, and one or more optional *exception handlers*.

The declarative part declares PL/SQL variables, exceptions, and cursors. The executable part contains PL/SQL code and SQL statements, and can contain nested blocks. Exception handlers contain code that is called when the exception is raised, either as a predefined PL/SQL exception (such as `NO_DATA_FOUND` or `ZERO_DIVIDE`) or as an exception that you define.

The following short example of a PL/SQL anonymous block prints the names of all employees in department 20 in the `hr.employees` table by using the `DBMS_OUTPUT` package:

```
DECLARE
   Last_name    VARCHAR2(10);
   Cursor       c1 IS SELECT last_name
                       FROM employees
                       WHERE department_id = 20;
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO Last_name;
      EXIT WHEN c1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Last_name);
   END LOOP;
END;
/
```

Note:
If you test this block using SQL*Plus, then enter the statement `SET SERVEROUTPUT ON` so that output using the `DBMS_OUTPUT` procedures (for example, `PUT_LINE`) is activated. Also, end the example with a slash (/) to activate it.

Exceptions let you handle Oracle Database error conditions within PL/SQL program logic. This allows your application to prevent the server from issuing an error that could cause the client application to end. The following anonymous block handles the predefined Oracle Database exception `NO_DATA_FOUND` (which would result in an `ORA-01403` error if not handled):

```
DECLARE
   Emp_number   INTEGER := 9999;
   Emp_name     VARCHAR2(10);
BEGIN
   SELECT Ename INTO Emp_name FROM Emp_tab
      WHERE Empno = Emp_number;   -- no such number
   DBMS_OUTPUT.PUT_LINE('Employee name is ' || Emp_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No such employee: ' || Emp_number);
END;
```

You can also define your own exceptions, declare them in the declaration part of a block, and define them in the exception part of the block. An example follows:

```
DECLARE
   Emp_name           VARCHAR2(10);
   Emp_number         INTEGER;
   Empno_out_of_range EXCEPTION;
BEGIN
   Emp_number := 10001;
   IF Emp_number > 9999 OR Emp_number < 1000 THEN
      RAISE Empno_out_of_range;
   ELSE
      SELECT Ename INTO Emp_name FROM Emp_tab
         WHERE Empno = Emp_number;
      DBMS_OUTPUT.PUT_LINE('Employee name is ' || Emp_name);
END IF;
EXCEPTION
   WHEN Empno_out_of_range THEN
      DBMS_OUTPUT.PUT_LINE('Employee number ' || Emp_number ||
       ' is out of range.');
```

```
END;
```

Anonymous blocks are usually used interactively from a tool, such as SQL*Plus, or in a precompiler, OCI, or SQL*Module application. They are usually used to call stored procedures or to open cursor variables.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for complete information about the `DBMS_OUTPUT` package
- *Oracle Database PL/SQL User's Guide and Reference* and "Handling Run-Time PL/SQL Errors"
- "Cursor Variables"

Stored Program Units (Procedures, Functions, and Packages)

A stored procedure, function, or package is a PL/SQL program unit that:

- Has a name.
- Can take parameters, and can return values.
- Is stored in the data dictionary.
- Can be called by many users.

Note:
The term **stored procedure** is sometimes used generically for both stored procedures and stored functions. The only difference between procedures and functions is that functions always return a single value to the caller, while procedures do not return a value to the caller.

Naming Procedures and Functions

Because a procedure or function is stored in the database, it must be named. This distinguishes it from other stored procedures and makes it possible for applications to call it. Each publicly-visible procedure or function in a schema must have a unique name, and the name must be a legal PL/SQL identifier.

Note:
If you plan to call a stored procedure using a stub generated by SQL*Module, then the stored procedure name must also be a legal identifier in the calling host 3GL language, such as Ada or C.

Parameters for Procedures and Functions

Stored procedures and functions can take parameters. The following example shows a stored procedure that is similar to the anonymous block in "Anonymous Blocks".

Caution:
To execute the following, use `CREATE OR REPLACE PROCEDURE`...

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER) IS
   Emp_name       VARCHAR2(10);
   CURSOR         c1 (Depno NUMBER) IS
                     SELECT Ename FROM Emp_tab
                        WHERE deptno = Depno;
BEGIN
   OPEN c1(Dept_num);
   LOOP
      FETCH c1 INTO Emp_name;
      EXIT WHEN C1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(Emp_name);
   END LOOP;
   CLOSE c1;
```

```
END;
```

In this stored procedure example, the department number is an input parameter which is used when the parameterized cursor `c1` is opened.

The formal parameters of a procedure have three major attributes, described in Table 7-1.

Table 7-1 Attributes of Procedure Parameters

Parameter Attribute	Description
Name	This must be a legal PL/SQL identifier.
Mode	This indicates whether the parameter is an input-only parameter (`IN`), an output-only parameter (`OUT`), or is both an input and an output parameter (`IN OUT`). If the mode is not specified, then `IN` is assumed.
Datatype	This is a standard PL/SQL datatype.

Parameter Modes

Parameter modes define the behavior of formal parameters. The three parameter modes, `IN` (the default), `OUT`, and `IN OUT`, can be used with any subprogram. However, avoid using the `OUT` and `IN OUT` modes with functions. The purpose of a function is to take no arguments and return a single value. It is poor programming practice to have a function return multiple values. Also, functions should be free from side effects, which change the values of variables not local to the subprogram.

Table 7-2 summarizes the information about parameter modes.

Table 7-2 Parameter Modes

IN	OUT	IN OUT
The default.	Must be specified.	Must be specified.
Passes values to a subprogram.	Returns values to the caller.	Passes initial values to a subprogram; returns updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an initialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression; must be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.

See Also:
Oracle Database PL/SQL User's Guide and Reference for details about parameter modes

Parameter Datatypes

The datatype of a formal parameter consists of one of the following:

- An *unconstrained* type name, such as `NUMBER` or `VARCHAR2`.
- A type that is *constrained* using the `%TYPE` or `%ROWTYPE` attributes.

Note:
Numerically constrained types such as `NUMBER(2)` or `VARCHAR2(20)` are not allowed in a parameter list.

%TYPE and %ROWTYPE Attributes

Use the type attributes `%TYPE` and `%ROWTYPE` to constrain the parameter. For example, the `Get_emp_names` procedure specification in "Parameters for Procedures and Functions" could be written as the following:

```
PROCEDURE Get_emp_names(Dept_num IN Emp_tab.Deptno%TYPE)
```

This has the `Dept_num` parameter take the same datatype as the `Deptno` column in the `Emp_tab` table. The column and table must be available when a declaration using `%TYPE` (or `%ROWTYPE`) is elaborated.

Using `%TYPE` is recommended, because if the type of the column in the table changes, then it is not necessary to change the application code.

If the `Get_emp_names` procedure is part of a package, then you can use previously-declared public (package) variables to constrain a parameter datatype. For example:

```
Dept_number    number(2);
...
PROCEDURE Get_emp_names(Dept_num IN Dept_number%TYPE);
```

Use the `%ROWTYPE` attribute to create a record that contains all the columns of the specified table. The following example defines the `Get_emp_rec` procedure, which returns all the columns of the `Emp_tab` table in a PL/SQL record for the given `empno`:

Caution:
To execute the following, use `CREATE OR REPLACE PROCEDURE`...

```
PROCEDURE Get_emp_rec (Emp_number  IN  Emp_tab.Empno%TYPE,
                       Emp_ret     OUT Emp_tab%ROWTYPE) IS
BEGIN
   SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
      INTO Emp_ret
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
```

You could call this procedure from a PL/SQL block as follows:

```
DECLARE
   Emp_row      Emp_tab%ROWTYPE;     -- declare a record matching a
                                     -- row in the Emp_tab table
BEGIN
   Get_emp_rec(7499, Emp_row);   -- call for Emp_tab# 7499
   DBMS_OUTPUT.PUT(Emp_row.Ename || ' '                || Emp_row.Empno);
   DBMS_OUTPUT.PUT(' '           || Emp_row.Job || ' ' || Emp_row.Mgr);
   DBMS_OUTPUT.PUT(' '           || Emp_row.Hiredate   || ' ' || Emp_row.Sal);
   DBMS_OUTPUT.PUT(' '           || Emp_row.Comm || ' '|| Emp_row.Deptno);
   DBMS_OUTPUT.NEW_LINE;
END;
```

Stored functions can also return values that are declared using `%ROWTYPE`. For example:

```
FUNCTION Get_emp_rec (Dept_num IN Emp_tab.Deptno%TYPE)
   RETURN Emp_tab%ROWTYPE IS ...
```

Tables and Records

You can pass PL/SQL tables as parameters to stored procedures and functions. You can also pass tables of records as parameters.

Note:
When passing a user defined type, such as a PL/SQL table or record to a remote procedure, to make PL/SQL use the same definition so that the type checker can verify the source, you must create a redundant loop back DBLINK so that when the PL/SQL compiles, both sources pull from the same location.

Default Parameter Values

Parameters can take default values. Use the DEFAULT keyword or the assignment operator to give a parameter a default value. For example, the specification for the Get_emp_names procedure could be written as the following:

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER DEFAULT 20) IS ...
```

or

```
PROCEDURE Get_emp_names (Dept_num IN NUMBER := 20) IS ...
```

When a parameter takes a default value, it can be omitted from the actual parameter list when you call the procedure. When you do specify the parameter value on the call, it overrides the default value.

Note:
Unlike in an anonymous PL/SQL block, you do not use the keyword DECLARE before the declarations of variables, cursors, and exceptions in a stored procedure. In fact, it is an error to use it.

Creating Stored Procedures and Functions

Use a text editor to write the procedure or function. At the beginning of the procedure, place the following statement:

```
CREATE PROCEDURE Procedure_name AS   ...
```

For example, to use the example in "%TYPE and %ROWTYPE Attributes ", create a text (source) file called get_emp.sql containing the following code:

```
CREATE PROCEDURE Get_emp_rec (Emp_number  IN  Emp_tab.Empno%TYPE,
                              Emp_ret     OUT Emp_tab%ROWTYPE) AS
BEGIN
   SELECT Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno
      INTO Emp_ret
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
/
```

Then, using an interactive tool such as SQL*Plus, load the text file containing the procedure by entering the following statement:

```
SQL> @get_emp
```

This loads the procedure into the current schema from the get_emp.sql file (.sql is the default file extension). Note the slash (/) at the end of the code. This is not part of the code; it just activates the loading of the procedure.

Use the CREATE [OR REPLACE] FUNCTION... statement to store functions.

Caution:
When developing a new procedure, it is usually much more convenient to use the CREATE OR REPLACE PROCEDURE statement. This replaces any previous version of that procedure in the same schema with the newer version, but note that this is done without warning.

You can use either the keyword IS or AS after the procedure parameter list.

See Also:
Oracle Database Reference for the complete syntax of the CREATE PROCEDURE and CREATE FUNCTION statements

Privileges to Create Procedures and Functions

To create a standalone procedure or function, or package specification or body, you must meet the following prerequisites:

- You must have the CREATE PROCEDURE system privilege to create a procedure or package in your schema, or the CREATE ANY PROCEDURE system privilege to create a procedure or package in another user's schema.

Note:
To create without errors (to compile the procedure or package successfully) requires the following additional privileges:

- The owner of the procedure or package must be explicitly granted the necessary object privileges for all objects referenced within the body of the code.
- The owner cannot obtain required privileges through roles.

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

The `EXECUTE` privilege on a procedure gives a user the right to run a procedure owned by another user. Privileged users run the procedure under the security domain of the owner of the procedure. Therefore, users never need to be granted the privileges to the objects referenced by a procedure. This allows for more disciplined and efficient security strategies with database applications and their users. Furthermore, all procedures and packages are stored in the data dictionary (in the `SYSTEM` tablespace). No quota controls the amount of space available to a user who creates procedures and packages.

Note:
Package creation requires a sort. So the user creating the package should be able to create a sort segment in the temporary tablespace with which the user is associated.

See Also:
"Privileges Required to Execute a Procedure"

Altering Stored Procedures and Functions

To alter a stored procedure or function, you must first drop it using the `DROP PROCEDURE` or `DROP FUNCTION` statement, then re-create it using the `CREATE PROCEDURE` or `CREATE FUNCTION` statement. Alternatively, use the `CREATE OR REPLACE PROCEDURE` or `CREATE OR REPLACE FUNCTION` statement, which first drops the procedure or function if it exists, then re-creates it as specified.

Caution:
The procedure or function is dropped without any warning.

Dropping Procedures and Functions

A standalone procedure, a standalone function, a package body, or an entire package can be dropped using the SQL statements `DROP PROCEDURE`, `DROP FUNCTION`, `DROP PACKAGE BODY`, and `DROP PACKAGE`, respectively. A `DROP PACKAGE` statement drops both the specification and body of a package.

The following statement drops the `Old_sal_raise` procedure in your schema:

```
DROP PROCEDURE Old_sal_raise;
```

Privileges to Drop Procedures and Functions

To drop a procedure, function, or package, the procedure or package must be in your schema, or you must have the `DROP ANY PROCEDURE` privilege. An individual procedure within a package cannot be dropped; the containing package specification and body must be re-created without the procedures to be dropped.

External Procedures

A PL/SQL procedure executing on an Oracle Database instance can call an external procedure written in a 3GL. The 3GL procedure runs in a separate address space from that of the database.

See Also:
Chapter 14, "Calling External Procedures" for information about external procedures

PL/SQL Packages

A *package* is an encapsulated collection of related program objects (for example, procedures, functions, variables, constants, cursors, and exceptions) stored together in the database.

Using packages is an alternative to creating procedures and functions as standalone schema objects. Packages have many advantages over standalone procedures and functions. For example, they:

- Let you organize your application development more efficiently.
- Let you grant privileges more efficiently.
- Let you modify package objects without recompiling dependent schema objects.
- Enable Oracle Database to read multiple package objects into memory at once.
- Can contain global variables and cursors that are available to all procedures and functions in the package.
- Let you **overload** procedures or functions. Overloading a procedure means creating multiple procedures with the same name in the same package, each taking arguments of different number or datatype.

See Also:
Oracle Database PL/SQL User's Guide and Reference for more information about subprogram name overloading

The **specification** part of a package *declares* the public types, variables, constants, and subprograms that are visible outside the immediate scope of the package. The **body** of a package *defines* the objects declared in the specification, as well as private objects that are not visible to applications outside the package.

Example of a PL/SQL Package Specification and Body

The following example shows a package specification for a package named `Employee_management`. The package contains one stored function and two stored procedures. The body for this package defines the function and the procedures:

```
CREATE PACKAGE BODY Employee_management AS
   FUNCTION Hire_emp (Name VARCHAR2, Job VARCHAR2,
      Mgr NUMBER, Hiredate DATE, Sal NUMBER, Comm NUMBER,
      Deptno NUMBER) RETURN NUMBER IS
       New_empno    NUMBER(10);

-- This function accepts all arguments for the fields in
-- the employee table except for the employee number.
-- A value for this field is supplied by a sequence.
-- The function returns the sequence number generated
-- by the call to this function.

   BEGIN
      SELECT Emp_sequence.NEXTVAL INTO New_empno FROM dual;
      INSERT INTO Emp_tab VALUES (New_empno, Name, Job, Mgr,
         Hiredate, Sal, Comm, Deptno);
      RETURN (New_empno);
   END Hire_emp;

   PROCEDURE fire_emp(emp_id IN NUMBER) AS

-- This procedure deletes the employee with an employee
-- number that corresponds to the argument Emp_id. If
-- no employee is found, then an exception is raised.

   BEGIN
      DELETE FROM Emp_tab WHERE Empno = Emp_id;
      IF SQL%NOTFOUND THEN
      Raise_application_error(-20011, 'Invalid Employee
         Number: ' || TO_CHAR(Emp_id));
   END IF;
END fire_emp;

PROCEDURE Sal_raise (Emp_id IN NUMBER, Sal_incr IN NUMBER) AS

-- This procedure accepts two arguments. Emp_id is a
-- number that corresponds to an employee number.
-- SAL_INCR is the amount by which to increase the
-- employee's salary. If employee exists, then update
-- salary with increase.
```

```
   BEGIN
      UPDATE Emp_tab
         SET Sal = Sal + Sal_incr
         WHERE Empno = Emp_id;
      IF SQL%NOTFOUND THEN
         Raise_application_error(-20011, 'Invalid Employee
            Number: ' || TO_CHAR(Emp_id));
      END IF;
   END Sal_raise;
END Employee_management;
```

Note:
If you want to try this example, then first create the sequence number `Emp_sequence`. Do this with the following SQL*Plus statement:

```
SQL> CREATE SEQUENCE Emp_sequence
   > START WITH 8000 INCREMENT BY 10;
```

PL/SQL Object Size Limitation

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

The most closely related number that a user can access is the `PARSED_SIZE` in the data dictionary view `USER_OBJECT_SIZE`. That 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.

Creating Packages

Each part of a package is created with a different statement. Create the package specification using the `CREATE PACKAGE` statement. The `CREATE PACKAGE` statement declares public package objects.

To create a package body, use the `CREATE PACKAGE BODY` statement. The `CREATE PACKAGE BODY` statement defines the procedural code of the public procedures and functions declared in the package specification.

You can also define private, or local, package procedures, functions, and variables in a package body. These objects can only be accessed by other procedures and functions in the body of the same package. They are not visible to external users, regardless of the privileges they hold.

It is often more convenient to add the `OR REPLACE` clause in the `CREATE PACKAGE` or `CREATE PACKAGE BODY` statements when you are first developing your application. The effect of this option is to drop the package or the package body without warning. The `CREATE` statements would then be the following:

```
CREATE OR REPLACE PACKAGE Package_name AS ...
```

and

```
CREATE OR REPLACE PACKAGE BODY Package_name AS ...
```

Creating Packaged Objects

The body of a package can contain:

- Procedures and functions declared in the package specification.
- Definitions of cursors declared in the package specification.
- Local procedures and functions, not declared in the package specification.
- Local variables.

Procedures, functions, cursors, and variables that are declared in the package specification are **global**. They can be called, or used, by external users that have `EXECUTE` permission for the package or that have `EXECUTE ANY PROCEDURE` privileges.

When you create the package body, make sure that each procedure that you define in the body has the same parameters, *by name, datatype, and mode*, as the declaration in the package specification. For functions in the package body, the parameters *and* the return type must agree in name and type.

Privileges to Create or Drop Packages

The privileges required to create or drop a package specification or package body are the same as those required to create or drop a standalone procedure or function.

See Also:

- "Privileges to Create Procedures and Functions"
- "Privileges to Drop Procedures and Functions"

Naming Packages and Package Objects

The names of a package and all public objects in the package must be unique within a given schema. The package specification and its body must have the same name. All package constructs must have unique names within the scope of the package, unless overloading of procedure names is desired.

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. As a result, the session state is lost for all package instances in the session.

When a package in a given session is invalidated, the session receives the following error the first time it attempts to use any object of the invalid package instance:

```
ORA-04068: existing state of packages has been discarded
```

The second time a session makes such a package call, the package is reinstantiated for the session without error.

Note:
For optimal performance, Oracle Database returns this error message only once—each time the package state is discarded.

If you handle this error in your application, ensure that your error handling strategy can accurately handle this error. For example, when a procedure in one package calls a procedure in another package, your application should be aware that the session state is lost for both packages.

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.

Packages Supplied With Oracle Database

There are many packages provided with Oracle Database, either to extend the functionality of the database or to give PL/SQL access to SQL features. You can call these packages from your application.

See Also:
Oracle Database PL/SQL Packages and Types Reference for an overview of these Oracle Database packages

---

Overview of Bulk Binds

Oracle Database uses two engines to run PL/SQL blocks and subprograms. The PL/SQL engine runs procedural statements, while the SQL engine runs SQL statements. During execution, every SQL statement causes a context switch between the two engines, resulting in performance overhead.

Performance can be improved substantially by minimizing the number of context switches required to run a particular block or subprogram. When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required by the block can cause poor performance. Collections include the following:

- Varrays

- Nested tables
- Index-by 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 back and forth between the two engines in a single operation.

Typically, using bulk binds 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 binds.

> **Note:**
> This section provides an overview of bulk binds to help you decide if you should use them in your PL/SQL applications. For detailed information about using bulk binds, including ways to handle exceptions that occur in the middle of a bulk bind operation, refer to the *Oracle Database PL/SQL User's Guide and Reference.*

When to Use Bulk Binds

If you have scenarios like these in your applications, consider using bulk binds to improve performance.

DML Statements that Reference Collections

The `FORALL` keyword can improve the performance of `INSERT`, `UPDATE`, or `DELETE` statements that reference collection elements.

For example, the following PL/SQL block increases the salary for employees whose manager's ID number is 7902, 7698, or 7839, both with and without using bulk binds:

```
DECLARE
   TYPE Numlist IS VARRAY (100) OF NUMBER;
   Id NUMLIST := NUMLIST(7902, 7698, 7839);
BEGIN

-- Efficient method, using a bulk bind
   FORALL i IN Id.FIRST..Id.LAST   -- bulk-bind the VARRAY
      UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);

-- Slower method, running the UPDATE statements within a regular loop
   FOR i IN Id.FIRST..Id.LAST LOOP
      UPDATE Emp_tab SET Sal = 1.1 * Sal
      WHERE Mgr = Id(i);
   END LOOP;
END;
```

Without the bulk bind, PL/SQL sends a SQL statement to the SQL engine for each employee that is updated, leading to context switches that hurt performance.

If you have a set of rows prepared in a PL/SQL table, you can bulk-insert or bulk-update the data using a loop like:

```
FORALL i in Emp_Data.FIRST..Emp_Data.LAST
    INSERT INTO Emp_tab VALUES(Emp_Data(i));
```

SELECT Statements that Reference Collections

The `BULK COLLECT INTO` clause can improve the performance of queries that reference collections.

For example, the following PL/SQL block queries multiple values into PL/SQL tables, both with and without bulk binds:

```
-- Find all employees whose manager's ID number is 7698.
DECLARE
   TYPE Var_tab IS TABLE OF VARCHAR2(20) INDEX BY BINARY_INTEGER;
   Empno VAR_TAB;
   Ename VAR_TAB;
   Counter NUMBER;
   CURSOR C IS
      SELECT Empno, Ename FROM Emp_tab WHERE Mgr = 7698;
BEGIN
```

```
-- Efficient method, using a bulk bind
     SELECT Empno, Ename BULK COLLECT INTO Empno, Ename
         FROM Emp_Tab WHERE Mgr = 7698;

-- Slower method, assigning each collection element within a loop.

   counter := 1;
   FOR rec IN C LOOP
      Empno(Counter) := rec.Empno;
      Ename(Counter) := rec.Ename;
      Counter := Counter + 1;
   END LOOP;
END;
```

You can use `BULK COLLECT INTO` with tables of scalar values, or tables of `%TYPE` values.

Without the bulk bind, PL/SQL sends a SQL statement to the SQL engine for each employee that is selected, leading to context switches that hurt performance.

FOR Loops that Reference Collections and the Returning Into Clause

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

For example, the following PL/SQL block updates the `Emp_tab` table by computing bonuses for a collection of employees; then it returns the bonuses in a column called `Bonlist`. The actions are performed both with and without using bulk binds:

```
DECLARE
   TYPE Emplist IS VARRAY(100) OF NUMBER;
   Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
   TYPE Bonlist IS TABLE OF Emp_tab.sal%TYPE;
   Bonlist_inst BONLIST;
BEGIN
   Bonlist_inst := BONLIST(1,2,3,4,5);

   FORALL i IN Empids.FIRST..empIDs.LAST
      UPDATE Emp_tab SET Bonus = 0.1 * Sal
      WHERE Empno = Empids(i)
      RETURNING Sal BULK COLLECT INTO Bonlist;

   FOR i IN Empids.FIRST..Empids.LAST LOOP
      UPDATE Emp_tab Set Bonus = 0.1 * sal
         WHERE Empno = Empids(i)
       RETURNING Sal INTO BONLIST(i);
   END LOOP;
END;
```

Without the bulk bind, PL/SQL sends a SQL statement to the SQL engine for each employee that is updated, leading to context switches that hurt performance.

Triggers

A trigger is a special kind of PL/SQL anonymous block. You can define triggers to fire before or after SQL statements, either on a statement level or for each row that is affected. You can also define `INSTEAD OF` triggers or system triggers (triggers on `DATABASE` and `SCHEMA`).

See Also:
Chapter 9, "Coding Triggers"

Compiling PL/SQL Procedures for Native Execution

You can speed up PL/SQL procedures by compiling them into native code residing in shared libraries. The procedures are translated into C code, then compiled with your usual C compiler and linked into the Oracle Database process.

You can use this technique with both the supplied Oracle Database PL/SQL packages, and procedures you write yourself. You can use the `ALTER SYSTEM` or `ALTER SESSION` command, or update your initialization file, to set the parameter `PLSQL_CODE_TYPE` to the value `NATIVE` (the default setting is the value `INTERPRETED`).

Because this technique cannot do much to speed up SQL statements called from these procedures, it is most effective for compute-intensive procedures that do not spend much time executing SQL.

With Java, you can use the `ncomp` tool to compile your own packages and classes.

See Also:

- *Oracle Database PL/SQL User's Guide and Reference* for details on PL/SQL native compilation
- *Oracle Database Java Developer's Guide* for details on Java native compilation

---

Remote Dependencies

Dependencies among PL/SQL program units can be handled in two ways:

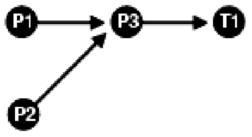
- Timestamps
- Signatures

Timestamps

If timestamps are used to handle dependencies among PL/SQL program units, then whenever you alter a program unit or a relevant schema object, all of its dependent units are marked as invalid and must be recompiled before they can be run.

Each program unit carries a timestamp that is set by the server when the unit is created or recompiled. Figure 7-1 demonstrates this graphically. Procedures `P1` and `P2` call stored procedure `P3`. Stored procedure `P3` references table `T1`. In this example, each of the procedures is dependent on table `T1`. `P3` depends upon `T1` directly, while `P1` and `P2` depend upon `T1` indirectly.

Figure 7-1 Dependency Relationships



Description of "Figure 7-1 Dependency Relationships"

If `P3` is altered, then `P1` and `P2` are marked as invalid immediately, if they are on the same server as `P3`. The compiled states of `P1` and `P2` contain records of the timestamp of `P3`. Therefore, if the procedure `P3` is altered and recompiled, then the timestamp on `P3` no longer matches the value that was recorded for `P3` during the compilation of `P1` and `P2`.

If `P1` and `P2` are on a client system, or on another Oracle Database instance in a distributed environment, then the timestamp information is used to mark them as invalid at runtime.

Disadvantages of the Timestamp Model

The disadvantage of this dependency model is that it is unnecessarily restrictive. Recompilation of dependent objects across the network are often performed when not strictly necessary, leading to performance degradation.

Furthermore, on the client side, the timestamp model can lead to situations that block an application from running at all, if the client-side application is built using PL/SQL version 2. Earlier releases of tools, such as Oracle Forms, that used PL/SQL version 1 on the client side did not use this dependency model, because PL/SQL version 1 had no support for stored procedures.

For releases of Oracle Forms that are integrated with PL/SQL version 2 on the client side, the timestamp model can present problems. For example, during the installation of the application, the application is rendered invalid unless the client-side PL/SQL procedures that it uses are recompiled at the client site. Also, if a client-side procedure depends on a server procedure, and if the server procedure is changed or automatically recompiled, then the client-side PL/SQL procedure must then be recompiled. Yet in many application environments (such as Forms runtime applications), there is no PL/SQL compiler available on the client. This blocks the application from running at all. The client application developer must then redistribute new versions of the application to all customers.

Signatures

To alleviate some of the problems with the timestamp-only dependency model, Oracle Database provides the additional capability of remote dependencies using signatures. The signature capability affects only remote dependencies. Local (same server) dependencies are not affected, as recompilation is always possible in this environment.

A signature is associated with each compiled stored program unit. It identifies the unit using the following criteria:

- The name of the unit (the package, procedure, or function name).
- The types of each of the parameters of the subprogram.
- The modes of the parameters (`IN`, `OUT`, `IN OUT`).
- The number of parameters.
- The type of the return value for a function.

The user has control over whether signatures or timestamps govern remote dependencies.

See Also:
"Controlling Remote Dependencies"

When the signature dependency model is used, a dependency on a remote program unit causes an invalidation of the dependent unit if the dependent unit contains a call to a subprogram in the parent unit, and if the signature of this subprogram has been changed in an incompatible manner.

For example, consider a procedure `get_emp_name` stored on a server in Boston (`BOSTON_SERVER`). The procedure is defined as the following:

Note:
You may need to set up data structures, similar to the following, for certain examples to work:

```
CONNECT system/manager
CREATE PUBLIC DATABASE LINK boston_server USING 'inst1_alias';
CONNECT scott/tiger
```

```
CREATE OR REPLACE PROCEDURE get_emp_name (
   emp_number   IN  NUMBER,
   hire_date    OUT VARCHAR2,
   emp_name     OUT VARCHAR2) AS
BEGIN
   SELECT ename, to_char(hiredate, 'DD-MON-YY')
      INTO emp_name, hire_date
      FROM emp
      WHERE empno = emp_number;
END;
```

When `get_emp_name` is compiled on `BOSTON_SERVER`, its signature, as well as its timestamp, is recorded.

Suppose that on another server in California, some PL/SQL code calls `get_emp_name` identifying it using a DBlink called `BOSTON_SERVER`, as follows:

```
CREATE OR REPLACE PROCEDURE print_ename (emp_number IN NUMBER) AS
   hire_date    VARCHAR2(12);
   ename        VARCHAR2(10);
BEGIN
   get_emp_name@BOSTON_SERVER(emp_number, hire_date, ename);
   dbms_output.put_line(ename);
   dbms_output.put_line(hire_date);
END;
```

When this California server code is compiled, the following actions take place:

- A connection is made to the Boston server.
- The signature of `get_emp_name` is transferred to the California server.
- The signature is recorded in the compiled state of `print_ename`.

At runtime, during the remote procedure call from the California server to the Boston server, the recorded signature of `get_emp_name` that was saved in the compiled state of `print_ename` gets sent to the Boston server, regardless of whether or not there were any changes.

If the timestamp dependency mode is in effect, then a mismatch in timestamps causes an error status to be returned to the calling procedure.

However, if the signature mode is in effect, then any mismatch in timestamps is ignored, and the recorded signature of `get_emp_name` in the compiled state of `Print_ename` on the California server is compared with the current signature of `get_emp_name` on the Boston server. If they match, then the call succeeds. If they do not match, then an error status is returned to the `print_name` procedure.

Note that the `get_emp_name` procedure on the Boston server could have been changed. Or, its timestamp could be different from that recorded in the `print_name` procedure on the California server, possibly due to the installation of a new release of the server. As long as the signature remote dependency mode is in effect on the California server, a timestamp mismatch does not cause an error when `get_emp_name` is called.

Note:
`DETERMINISTIC`, `PARALLEL_ENABLE`, and purity information do not show in the signature mode. Optimizations based on these settings are not automatically reconsidered if a function on a remote system is redefined with different settings. This may lead to incorrect query results when calls to the remote function occur, even indirectly, in a SQL statement, or if the remote function is used, even indirectly, in a function-based index.

When Does a Signature Change?

Here is information on when a signature changes.

Switching Datatype Classes

A signature changes when you switch from one class of datatype to another. Within each datatype class, there can be several types. Changing a parameter datatype from one type to another within a class does not cause the signature to change. Datatypes that are not listed in the following table, such as `NCHAR` or `TIMESTAMP`, are not part of any class; changing their type always causes a signature mismatch.

VARCHAR types: `VARCHAR2, VARCHAR, STRING, LONG, ROWID`

Character types: `CHARACTER, CHAR`

Raw types: `RAW, LONG RAW`

Integer types: `BINARY_INTEGER, PLS_INTEGER, BOOLEAN, NATURAL, POSITIVE, POSITIVEN, NATURALN`

Number types: `NUMBER, INTEGER, INT, SMALLINT, DECIMAL, DEC, REAL, FLOAT, NUMERIC, DOUBLE PRECISION, DOUBLE PRECISION, NUMERIC`

Date types: `DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND`

Modes

Changing to or from an explicit specification of the default parameter mode `IN` does not change the signature of a subprogram. For example, changing between:

```
PROCEDURE P1 (Param1 NUMBER);
PROCEDURE P1 (Param1 IN NUMBER);
```

does not change the signature. Any other change of parameter mode *does* change the signature.

Default Parameter Values

Changing the specification of a default parameter value does not change the signature. For example, procedure `P1` has the same signature in the following two examples:

```
PROCEDURE P1 (Param1 IN NUMBER := 100);
PROCEDURE P1 (Param1 IN NUMBER := 200);
```

An application developer who requires that callers get the new default value must recompile the called procedure, but no signature-based invalidation occurs when a default parameter value assignment is changed.

Examples of Changing Procedure Signatures

Using the `Get_emp_names` procedure defined in "Parameters for Procedures and Functions", if the procedure body is changed to the following:

```
DECLARE
   Emp_number  NUMBER;
   Hire_date   DATE;
BEGIN
-- date format model changes

   SELECT Ename, To_char(Hiredate, 'DD/MON/YYYY')
      INTO Emp_name, Hire_date
      FROM Emp_tab
      WHERE Empno = Emp_number;
END;
```

The specification of the procedure has not changed, so its signature has not changed.

But if the procedure specification is changed to the following:

```
CREATE OR REPLACE PROCEDURE Get_emp_name (
   Emp_number  IN  NUMBER,
   Hire_date   OUT DATE,
   Emp_name    OUT VARCHAR2) AS
```

And if the body is changed accordingly, then the signature changes, because the parameter `Hire_date` has a different datatype.

However, if the name of that parameter changes to `When_hired`, and the datatype remains `VARCHAR2`, and the mode remains `OUT`, the signature does *not* change. Changing the *name* of a formal parameter does not change the signature of the unit.

Consider the following example:

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_type IS RECORD (
        Emp_number NUMBER,
        Hire_date  VARCHAR2(12),
        Emp_name   VARCHAR2(10));
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type);
END;

CREATE OR REPLACE PACKAGE BODY Emp_package AS
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type) IS
   BEGIN
       SELECT Empno, Ename, TO_CHAR(Hiredate, 'DD/MON/YY')
           INTO Emp_data
           FROM Emp_tab
           WHERE Empno = Emp_data.Emp_number;
   END;
END;
```

If the package specification is changed so that the record's field names are changed, but the types remain the same, then this does not affect the signature. For example, the following package specification has the same signature as the previous package specification example:

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_type IS RECORD (
        Emp_num    NUMBER,         -- was Emp_number
        Hire_dat   VARCHAR2(12),   -- was Hire_date
        Empname    VARCHAR2(10));  -- was Emp_name
    PROCEDURE Get_emp_data
        (Emp_data IN OUT Emp_data_type);
END;
```

Changing the name of the type of a parameter does not cause a change in the signature if the type remains the same as before. For example, the following package specification for `Emp_package` is the same as the first one:

```
CREATE OR REPLACE PACKAGE Emp_package AS
    TYPE Emp_data_record_type IS RECORD (
        Emp_number NUMBER,
        Hire_date  VARCHAR2(12),
        Emp_name   VARCHAR2(10));
    PROCEDURE Get_emp_data
```

```
        (Emp_data IN OUT Emp_data_record_type);
END;
```

Controlling Remote Dependencies

The dynamic initialization parameter `REMOTE_DEPENDENCIES_MODE` controls whether the timestamp or the signature dependency model is in effect.

- If the initialization parameter file contains the following specification:

  ```
  REMOTE_DEPENDENCIES_MODE = TIMESTAMP
  ```

Then only timestamps are used to resolve dependencies (if this is not explicitly overridden dynamically).

- If the initialization parameter file contains the following parameter specification:

  ```
  REMOTE_DEPENDENCIES_MODE = SIGNATURE
  ```

Then signatures are used to resolve dependencies (if this not explicitly overridden dynamically).

- You can alter the mode dynamically by using the DDL statements. For example, this example alters the dependency model for the current session:

  ```
  ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}

  Thise example alters the dependency model systemwide after startup:
  ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =
      {SIGNATURE | TIMESTAMP}
  ```

If the `REMOTE_DEPENDENCIES_MODE` parameter is not specified, either in the `init.ora` parameter file or using the `ALTER SESSION` or `ALTER SYSTEM` DDL statements, then timestamp is the default value. Therefore, unless you explicitly use the `REMOTE_DEPENDENCIES_MODE` parameter, or the appropriate DDL statement, your server is operating using the timestamp dependency model.

When you use `REMOTE_DEPENDENCIES_MODE=SIGNATURE`:

- If you change the default value of a parameter of a remote procedure, then the local procedure calling the remote procedure is not invalidated. If the call to the remote procedure does not supply the parameter, then the default value is used. In this case, because invalidation/recompilation does not automatically occur, the old default value is used. If you want to see the new default values, then you must recompile the calling procedure manually.
- If you add a new overloaded procedure in a package (a new procedure with the same name as an existing one), then local procedures that call the remote procedure are not invalidated. If it turns out that this overloading results in a rebinding of existing calls from the local procedure under the timestamp mode, then this rebinding does not happen under the signature mode, because the local procedure does not get invalidated. You must recompile the local procedure manually to achieve the new rebinding.
- If the types of parameters of an existing packaged procedure are changed so that the new types have the same shape as the old ones, then the local calling procedure is not invalidated or recompiled automatically. You must recompile the calling procedure manually to get the semantics of the new type.

Dependency Resolution

When `REMOTE_DEPENDENCIES_MODE` = `TIMESTAMP` (the default value), dependencies among program units are handled by comparing timestamps at runtime. If the timestamp of a called remote procedure does not match the timestamp of the called procedure, then the calling (dependent) unit is invalidated and must be recompiled. In this case, if there is no local PL/SQL compiler, then the calling application cannot proceed.

In the timestamp dependency mode, signatures are not compared. If there is a local PL/SQL compiler, then recompilation happens automatically when the calling procedure is run.

When `REMOTE_DEPENDENCIES_MODE` = `SIGNATURE`, the recorded timestamp in the calling unit is first compared to the current timestamp in the called remote unit. If they match, then the call proceeds. If the timestamps do not match, then the signature of the called remote subprogram, as recorded in the calling subprogram, is compared with the current signature of the called subprogram. If they do not match (using the criteria described in the section "When Does a Signature Change?"), then an error is returned to the calling session.

Suggestions for Managing Dependencies

Follow these guidelines for setting the `REMOTE_DEPENDENCIES_MODE` parameter:

- Server-side PL/SQL users can set the parameter to `TIMESTAMP` (or let it default to that) to get the timestamp dependency mode.

- Server-side PL/SQL users can choose to use the signature dependency mode if they have a distributed system and they want to avoid possible unnecessary recompilations.
- Client-side PL/SQL users should set the parameter to `SIGNATURE`. This allows:
    - Installation of new applications at client sites, without the need to recompile procedures.
    - Ability to upgrade the server, without encountering timestamp mismatches.
- When using signature mode on the server side, add new procedures to the end of the procedure (or function) declarations in a package specification. Adding a new procedure in the middle of the list of declarations can cause unnecessary invalidation and recompilation of dependent procedures.

Cursor Variables

A cursor is a static object; a cursor variable is a pointer to a cursor. Because cursor variables are pointers, they can be passed and returned as parameters to procedures and functions. A cursor variable can also refer to different cursors in its lifetime.

Some additional advantages of cursor variables include:

- *Encapsulation* Queries are centralized in the stored procedure that opens the cursor variable.
- *Ease of maintenance* If you need to change the cursor, then you only need to make the change in one place: the stored procedure. There is no need to change each application.
- *Convenient security* The user of the application is the username used when the application connects to the server. The user must have `EXECUTE` permission on the stored procedure that opens the cursor. But, the user does not need to have `READ` permission on the tables used in the query. This capability can be used to limit access to the columns in the table, as well as access to other stored procedures.

See Also:
Oracle Database PL/SQL User's Guide and Reference for details on cursor variables

Declaring and Opening Cursor Variables

Memory is usually allocated for a cursor variable in the client application using the appropriate `ALLOCATE` statement. In Pro*C, use the `EXEC SQL ALLOCATE <cursor_name>` statement. In OCI, use the Cursor Data Area.

You can also use cursor variables in applications that run entirely in a single server session. You can declare cursor variables in PL/SQL subprograms, open them, and use them as parameters for other PL/SQL subprograms.

Examples of Cursor Variables

This section includes several examples of cursor variable usage in PL/SQL. For additional cursor variable examples that use the programmatic interfaces, refer to the following manuals:

- *Pro*C/C++ Programmer's Guide*
- *Pro*COBOL Programmer's Guide*
- *Oracle Call Interface Programmer's Guide*
- *Oracle SQL*Module for Ada Programmer's Guide*

Fetching Data

The following package defines a PL/SQL cursor variable type `Emp_val_cv_type`, and two procedures. The first procedure, `Open_emp_cv`, opens the cursor variable using a bind variable in the `WHERE` clause. The second procedure, `Fetch_emp_data`, fetches rows from the `Emp_tab` table using the cursor variable.

```
CREATE OR REPLACE PACKAGE Emp_data AS
  TYPE Emp_val_cv_type IS REF CURSOR RETURN Emp_tab%ROWTYPE;
  PROCEDURE Open_emp_cv (Emp_cv           IN OUT Emp_val_cv_type,
                         Dept_number      IN     INTEGER);
  PROCEDURE Fetch_emp_data (emp_cv        IN     Emp_val_cv_type,
                            emp_row       OUT    Emp_tab%ROWTYPE);
END Emp_data;

CREATE OR REPLACE PACKAGE BODY Emp_data AS
  PROCEDURE Open_emp_cv (Emp_cv      IN OUT Emp_val_cv_type,
                         Dept_number IN     INTEGER) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM Emp_tab WHERE deptno = dept_number;
  END open_emp_cv;
  PROCEDURE Fetch_emp_data (Emp_cv       IN  Emp_val_cv_type,
                            Emp_row      OUT Emp_tab%ROWTYPE) IS
  BEGIN
    FETCH Emp_cv INTO Emp_row;
  END Fetch_emp_data;
END Emp_data;
```

The following example shows how to call the `Emp_data` package procedures from a PL/SQL block:

```
DECLARE
-- declare a cursor variable
   Emp_curs Emp_data.Emp_val_cv_type;
   Dept_number Dept_tab.Deptno%TYPE;
   Emp_row Emp_tab%ROWTYPE;

BEGIN
   Dept_number := 20;
-- open the cursor using a variable
   Emp_data.Open_emp_cv(Emp_curs, Dept_number);
-- fetch the data and display it
   LOOP
     Emp_data.Fetch_emp_data(Emp_curs, Emp_row);
     EXIT WHEN Emp_curs%NOTFOUND;
     DBMS_OUTPUT.PUT(Emp_row.Ename || '  ');
     DBMS_OUTPUT.PUT_LINE(Emp_row.Sal);
   END LOOP;
END;
```

Implementing Variant Records

The power of cursor variables comes from their ability to point to different cursors. In the following package example, a discriminant is used to open a cursor variable to point to one of two different cursors:

```
CREATE OR REPLACE PACKAGE Emp_dept_data 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 Emp_tab WHERE Sal > 2000;
    ELSIF Discrim = 2 THEN
      OPEN Cv FOR SELECT * FROM Dept_tab;
    END IF;
  END Open_cv;
END Emp_dept_data;
```

You can call the `Open_cv` procedure to open the cursor variable and point it to either a query on the `Emp_tab` table or the `Dept_tab` table. The following PL/SQL block shows how to fetch using the cursor variable, and then use the `ROWTYPE_MISMATCH` predefined exception to handle either fetch:

```
DECLARE
  Emp_rec  Emp_tab%ROWTYPE;
  Dept_rec Dept_tab%ROWTYPE;
  Cv       Emp_dept_data.CV_TYPE;

BEGIN
  Emp_dept_data.open_cv(Cv, 1); -- Open Cv For Emp_tab Fetch
  Fetch cv INTO Dept_rec;       -- but fetch into Dept_tab record
                                -- which raises ROWTYPE_MISMATCH
  DBMS_OUTPUT.PUT(Dept_rec.Deptno);
  DBMS_OUTPUT.PUT_LINE('  ' || Dept_rec.Loc);
```

```
EXCEPTION
  WHEN ROWTYPE_MISMATCH THEN
    BEGIN
      DBMS_OUTPUT.PUT_LINE
           ('Row type mismatch, fetching Emp_tab data...');
      FETCH Cv INTO Emp_rec;
      DBMS_OUTPUT.PUT(Emp_rec.Deptno);
      DBMS_OUTPUT.PUT_LINE('  ' || Emp_rec.Ename);
    END;
```

Handling PL/SQL Compile-Time Errors

When you use SQL*Plus to submit PL/SQL code, and when the code contains errors, you receive notification that compilation errors have occurred, but there is no immediate indication of what the errors are. For example, if you submit a standalone (or stored) procedure `PROC1` in the file `proc1.sql` as follows:

```
SQL> @proc1
```

And, if there are one or more errors in the code, then you receive a notice such as the following:

```
MGR-00072: Warning: Procedure proc1 created with compilation errors
```

In this case, use the `SHOW ERRORS` statement in SQL*Plus to get a list of the errors that were found. `SHOW ERRORS` with no argument lists the errors from the most recent compilation. You can qualify `SHOW ERRORS` using the name of a procedure, function, package, or package body:

```
SQL> SHOW ERRORS PROC1
SQL> SHOW ERRORS PROCEDURE PROC1
```

See Also:
*SQL*Plus User's Guide and Reference* for complete information about the `SHOW ERRORS` statement

Note:
Before issuing the `SHOW ERRORS` statement, use the `SET LINESIZE` statement to get long lines on output. The value 132 is usually a good choice. For example:

```
SET LINESIZE 132
```

Assume that you want to create a simple procedure that deletes records from the employee table using SQL*Plus:

```
CREATE OR REPLACE PROCEDURE Fire_emp(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab WHER Empno = Emp_id;
   END
/
```

Notice that the `CREATE PROCEDURE` statement has two errors: the `DELETE` statement has an error (the `E` is absent from `WHERE`), and the semicolon is missing after `END`.

After the `CREATE PROCEDURE` statement is entered and an error is returned, a `SHOW ERRORS` statement returns the following lines:

```
SHOW ERRORS;

ERRORS FOR PROCEDURE Fire_emp:
LINE/COL       ERROR
-------------- --------------------------------------------
3/27           PL/SQL-00103: Encountered the symbol "EMPNO" wh. . .
5/0            PL/SQL-00103: Encountered the symbol "END" when . . .
2 rows selected.
```

Notice that each line and column number where errors were found is listed by the `SHOW ERRORS` statement.

Alternatively, you can query the following data dictionary views to list errors when using any tool or application:

- `USER_ERRORS`

- ALL_ERRORS
- DBA_ERRORS

The error text associated with the compilation of a procedure is updated when the procedure is replaced, and it is deleted when the procedure is dropped.

Original source code can be retrieved from the data dictionary using the following views: ALL_SOURCE, USER_SOURCE, and DBA_SOURCE.

See Also:
Oracle Database Reference for more information about these data dictionary views

Handling Run-Time PL/SQL Errors

Oracle Database allows user-defined errors in PL/SQL code to be handled so that user-specified error numbers and messages are returned to the client application. After received, the client application can handle the error based on the user-specified error number and message returned by Oracle Database.

User-specified error messages are returned using the RAISE_APPLICATION_ERROR procedure. For example:

```
RAISE_APPLICATION_ERROR(Error_number, 'text', Keep_error_stack)
```

This procedure stops procedure execution, rolls back any effects of the procedure, and returns a user-specified error number and message (unless the error is trapped by an exception handler). ERROR_NUMBER must be in the range of -20000 to -20999.

Error number -20000 should be used as a generic number for messages where it is important to relay information to the user, but having a unique error number is not required. Text must be a character expression, 2 Kbytes or less (longer messages are ignored). Keep_error_stack can be TRUE if you want to add the error to any already on the stack, or FALSE if you want to replace the existing errors. By default, this option is FALSE.

Note:
Some of the Oracle Database packages, such as DBMS_OUTPUT, DBMS_DESCRIBE, and DBMS_ALERT, use application error numbers in the range -20000 to -20005. Refer to the descriptions of these packages for more information.

The RAISE_APPLICATION_ERROR procedure is often used in exception handlers or in the logic of PL/SQL code. For example, the following exception handler selects the string for the associated user-defined error message and calls the RAISE_APPLICATION_ERROR procedure:

```
...
WHEN NO_DATA_FOUND THEN
   SELECT Error_string INTO Message
   FROM Error_table,
   V$NLS_PARAMETERS V
   WHERE Error_number = -20101 AND Lang = v.value AND
      v.parameter = "NLS_LANGUAGE";
   Raise_application_error(-20101, Message);
...
```

See Also:
"Handling Errors in Remote Procedures" for information on exception handling when calling remote procedures

The following section includes an example of passing a user-specified error number from a trigger to a procedure.

Declaring Exceptions and Exception Handling Routines

User-defined exceptions are explicitly defined and signaled within the PL/SQL block to control processing of errors specific to the application. When an exception is *raised* (signaled), the usual execution of the PL/SQL block stops, and a routine called an exception handler is called. Specific exception handlers can be written to handle any internal or user-defined exception.

Application code can check for a condition that requires special attention using an IF statement. If there is an error condition, then two options are available:

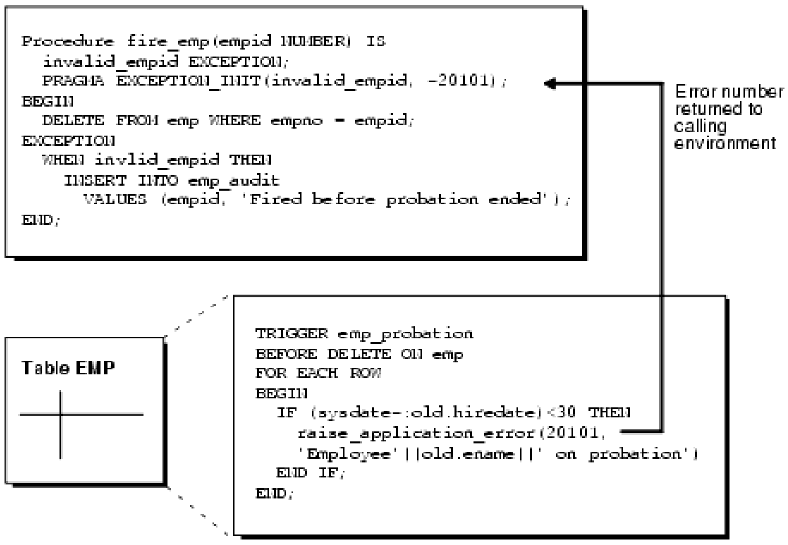
- Enter a `RAISE` statement that names the appropriate exception. A `RAISE` statement stops the execution of the procedure, and control passes to an exception handler (if any).
- Call the `RAISE_APPLICATION_ERROR` procedure to return a user-specified error number and message.

You can also define an exception handler to handle user-specified error messages. For example, Figure 7-2 illustrates the following:

- An exception and associated exception handler in a procedure
- A conditional statement that checks for an error (such as transferring funds not available) and enters a user-specified error number and message within a trigger
- How user-specified error numbers are returned to the calling environment (in this case, a procedure), and how that application can define an exception that corresponds to the user-specified error number

Declare a user-defined exception in a procedure or package body (private exceptions), or in the specification of a package (public exceptions). *Define* an exception handler in the body of a procedure (standalone or package).

Figure 7-2 Exceptions and User-Defined Errors



```
Procedure fire_emp(empid NUMBER) IS
  invalid_empid EXCEPTION;
  PRAGMA EXCEPTION_INIT(invalid_empid, -20101);
BEGIN
  DELETE FROM emp WHERE empno = empid;
EXCEPTION
  WHEN invlid_empid THEN
    INSERT INTO emp_audit
      VALUES (empid, 'Fired before probation ended');
END;
```

```
TRIGGER emp_probation
BEFORE DELETE ON emp
FOR EACH ROW
BEGIN
  IF (sysdate-:old.hiredate)<30 THEN
    raise_application_error(20101,
    'Employee'||old.ename||' on probation')
  END IF;
END;
```

Description of "Figure 7-2 Exceptions and User-Defined Errors"

Unhandled Exceptions

In database PL/SQL program units, an unhandled user-error condition or internal error condition that is not trapped by an appropriate exception handler causes the implicit rollback of the program unit. If the program unit includes a `COMMIT` statement before the point at which the unhandled exception is observed, then the implicit rollback of the program unit can only be completed back to the previous `COMMIT`.

Additionally, unhandled exceptions in database-stored PL/SQL program units propagate back to client-side applications that call the containing program unit. In such an application, only the application program unit call is rolled back (not the entire application program unit), because it is submitted to the database as a SQL statement.

If unhandled exceptions in database PL/SQL program units are propagated back to database applications, then the database PL/SQL code should be modified to handle the exceptions. Your application can also trap for unhandled exceptions when calling database program units and handle such errors appropriately.

Handling Errors in Distributed Queries

You can use a trigger or a stored procedure 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 due to an integrity constraint violation, then Oracle Database returns error number `ORA-02055`. Subsequent statements, or procedure calls, return error number `ORA-02067` until a rollback or a rollback to savepoint is entered.

You should 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, then you should rollback the entire transaction (or rollback to a savepoint) before allowing the application to proceed.

Handling Errors in Remote Procedures

When a procedure is run locally or at a remote location, four types of exceptions can occur:

- PL/SQL user-defined exceptions, which must be declared using the keyword `EXCEPTION`.
- PL/SQL predefined exceptions, such as `NO_DATA_FOUND`.
- SQL errors, such as `ORA-00900` and `ORA-02015`.
- Application exceptions, which are generated using the `RAISE_APPLICATION_ERROR`() procedure.

When using local procedures, all of these messages can be trapped by writing an exception handler, such as shown in the following example:

```
EXCEPTION
    WHEN ZERO_DIVIDE THEN
    /* ...handle the exception */
```

Notice that the `WHEN` clause requires an exception name. If the exception that is raised does not have a name, such as those generated with `RAISE_APPLICATION_ERROR`, then one can be assigned using `PRAGMA_EXCEPTION_INIT`, as shown in the following example:

```
DECLARE
    ...
    Null_salary EXCEPTION;
    PRAGMA EXCEPTION_INIT(Null_salary, -20101);
BEGIN
    ...
    RAISE_APPLICATION_ERROR(-20101, 'salary is missing');
    ...
EXCEPTION
    WHEN Null_salary THEN
        ...
```

When calling a remote procedure, exceptions are also handled by creating a local exception handler. The remote procedure must return an error number to the local calling procedure, which then handles the exception, as shown in the previous example. Because PL/SQL user-defined exceptions always return `ORA-06510` to the local procedure, these exceptions cannot be handled. All other remote exceptions can be handled in the same manner as local exceptions.

Debugging Stored Procedures

Compiling a stored procedure involves fixing any syntax errors in the code. You might need to do additional debugging to make sure that the procedure works correctly, performs well, and recovers from errors. Such debugging might involve:

- Adding extra output statements to verify execution progress and check data values at certain points within the procedure.
- Running a separate debugger to analyze execution in greater detail.

Oracle JDeveloper

Recent releases of Oracle JDeveloper have extensive features for debugging PL/SQL, Java, and multi-language programs. You can get Oracle JDeveloper as part of various Oracle product suites. Often, a more recent release is available as a download at `http://www.oracle.com/technology/`.

Oracle Procedure Builder and TEXT_IO Package

Oracle Procedure Builder is an advanced client/server debugger that transparently debugs your database applications. It lets you run PL/SQL procedures and triggers in a controlled debugging environment, and you can set breakpoints, list the values of variables, and perform other debugging tasks. Oracle Procedure Builder is part of the Oracle Developer tool set. It also provides the `TEXT_IO` package that is useful for printing debug information.

DBMS_OUTPUT Package

You can also debug stored procedures and triggers using the `DBMS_OUTPUT` supplied package. Put `PUT` and `PUT_LINE` statements in your code to output the value of variables and expressions to your terminal.

Privileges for Debugging PL/SQL and Java Stored Procedures

Starting with Oracle Database 10*g*, a new privilege model applies to debugging PL/SQL and Java code running within the database. This model applies whether you are using Oracle JDeveloper, Oracle Developer, or any of the various third-party PL/SQL or Java development environments, and it affects both the `DBMS_DEBUG` and `DBMS_DEBUG_JDWP` APIs.

For a session to connect to a debugger, the effective user at the time of the connect operation must have the `DEBUG CONNECT SESSION` system privilege. This effective user may be the owner of a definer's rights routine involved in making the connect call.

When a debugger becomes connected to a session, the session login user and the currently enabled session-level roles are fixed as the privilege environment for that debugging connection. Any `DEBUG` or `EXECUTE` privileges needed for debugging must be granted to that combination of user and roles.

- To be able to display and change Java public variables or variables declared in a PL/SQL package specification, the debugging connection must be granted either `EXECUTE` or `DEBUG` privilege on the relevant code.
- To be able to either display and change private variables or breakpoint and execute code lines step by step, the debugging connection must be granted `DEBUG` privilege on the relevant code

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

In addition to these privilege requirements, the ability to stop on individual code lines and debugger access to variables are allowed only in code compiled with debug information generated. The `PLSQL_DEBUG` parameter and the `DEBUG` keyword on commands such as `ALTER PACKAGE` can be used to control whether the PL/SQL compiler includes debug information in its results. If it does not, variables will not be accessible, and neither stepping nor breakpoints will stop on code lines. The PL/SQL compiler will never generate debug information for code that has been obfuscated using the PL/SQL `wrap` utility.

See Also:
Oracle Database PL/SQL User's Guide and Reference, "Obfuscating PL/SQL Source Code"

The `DEBUG ANY PROCEDURE` system privilege is equivalent to the `DEBUG` privilege granted on *all* objects in the database. Objects owned by `SYS` are included if the value of the `O7_DICTIONARY_ACCESSIBILITY` parameter is `TRUE`.

A debug role mechanism is available to carry privileges needed for debugging that are not normally enabled in the session. Refer to the documentation on the `DBMS_DEBUG` and `DBMS_DEBUG_JDWP` packages for details on how to specify a debug role and any necessary related password.

The `JAVADEBUGPRIV` role carries the `DEBUG CONNECT SESSION` and `DEBUG ANY PROCEDURE` privileges. Grant it only with the care those privileges warrant.

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

Writing Low-Level Debugging Code

If you are actually writing code that will be part of a debugger, you might need to use packages such as `DBMS_DEBUG_JDWP` or `DBMS_DEBUG`.

DBMS_DEBUG_JDWP Package

The `DBMS_DEBUG_JDWP` package, provided starting with Oracle9*i* Release 2, provides a framework for multi-language debugging that is expected to supersede the `DBMS_DEBUG` package over time. It is especially useful for programs that combine PL/SQL and Java.

DBMS_DEBUG Package

The `DBMS_DEBUG` package, provided starting with Oracle8*i*, implements server-side debuggers and provides a way to debug server-side PL/SQL program units. Several of the debuggers available, such as Oracle Procedure Builder and various third-party vendor solutions, use this API.

See Also:

- *Oracle Procedure Builder Developer's Guide*
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_DEBUG` and `DBMS_OUTPUT` packages and associated privileges
- The Oracle JDeveloper documentation for information on using package `DBMS_DEBUG_JDWP`
- *Oracle Database SQL Reference* for more details on privileges

- The PL/SQL page at `http://www.oracle.com/technology/` for information about writing low-level debug code

---

Calling Stored Procedures

Note:
You may need to set up data structures, similar to the following, for certain examples to work:

```
CREATE TABLE Emp_tab (
   Empno    NUMBER(4) NOT NULL,
   Ename    VARCHAR2(10),
   Job      VARCHAR2(9),
   Mgr      NUMBER(4),
   Hiredate DATE,
   Sal      NUMBER(7,2),
   Comm     NUMBER(7,2),
   Deptno   NUMBER(2));

CREATE OR REPLACE PROCEDURE fire_emp1(Emp_id NUMBER) AS
   BEGIN
      DELETE FROM Emp_tab WHERE Empno = Emp_id;
   END;
VARIABLE Empnum NUMBER;
```

Procedures can be called from many different environments. For example:

- A procedure can be called within the body of another procedure or a trigger.
- A procedure can be interactively called by a user using an Oracle Database tool.
- A procedure can be explicitly called within an application, such as a SQL*Forms or a precompiler application.
- A stored function can be called from a SQL statement in a manner similar to calling a built-in SQL function, such as `LENGTH` or `ROUND`.

This section includes some common examples of calling procedures from within these environments.

See Also:
"Calling Stored Functions from SQL Expressions"

A Procedure or Trigger Calling Another Procedure

A procedure or trigger can call another stored procedure. For example, included in the body of one procedure might be the following line:

```
. . .
Sal_raise(Emp_id, 200);
. . .
```

This line calls the `Sal_raise` procedure. `Emp_id` is a variable within the context of the procedure. Recursive procedure calls are allowed within PL/SQL: A procedure can call itself.

Interactively Calling Procedures From Oracle Database Tools

A procedure can be called interactively from an Oracle Database tool, such as SQL*Plus. For example, to call a procedure named `SAL_RAISE`, owned by you, you can use an anonymous PL/SQL block, as follows:

```
BEGIN
    Sal_raise(7369, 200);
END;
```

Note:
Interactive tools, such as SQL*Plus, require you to follow these lines with a slash (/) to run the PL/SQL block.

An easier way to run a block is to use the SQL*Plus statement `EXECUTE`, which wraps `BEGIN` and `END` statements around the code you enter. For example:

```
EXECUTE Sal_raise(7369, 200);
```

Some interactive tools allow session variables to be created. For example, when using SQL*Plus, the following statement creates a session variable:

```
VARIABLE Assigned_empno NUMBER
```

After defined, any session variable can be used for the duration of the session. For example, you might run a function and capture the return value using a session variable:

```
EXECUTE :Assigned_empno := Hire_emp('JSMITH', 'President',
   1032, SYSDATE, 5000, NULL, 10);
PRINT Assigned_empno;
ASSIGNED_EMPNO
--------------
          2893
```

See Also:

- *SQL*Plus User's Guide and Reference*
- Your tools documentation for information about performing similar operations using your development tool

Calling Procedures within 3GL Applications

A 3GL database application, such as a precompiler or an OCI application, can include a call to a procedure within the code of the application.

To run a procedure within a PL/SQL block in an application, simply call the procedure. The following line within a PL/SQL block calls the `Fire_emp` procedure:

```
Fire_emp1(:Empnun);
```

In this case, :`Empno` is a host (bind) variable within the context of the application.

To run a procedure within the code of a precompiler application, you must use the `EXEC` call interface. For example, the following statement calls the `Fire_emp` procedure in the code of a precompiler application:

```
EXEC SQL EXECUTE
   BEGIN
      Fire_emp1(:Empnum);
   END;
END-EXEC;
```

See Also:
For information about calling PL/SQL procedures from within 3GL applications:

- *Oracle Call Interface Programmer's Guide*
- *Pro*C/C++ Programmer's Guide*
- *Oracle SQL*Module for Ada Programmer's Guide*

Name Resolution When Calling Procedures

References to procedures and packages are resolved according to the algorithm described in the "Rules for Name Resolution in SQL Statements" section of Chapter 2, "Designing Schema Objects".

Privileges Required to Execute a Procedure

If you are the owner of a standalone procedure or package, then you can run the standalone procedure or packaged procedure, or any public procedure or packaged procedure at any time, as described in the previous sections. If you want to run a standalone or packaged procedure owned by another user, then the following conditions apply:

- You must have the `EXECUTE` privilege for the standalone procedure or package containing the procedure, or you must have the `EXECUTE ANY PROCEDURE` system privilege. If you are executing a remote procedure, then you must be granted the `EXECUTE` privilege or `EXECUTE ANY PROCEDURE` system privilege directly, *not* through a role.
- You must include the name of the owner in the call. For example:[Foot 1]

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

- If the procedure is a **definer's-rights procedure**, then it runs with the privileges of the procedure *owner*. The owner must have all the necessary object privileges for any referenced objects.
- If the procedure is an **invoker's-rights procedure**, then it runs with your privileges (as the invoker). In this case, you also need privileges on all referenced objects; that is, all objects accessed by the procedure through external references that are resolved in your schema. You may hold these privileges directly or through a role. Roles are enabled unless an invoker's-rights procedure is called directly or indirectly by a definer's-rights procedure.

Specifying Values for Procedure Arguments

When you call a procedure, specify a value or parameter for each of the procedure's arguments. Identify the argument values using either of the following methods, or a combination of both:

- List the values in the order the arguments appear in the procedure declaration.
- Specify the argument names and corresponding values, in any order.

For example, these statements each call the procedure `Sal_raise` to increase the salary of employee number 7369 by 500:

```
Sal_raise(7369, 500);

Sal_raise(Sal_incr=>500, Emp_id=>7369);

Sal_raise(7369, Sal_incr=>500);
```

The first statement identifies the argument values by listing them in the order in which they appear in the procedure specification.

The second statement identifies the argument values by name and in an order different from that of the procedure specification. If you use argument names, then you can list the arguments in any order.

The third statement identifies the argument values using a combination of these methods. If you use a combination of order and argument names, then values identified in order must precede values identified by name.

If you used the `DEFAULT` option to define default values for `IN` parameters to a subprogram (see the *Oracle Database PL/SQL User's Guide and Reference*),then you can pass different numbers of actual parameters to the first subprogram, accepting or overriding the default values as you please. If an actual value is not passed, then the corresponding default value is used. If you want to assign a value to an argument that occurs after an omitted argument (for which the corresponding default is used), then you must explicitly designate the name of the argument, as well as its value.

Calling Remote Procedures

Call remote procedures using an appropriate database link and the procedure name. The following SQL*Plus statement runs the procedure `Fire_emp` located in the database and pointed to by the local database link named `BOSTON_SERVER`:

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

See Also:
"Handling Errors in Remote Procedures" for information on exception handling when calling remote procedures

Remote Procedure Calls and Parameter Values

You must explicitly pass values to all remote procedure parameters, even if there are defaults. You cannot access remote package variables and constants.

Referencing Remote Objects

Remote objects can be referenced within the body of a locally defined procedure. The following procedure deletes a row from the remote employee table:

```
CREATE OR REPLACE PROCEDURE fire_emp(emp_id NUMBER) IS
BEGIN
    DELETE FROM emp@boston_server WHERE empno = emp_id;
END;
```

The following list explains how to properly call remote procedures, depending on the calling environment.

- Remote procedures (standalone and packaged) can be called from within a procedure, an OCI application, or a precompiler application by specifying the remote procedure name, a database link, and the arguments for the remote procedure.

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

- In the previous example, you could create a synonym for `FIRE_EMP1@BOSTON_SERVER`. This would enable you to call the remote procedure from an Oracle Database tool application, such as a SQL*Forms application, as well from within a procedure, OCI application, or precompiler application.

  ```
  CREATE SYNONYM synonym1 for  fire_emp1@boston_server;
  CREATE OR REPLACE PROCEDURE local_procedure(arg IN NUMBER) AS
  BEGIN
    synonym1(arg);
  END;
  ```

- If you do not want to use a synonym, then you could write a local cover procedure to call the remote procedure.

  ```
  DECLARE
     arg NUMBER;
  BEGIN
     local_procedure(arg);
  END;
  ```

  Here, `local_procedure` is defined as in the first item of this list.

See Also:
"Synonyms for Procedures and Packages"

Caution:
Unlike stored procedures, which use compile-time binding, runtime binding is used when referencing remote procedures. The user account to which you connect depends on the database link.

All calls to remotely stored procedures are assumed to perform updates; therefore, this type of referencing always requires two-phase commit of that transaction (even if the remote procedure is read-only). Furthermore, if a transaction that includes a remote procedure call is rolled back, then the work done by the remote procedure is also rolled back.

A procedure called remotely can usually execute a `COMMIT`, `ROLLBACK`, or `SAVEPOINT` statement, the same as a local procedure. However, there are some differences in behavior:

- If the transaction was originated by a non-Oracle database, as may be the case in XA applications, these operations are not allowed in the remote procedure.
- After doing one of these operations, the remote procedure cannot start any distributed transactions of its own.
- If the remote procedure does not commit or roll back its work, the commit is done implicitly when the database link is closed. In the meantime, further calls to the remote procedure are not allowed because it is still considered to be performing a transaction.

A **distributed update** modifies data on two or more databases. A distributed update is possible using a procedure that includes two or more remote updates that access data on different databases. Statements in the construct are sent to the remote databases, and the execution of the construct succeeds or fails as a unit. If part of a distributed update fails and part succeeds, then a rollback (of the entire transaction or to a savepoint) is required to proceed. Consider this when creating procedures that perform distributed updates.

Pay special attention when using a local procedure that calls a remote procedure. If a timestamp mismatch is found during execution of the local procedure, then the remote procedure is not run, and the local procedure is invalidated.

Synonyms for Procedures and Packages

Synonyms can be created for standalone procedures and packages to do the following:

- Hide the identity of the name and owner of a procedure or package.
- Provide location transparency for remotely stored procedures (standalone or within a package).

When a privileged user needs to call a procedure, an associated synonym can be used. Because the procedures defined within a package are not individual objects (the package is the object), synonyms cannot be created for individual procedures within a package.

Calling Stored Functions from SQL Expressions

You can include user-written PL/SQL functions in SQL expressions. (You must be using PL/SQL release 2.1 or higher.) By using PL/SQL functions in SQL statements, you can do the following:

- 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 used in the `WHERE` clause of a query can filter data using criteria that would otherwise need to be evaluated by the application.
- Manipulate character strings to represent special datatypes (for example, latitude, longitude, or temperature).
- Provide parallel query execution: If the query is parallelized, then SQL statements in your PL/SQL function may also be run in parallel (using the parallel query option).

Using PL/SQL Functions

PL/SQL functions must be created as top-level functions or declared within a package specification before they can be named within a SQL statement. Stored PL/SQL functions are used in the same manner as built-in Oracle functions (such as `SUBSTR` or `ABS`).

PL/SQL functions can be placed wherever an Oracle function can be placed within a SQL statement, or, wherever expressions can occur in SQL. For example, they can be called from the following:

- The select list of the `SELECT` statement.
- The condition of the `WHERE` and `HAVING` clause.
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses.
- The `VALUES` clause of the `INSERT` statement.
- The `SET` clause of the `UPDATE` statement.

You cannot call stored PL/SQL functions from a `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` statement or use them to specify a default value for a column. These situations require an unchanging definition.

Note:
Unlike functions, which are called as part of an expression, procedures are called as statements. Therefore, PL/SQL procedures are *not* directly callable from SQL statements. However, functions called from a PL/SQL statement or referenced in a SQL expression can call a PL/SQL procedure.

Syntax for SQL Calling a PL/SQL Function

Use the following syntax to reference a PL/SQL function from SQL:

```
[[schema.]package.]function_name[@dblink][(param_1...param_n)]
```

For example, to reference a function you created that is called `My_func`, in the `My_funcs_pkg` package, in the `Scott` schema, that takes two numeric parameters, you could call the following:

```
SELECT Scott.My_funcs_pkg.My_func(10,20) FROM dual;
```

Naming Conventions

If only one of the optional schema or package names is given, then the first identifier can be either a schema name or a package name. For example, to determine whether `Payroll` in the reference `Payroll.Tax_rate` is a schema or package name, Oracle Database proceeds as follows:

- Oracle Database first checks for the `Payroll` package in the current schema.
- If the `PAYROLL` package is found in the current schema, then Oracle Database looks for a `Tax_rate` function in the `Payroll` package. If a `Tax_rate` function is not found in the `Payroll` package, then an error message is returned.
- If a `Payroll` package is not found, then Oracle Database looks for a schema named `Payroll` that contains a top-level `Tax_rate` function. If the `Tax_rate` function is not found in the `Payroll` schema, then an error message is returned.

You can also refer to a stored top-level function using any synonym that you have defined for it.

Name Precedence

In SQL statements, the names of database columns take precedence over the names of functions with no parameters. For example, if schema `Scott` creates the following two objects:

```
CREATE TABLE Emp_tab(New_sal NUMBER ...);
CREATE FUNCTION New_sal RETURN NUMBER IS ...;
```

Then, in the following two statements, the reference to `New_sal` refers to the column `Emp_tab.New_sal`:

```
SELECT New_sal FROM Emp_tab;
SELECT Emp_tab.New_sal FROM Emp_tab;
```

To access the function `new_sal`, enter the following:

```
SELECT Scott.New_sal FROM Emp_tab;
```

Example of Calling a PL/SQL Function from SQL

For example, to call the `Tax_rate` PL/SQL function from schema `Scott`, run it against the `Ss_no` and `sal` columns in `Tax_table`, and place the results in the variable `Income_tax`, specify the following:

Note:
You may need to set up data structures similar to the following for certain examples to work:

```
CREATE TABLE Tax_table (
   Ss_no  NUMBER,
   Sal    NUMBER);

CREATE OR REPLACE FUNCTION tax_rate (ssn IN NUMBER, salary IN NUMBER) RETURN NUMBER I
   sal_out NUMBER;
   BEGIN
      sal_out := salary * 1.1;
   END;
```

```
DECLARE
   Tax_id     NUMBER;
   Income_tax NUMBER;
BEGIN
   SELECT scott.tax_rate (Ss_no, Sal)
      INTO Income_tax
      FROM Tax_table
      WHERE Ss_no = Tax_id;
END;
```

These sample calls to PL/SQL functions are allowed in SQL expressions:

```
Circle_area(Radius)
Payroll.Tax_rate(Empno)
scott.Payroll.Tax_rate@boston_server(Dependents, Empno)
```

Arguments

To pass any number of arguments to a function, supply the arguments within the parentheses. You must use positional notation; named notation is not supported. For functions that do not accept arguments, use `()`.

Using Default Values

The stored function `Gross_pay` initializes two of its formal parameters to default values using the `DEFAULT` clause. For example:

```
CREATE OR REPLACE FUNCTION Gross_pay
    (Emp_id  IN NUMBER,
     St_hrs  IN NUMBER DEFAULT 40,
     Ot_hrs  IN NUMBER DEFAULT 0) RETURN NUMBER AS
  ...
```

When calling `Gross_pay` from a procedural statement, you can always accept the default value of `St_hrs`. This is because you can use named notation, which lets you skip parameters. For example:

```
IF Gross_pay(Eenum, Ot_hrs => Otime) > Pay_limit
THEN ...
```

However, when calling `Gross_pay` from a SQL expression, you cannot accept the default value of `St_hrs`, unless you accept the default value of `Ot_hrs`. This is because you cannot use named notation.

Privileges

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

Requirements for Calling PL/SQL Functions from SQL Expressions

To be callable from SQL expressions, a user-defined PL/SQL function must meet the following basic requirements:

- It must be a stored function, *not* a function defined within a PL/SQL block or subprogram.
- It must be a row function, *not* a column (group) function; in other words, it cannot take an entire column of data as its argument.
- All its formal parameters must be `IN` parameters; none can be an `OUT` or `IN OUT` parameter.
- The datatypes of its formal parameters must be Oracle built-in types, such as `CHAR`, `DATE`, or `NUMBER`, *not* PL/SQL types, such as `BOOLEAN`, `RECORD`, or `TABLE`.
- Its return type (the datatype of its result value) must be an Oracle built-in type.

For example, the following stored function meets the basic requirements:

Note:
You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Payroll(

Srate             NUMBER,
 Orate             NUMBER,
 Acctno            NUMBER);
```

```
CREATE FUNCTION Gross_pay
      (Emp_id IN NUMBER,
       St_hrs IN NUMBER DEFAULT 40,
       Ot_hrs IN NUMBER DEFAULT 0) RETURN NUMBER AS
   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;
```

Controlling Side Effects

The **purity** of a stored subprogram (function or procedure) refers to the side effects of that subprogram on database tables or package variables. 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. Various side effects are not allowed when a subprogram is called from a SQL query or DML statement.

In releases prior to Oracle8*i*, Oracle Database leveraged the PL/SQL compiler to enforce restrictions during the compilation of a stored subprogram or a SQL statement. Starting with Oracle8*i*, the compile-time restrictions were relaxed, and a smaller set of restrictions are enforced during execution.

This change provides uniform support for stored subprograms written in PL/SQL, Java, and C, and it allows programmers the most flexibility possible.

Restrictions

When a SQL statement is run, checks are made to see if it is logically embedded within the execution of an already running SQL statement. This occurs if the statement is run from a trigger or from a subprogram that was in turn called from the already running SQL statement. In these cases, further checks occur to determine if the new SQL statement is safe in the specific context.

The following restrictions are enforced on subprograms:

- A subprogram called from a query or DML statement may not end the current transaction, create or rollback to a savepoint, or `ALTER` the system or session.
- A subprogram called from a query (`SELECT`) statement or from a parallelized DML statement may not execute a DML statement or otherwise modify the database.
- A subprogram called from a DML statement may not read or modify the particular table being modified by that DML statement.

These restrictions apply regardless of what mechanism is used to run the SQL statement inside the subprogram or trigger. For example:

- They apply to a SQL statement called from PL/SQL, whether embedded directly in a subprogram or trigger body, run using the native dynamic mechanism (`EXECUTE IMMEDIATE`), or run using the `DBMS_SQL` package.
- They apply to statements embedded in Java with SQLJ syntax or run using JDBC.
- They apply to statements run with OCI using the callback context from within an "external" C function.

You can avoid these restrictions if the execution of the new SQL statement is not logically embedded in the context of the already running statement. PL/SQL's autonomous transactions provide one escape (see "Autonomous Transactions"). Another escape is available using Oracle Call Interface (OCI) from an external C function, if you create a new connection rather than using the handle available from the `OCIExtProcContext` argument.

Declaring a Function

You can use the keywords `DETERMINISTIC` and `PARALLEL_ENABLE` in the syntax for declaring a function. These are optimization hints that inform the query optimizer and other software components about the following:

- Functions that need not be called redundantly
- Functions permitted within a parallelized query or parallelized DML statement

Only functions that are `DETERMINISTIC` are allowed in function-based indexes and in certain snapshots and materialized views.

A deterministic function depends solely on the values passed into it as arguments and does not reference or modify the contents of package variables or the database or have other side-effects. Such a function produces the same result value for any combination of argument values passed into it.

You place the `DETERMINISTIC` keyword after the return value type in a declaration of the function. For example:

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN P1 * 2;
END;
```

You may place this keyword in the following places:

- On a function defined in a `CREATE FUNCTION` statement
- In a function declaration in a `CREATE PACKAGE` statement

- On a method declaration in a CREATE TYPE statement

You should not repeat the keyword on the function or method body in a CREATE PACKAGE BODY or CREATE TYPE BODY statement.

Certain performance optimizations occur on calls to functions that are marked DETERMINISTIC without any other action being required. The following features require that any function used with them be declared DETERMINISTIC:

- Any user-defined function used in a function-based index.
- Any function used in a materialized view, if that view is to qualify for Fast Refresh or is marked ENABLE QUERY REWRITE.

The preceding functions features attempt to use previously calculated results rather than calling the function when it is possible to do so.

Functions that fall in the following categories should typically be DETERMINISTIC:

- Functions used in a WHERE, ORDER BY, or GROUP BY clause
- Functions that MAP or ORDER methods of a SQL type
- Functions that in any other way help determine whether or where a row should appear in a result set

Oracle Database cannot require that you should explicitly declare functions in the preceding categories as DETERMINISTIC without breaking existing applications, but the use of the keyword might be a wise choice of style within your application.

Keep the following points in mind when you create DETERMINISTIC functions:

- The database cannot recognize if the behavior of the function is indeed deterministic. If the DETERMINISTIC keyword is applied to a function whose behavior is not truly deterministic, then the result of queries involving that function is unpredictable.
- If you change the semantics of a DETERMINISTIC function and recompile it, then existing function-based indexes and materialized views report results for the prior version of the function. Thus, if you change the semantics of a function, you must manually rebuild any dependent function-based indexes and materialized views.

See Also:
Oracle Database SQL Reference for an account of CREATE FUNCTION restrictions

Parallel Query and Parallel DML

Oracle Database's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement which is run in parallel may have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process.

Each process has its own copy of package variables. When parallel execution begins, these are initialized based on the information in the package specification and body as if a new user is logging into the system; the values in package variables are not copied from the original login session. And changes made to package variables are not automatically propagated between the various sessions or back to the original session. Java STATIC class attributes are similarly initialized and modified independently in each process. Because a function can use package (or Java STATIC) variables to accumulate some value across the various rows it encounters, Oracle Database cannot assume that it is safe to parallelize the execution of all user-defined functions.

For query (SELECT) statements in Oracle Database versions prior to 8.1.5, the parallel query optimization looked to see if a function was noted as RNPS and WNPS in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as both RNPS and WNPS could be run in parallel. Functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine if they were pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

See Also:
"PRAGMA RESTRICT_REFERENCES – for Backward Compatibility"

For DML statements in Oracle Database versions prior to 8.1.5, the parallelization optimization looked to see if a function was noted as having all four of RNDS, WNDS, RNPS and WNPS specified in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine if they were actually pure enough; parallelized execution might occur even though a pragma cannot be specified on these functions.

Oracle Database versions 8.1.5 and later continue to parallelize those functions that earlier versions recognize as parallelizable. The PARALLEL_ENABLE keyword is the preferred way to mark your code as safe for parallel execution. This keyword is syntactically similar to DETERMINISTIC as described in "Declaring a Function"; it is placed after the return value type in a declaration of the function, as in:

```
CREATE FUNCTION F1 (P1 NUMBER) RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
```

```
  RETURN P1 * 2;
END;
```

A PL/SQL function defined with `CREATE FUNCTION` may still be run in parallel without any explicit declaration that it is safe to do so, if the system can determine that it neither reads nor writes package variables nor calls any function that might do so. A Java method or C function is never seen by the system as safe to run in parallel, unless the programmer explicitly indicates `PARALLEL_ENABLE` on the "call specification", or provides a `PRAGMA RESTRICT_REFERENCES` indicating that the function is sufficiently pure.

An additional runtime restriction is imposed on functions run in parallel as part of a parallelized DML statement. Such a function is not permitted to in turn execute a DML statement; it is subject to the same restrictions that are enforced on functions that are run inside a query (`SELECT`) statement.

See Also:
"Restrictions"

PRAGMA RESTRICT_REFERENCES – for Backward Compatibility

In Oracle Database versions prior to 8.1.5 (Oracle8*i*), programmers used the pragma `RESTRICT_REFERENCES` to assert the purity level of a subprogram. In subsequent versions, use the hints `parallel-enable` and `deterministic`, instead, to communicate subprogram purity to Oracle Database.

You can remove `RESTRICT_REFERENCES` from your code. However, this pragma remains available for *backward compatibility* in situations where one of the following is true:

- It is impossible or impractical to edit existing code to remove `RESTRICT_REFERENCES` completely. If you do not remove it from a subprogram *S1* that depends on another subprogram *S2*, then `RESTRICT_REFERENCES` might also be needed in *S2*, so that *S1* will compile.
- Replacing `RESTRICT_REFERENCES` in existing code with hints `parallel-enable` and `deterministic` would negatively affect the behavior of new, dependent code. Use `RESTRICT_REFERENCES` to preserve the behavior of the existing code.

An existing PL/SQL application can thus continue using the pragma even on new functionality, to ease integration with the existing code. Do not use the pragma in a wholly new application.

If you use the pragma `RESTRICT_REFERENCES`, place it in a package specification, not in a package body. It must follow the declaration of a subprogram (function or procedure), but it need not follow immediately. Only one pragma can reference a given subprogram declaration.

Note:
The pragma `RESTRICT_REFERENCES` applies to both functions and procedures. Purity levels are important for functions, but also for procedures that are called by functions.

To code the pragma `RESTRICT_REFERENCES`, use the following syntax:

```
PRAGMA RESTRICT_REFERENCES (
    Function_name, WNDS [, WNPS] [, RNDS] [, RNPS] [, TRUST] );
```

Where:

Keyword	Description
`WNDS`	The subprogram writes no database state (does not modify database tables).
`RNDS`	The subprogram reads no database state (does not query database tables).
`WNPS`	The subprogram writes no package state (does not change the values of packaged variables).
`RNPS`	The subprogram reads no package state (does not reference the values of packaged variables).
`TRUST`	The other restrictions listed in the pragma are not enforced; they are simply assumed to be true. This allows easy calling from functions that have `RESTRICT_REFERENCES` declarations to those that do not.

You can pass the arguments in any order. If any SQL statement inside the subprogram body violates a rule, then you get an error when the statement is parsed.

In the following example, the function `compound` neither reads nor writes database or package state; therefore, you can assert the maximum purity level. Always assert the highest purity level that a subprogram allows. That way, the PL/SQL compiler never rejects the subprogram unnecessarily.

Note:
You may need to set up the following data structures for certain examples here to work:

```
CREATE TABLE Accts (
 Yrs      NUMBER,
 Amt      NUMBER,
 Acctno   NUMBER,
 Rte      NUMBER);
```

```
CREATE PACKAGE Finance AS  -- package specification
   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  --package body
   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;
```

Later, you might call `compound` from a PL/SQL block, as follows:

```
DECLARE
   Interest NUMBER;
   Acct_id NUMBER;
BEGIN
   SELECT Finance.Compound(Yrs, Amt, Rte)  -- function call
   INTO   Interest
   FROM   Accounts
   WHERE  Acctno = Acct_id;
```

Using the Keyword TRUST

The keyword `TRUST` in the `RESTRICT_REFERENCES` syntax allows easy calling from functions that have `RESTRICT_REFERENCES` declarations to those that do not. When `TRUST` is present, the restrictions listed in the pragma are not actually enforced, but rather are simply assumed to be true.

When calling from a section of code that is using pragmas to one that is not, there are two likely usage styles. One is to place a pragma on the routine to be called, for example on a "call specification" for a Java method. Then, calls from PL/SQL to this method will complain if the method is less restricted than the calling subprogram. For example:

```
CREATE OR REPLACE PACKAGE P1 IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
      PRAGMA RESTRICT_REFERENCES(F1,WNDS,TRUST);
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;

   PRAGMA RESTRICT_REFERENCES(F2,WNDS);
END;

CREATE OR REPLACE PACKAGE BODY P1 IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

Here, `F2` can call `F1`, as `F1` has been declared to be `WNDS`.

The other approach is to mark only the caller, which may then make a call to any subprogram without complaint. For example:

```
CREATE OR REPLACE PACKAGE P1a IS
   FUNCTION F1 (P1 NUMBER) RETURN NUMBER IS
      LANGUAGE JAVA NAME 'CLASS1.METHODNAME(int) return int';
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER;
   PRAGMA RESTRICT_REFERENCES(F2,WNDS,TRUST);
END;

CREATE OR REPLACE PACKAGE BODY P1a IS
   FUNCTION F2 (P1 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN F1(P1);
   END;
END;
```

Here, `F2` can call `F1` because while `F2` is promised to be `WNDS` (because `TRUST` is specified), the body of `F2` is not actually examined to determine if it truly satisfies the `WNDS` restriction. Because `F2` is not examined, its call to `F1` is allowed, even though there is no `PRAGMA RESTRICT_REFERENCES` for F1.

Differences between Static and Dynamic SQL Statements.

Static `INSERT`, `UPDATE`, and `DELETE` statements do not violate `RNDS` if these statements do not explicitly read any database states, such as columns of a table. However, dynamic `INSERT`, `UPDATE`, and `DELETE` statements *always* violate `RNDS`, regardless of whether or not the statements explicitly read database states.

The following `INSERT` violates `RNDS` if it is executed dynamically, but it does *not* violate `RNDS` if it is executed statically.

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

The following `UPDATE` always violates `RNDS` statically and dynamically, because it explicitly reads the column `name` of `my_table`.

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

Overloading Packaged PL/SQL Functions

PL/SQL lets you **overload** packaged (but not standalone) functions: You can use the same name for different functions if their formal parameters differ in number, order, or datatype family.

However, a `RESTRICT_REFERENCES` pragma can apply to only one function declaration. Therefore, a pragma that references the name of overloaded functions always applies to the nearest preceding function declaration.

In this example, the pragma applies to the second declaration of `valid`:

```
CREATE PACKAGE Tests AS
    FUNCTION Valid (x NUMBER) RETURN CHAR;
    FUNCTION Valid (x DATE) RETURN CHAR;
    PRAGMA RESTRICT_REFERENCES (valid, WNDS);
 END;
```

Serially Reusable PL/SQL Packages

PL/SQL packages usually consume user global area (UGA) memory corresponding to the number of package variables and cursors in the package. This limits scalability, because the memory increases linearly with the number of users. The solution is to allow some packages to be marked as `SERIALLY_REUSABLE` (using pragma syntax).

For serially reusable packages, the package global memory is not kept in the UGA for each user; rather, it is kept in a small pool and reused for different users. This means that the global memory for such a package is only used within a unit of work. At the end of that unit of work, the memory can therefore be released to the pool to be reused by another user (after running the initialization code for all the global variables).

The unit of work for serially reusable packages is implicitly a call to the server; for example, an OCI call to the server, or a PL/SQL RPC call from a client to a server, or an RPC call from a server to another server.

Package States

The state of a nonreusable package (one not marked `SERIALLY_REUSABLE`) persists for the lifetime of a session. A package **state** includes global variables, cursors, and so on.

The state of a serially reusable package persists only for the lifetime of a call to the server. On a subsequent call to the server, if a reference is made to the serially reusable package, then Oracle Database creates a new *instantiation* of the serially reusable package and initializes all the global variables to `NULL` or to the default values provided. Any changes made to the serially reusable package state in the previous calls to the server are not visible.

Note:
Creating a new instantiation of a serially reusable package on a call to the server does not necessarily imply that Oracle Database allocates memory or configures the instantiation object. Oracle Database looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in the SGA.

At the end of the call to the server, this work area is returned back to the LRU pool. The reason for keeping the pool in the SGA is that the work area can be reused across users who have requests for the same package.

Why Serially Reusable Packages?

Because the state of a non-reusable package persists for the lifetime of the session, this locks up UGA memory for the whole session. In applications, such as Oracle Office, a log-on session can typically exist for days together. Applications often need to use certain packages only for certain localized periods in the session and would ideally like to de-instantiate the package state in the middle of the session, after they are done using the package.

With `SERIALLY_REUSABLE` packages, application developers have a way of modelling their applications to manage their memory better for scalability. Package state that they care about only for the duration of a call to the server should be captured in `SERIALLY_REUSABLE` packages.

Syntax of Serially Reusable Packages

A package can be marked serially reusable by a pragma. The syntax of the pragma is:

```
PRAGMA SERIALLY_REUSABLE;
```

A package specification can be marked serially reusable, whether or not it has a corresponding package body. If the package has a body, then the body must have the serially reusable pragma, if its corresponding specification has the pragma; it cannot have the serially reusable pragma unless the specification also has the pragma.

Semantics of Serially Reusable Packages

A package that is marked `SERIALLY_REUSABLE` has the following properties:

- Its package variables are meant for use only within the work boundaries, which correspond to calls to the server (either OCI call boundaries or PL/SQL RPC calls to the server).

  Note:
  If the application programmer makes a mistake and depends on a package variable that is set in a previous unit of work, then the application program can fail. PL/SQL cannot check for such cases.

- A pool of package instantiations is kept, and whenever a "unit of work" needs this package, one of the instantiations is "reused", as follows:
  - The package variables are reinitialized (for example, if the package variables have default values, then those values are reinitialized).
  - The initialization code in the package body is run again.
- At the "end work" boundary, cleanup is done.
  - If any cursors were left open, then they are silently closed.
  - Some non-reusable secondary memory is freed (such as memory for collection variables or long `VARCHAR2`s).
  - This package instantiation is returned back to the pool of reusable instantiations kept for this package.
- Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are called from SQL statements. If you try, then Oracle Database generates an error.

Examples of Serially Reusable Packages

This section presents a few examples of serially reusable packages.

Example 1: How Package Variables Act Across Call Boundaries

This example has a serially reusable package specification (there is no body).

```
CONNECT Scott/Tiger

CREATE OR REPLACE PACKAGE Sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  N NUMBER := 5;                -- default initialization
END Sr_pkg;
```

Suppose your Enterprise Manager (or SQL*Plus) application issues the following:

```
CONNECT Scott/Tiger

# first CALL to server
BEGIN
   Sr_pkg.N := 10;
END;

# second CALL to server
BEGIN
   DBMS_OUTPUT.PUT_LINE(Sr_pkg.N);
END;
```

This program prints:

```
5
```

Note:
If the package had not had the pragma `SERIALLY_REUSABLE`, the program would have printed '10'.

Example 2: How Package Variables Act Across Call Boundaries

This example has both a package specification and package body, which are serially reusable.

```
CONNECT Scott/Tiger

DROP PACKAGE Sr_pkg;
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   TYPE Str_table_type IS TABLE OF VARCHAR2(200) INDEX BY BINARY_INTEGER;
   Num     NUMBER        := 10;
   Str     VARCHAR2(200) := 'default-init-str';
   Str_tab STR_TABLE_TYPE;

    PROCEDURE Print_pkg;
    PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2);
END Sr_pkg;
CREATE OR REPLACE PACKAGE BODY Sr_pkg IS
   -- the body is required to have the pragma because the
  -- specification of this package has the pragma
  PRAGMA SERIALLY_REUSABLE;
   PROCEDURE Print_pkg IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('num: ' || Sr_pkg.Num);
      DBMS_OUTPUT.PUT_LINE('str: ' || Sr_pkg.Str);
      DBMS_OUTPUT.PUT_LINE('number of table elems: ' || Sr_pkg.Str_tab.Count);
      FOR i IN 1..Sr_pkg.Str_tab.Count LOOP
         DBMS_OUTPUT.PUT_LINE(Sr_pkg.Str_tab(i));
      END LOOP;
   END;
   PROCEDURE Init_and_print_pkg(N NUMBER, V VARCHAR2) IS
   BEGIN
   -- init the package globals
      Sr_pkg.Num := N;
      Sr_pkg.Str := V;
      FOR i IN 1..n LOOP
         Sr_pkg.Str_tab(i) := V || ' ' || i;
   END LOOP;
   -- print the package
   Print_pkg;
   END;
 END Sr_pkg;

SET SERVEROUTPUT ON;

Rem SR package access in a CALL:

BEGIN
   -- initialize and print the package
   DBMS_OUTPUT.PUT_LINE('Initing and printing pkg state..');
   Sr_pkg.Init_and_print_pkg(4, 'abracadabra');
   -- print it in the same call to the server.
   -- we should see the initialized values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the same CALL...');
   Sr_pkg.Print_pkg;
END;

Initing and printing pkg state..
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
Printing package state in the same CALL...
num: 4
str: abracadabra
number of table elems: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4

REM SR package access in subsequent CALL:
BEGIN
   -- print the package in the next call to the server.
   -- We should that the package state is reset to the initial (default) values.
   DBMS_OUTPUT.PUT_LINE('Printing package state in the next CALL...');
   Sr_pkg.Print_pkg;
END;
Statement processed.
Printing package state in the next CALL...
num: 10
str: default-init-str
number of table elems: 0
```

Example 3: Open Cursors in Serially Reusable Packages at Call Boundaries

This example demonstrates that any open cursors in serially reusable packages get closed automatically at the end of a work boundary (which is a call). Also, in a new call, these cursors need to be opened again.

```
REM  For serially reusable pkg: At the end work boundaries
REM  (which is currently the OCI call boundary) all open
REM  cursors will be closed.
REM
REM  Because the cursor is closed - every time we fetch we
REM  will start at the first row again.

CONNECT Scott/Tiger
DROP PACKAGE  Sr_pkg;
DROP TABLE People;
CREATE TABLE People (Name VARCHAR2(20));
INSERT INTO  People  VALUES ('ET');
INSERT INTO  People  VALUES ('RAMBO');
```

```
CREATE OR REPLACE PACKAGE Sr_pkg IS
   PRAGMA SERIALLY_REUSABLE;
   CURSOR C IS SELECT Name FROM People;
END Sr_pkg;
SQL> SET SERVEROUTPUT ON;
SQL>
CREATE OR REPLACE PROCEDURE Fetch_from_cursor IS
Name VARCHAR2(200);
BEGIN
   IF (Sr_pkg.C%ISOPEN) THEN
      DBMS_OUTPUT.PUT_LINE('cursor is already open.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('cursor is closed; opening now.');
      OPEN Sr_pkg.C;
   END IF;
   -- fetching from cursor.
   FETCH sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   FETCH Sr_pkg.C INTO name;
   DBMS_OUTPUT.PUT_LINE('fetched: ' || Name);
   -- Oops forgot to close the cursor (Sr_pkg.C).
   -- But, because it is a Serially Reusable pkg's cursor,
   -- it will be closed at the end of this CALL to the server.
END;
EXECUTE fetch_from_cursor;
cursor is closed; opening now.
fetched: ET
fetched: RAMBO
```

Returning Large Amounts of Data from a Function

In a data warehousing environment, you might use a PL/SQL function to transform large amounts of data. Perhaps the data is passed 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.

In this technique:

- The producer function uses the `PIPELINED` keyword in its declaration.
- The producer function uses an `OUT` parameter that is a record, corresponding to a row in the result set.
- As each output record is completed, it is sent to the consumer function using the `PIPE ROW` keyword.
- The producer function ends with a `RETURN` statement that does not specify any return value.
- The consumer function or SQL statement uses the `TABLE` keyword to treat the resulting rows like a regular table.

For example:

```
CREATE FUNCTION StockPivot(p refcur_pkg.refcur_t) RETURN TickerTypeSet PIPELINED IS
  out_rec TickerType := TickerType(NULL,NULL,NULL);
  in_rec p%ROWTYPE;
BEGIN
  LOOP
-- Function accepts multiple rows through a REF CURSOR argument.
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
-- Return value is a record type that matches the table definition.
    out_rec.ticker := in_rec.Ticker;
    out_rec.PriceType := 'O';
    out_rec.price := in_rec.OpenPrice;
-- Once a result row is ready, we send it back to the calling program,
-- and continue processing.
    PIPE ROW(out_rec);
-- This function outputs twice as many rows as it receives as input.
    out_rec.PriceType := 'C';
    out_rec.Price := in_rec.ClosePrice;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
```

```
-- The function ends with a RETURN statement that does not specify any value.
  RETURN;
END;
/

-- Here we use the result of this function in a SQL query.
SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable)));

-- Here we use the result of this function in a PL/SQL block.
DECLARE
  total NUMBER := 0;
  price_type VARCHAR2(1);
BEGIN
  FOR item IN (SELECT * FROM TABLE(StockPivot(CURSOR(SELECT * FROM StockTable))))
  LOOP
-- Access the values of each output row.
-- We know the column names based on the declaration of the output type.
-- This computation is just for illustration.
    total := total + item.price;
    price_type := item.price_type;
  END LOOP;
END;
/
```

Coding Your Own Aggregate Functions

To analyze a set of rows and compute a result value, you can code your own aggregate function that works the same as a built-in aggregate like `SUM`:

- Define a SQL object type that defines these member functions:
  - `ODCIAggregateInitialize`
  - `ODCIAggregateIterate`
  - `ODCIAggregateMerge`
  - `ODCIAggregateTerminate`
- Code the member functions. In particular, `ODCIAggregateIterate` accumulates the result as it is called once for each row that is processed. Store any intermediate results using the attributes of the object type.
- Create the aggregate function, and associate it with the new object type.
- Call the aggregate function from SQL queries, DML statements, or other places that you might use the built-in aggregates. You can include typical options such as `DISTINCT` and `ALL` in the calls to the aggregate function.

See Also:
Oracle Database Data Cartridge Developer's Guide for details of this process and the requirements for the member functions

---

Footnote Legend

Footnote 1: You may need to set up the following data structures for certain examples to work: *`CONNECT SYS/password AS SYSDBA`*`;CREATE USER Jward IDENTIFIED BY Jward;GRANT CREATE ANY PACKAGE TO Jward;GRANT CREATE SESSION TO Jward;GRANT EXECUTE ANY PROCEDURE TO Jward;CONNECT Scott/Tiger`