7

Design Considerations for PL/SQL Code

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- **Use package specifications to create standard constants and exceptions**
- **Write and call local subprograms**
- **Set the `AUTHID` directive to control the run-time privileges of a subprogram**
- **Execute subprograms to perform autonomous transactions**
- **Use bulk binding and the `RETURNING` clause with DML**
- **Pass parameters by reference using a `NOCOPY` hint**
- **Use the `PARALLEL ENABLE` hint for optimization**

ORACLE

Lesson Aim

In this lesson, you learn to use package specifications to standardize names for constant values and exceptions. You learn how to create subprograms in the `Declaration` section of any PL/SQL block for using locally in the block. The `AUTHID` compiler directive is discussed to show how you can manage run-time privileges of the PL/SQL code, and create independent transactions by using the `AUTONOMOUS TRANSACTION` directive for subprograms.

This lesson also covers some performance considerations that can be applied to PL/SQL applications, such as bulk binding operations with a single SQL statement, the `RETURNING` clause, and the `NOCOPY` and `PARALLEL ENABLE` hints.

Standardizing Constants and Exceptions

Constants and exceptions are typically implemented using a bodiless package (that is, in a package specification).

- **Standardizing helps to:**
  - **Develop programs that are consistent**
  - **Promote a higher degree of code reuse**
  - **Ease code maintenance**
  - **Implement company standards across entire applications**
- **Start with standardization of:**
  - **Exception names**
  - **Constant definitions**

ORACLE

Standardizing Constants and Exceptions

When several developers are writing their own exception handlers in an application, there could be inconsistencies in the handling of error situations. Unless certain standards are adhered to, the situation can become confusing because of the different approaches followed in handling the same error or because of the display of conflicting error messages that confuse users. To overcome these, you can:

- Implement company standards that use a consistent approach to error handling across the entire application
- Create predefined, generic exception handlers that produce consistency in the application
- Write and call programs that produce consistent error messages

All good programming environments promote naming and coding standards. In PL/SQL, a good place to start implementing naming and coding standards is with commonly used constants and exceptions that occur in the application domain.

The PL/SQL package specification construct is an excellent component to support standardization because all identifiers declared in the package specification are public. They are visible to the subprograms that are developed by the owner of the package and all code with `EXECUTE` rights to the package specification.

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  fk_err       EXCEPTION;
  seq_nbr_err  EXCEPTION;
  PRAGMA EXCEPTION_INIT (fk_err, -2292);
  PRAGMA EXCEPTION_INIT (seq_nbr_err, -2277);
  ...
END error_pkg;
/
```

ORACLE

Standardizing Exceptions

In the example in the slide, the `error_pkg` package is a standardized exception package. It declares a set of programmer-defined exception identifiers. Because many of the Oracle database predefined exceptions do not have identifying names, the example package shown in the slide uses the `PRAGMA EXCEPTION_INIT` directive to associate selected exception names with an Oracle database error number. This enables you to refer to any of the exceptions in a standard way in your applications, as in the following example:

```
BEGIN
  DELETE FROM departments
WHERE department_id = deptno;
  ...
EXCEPTION
  WHEN error_pkg.fk_err THEN
  ...
  WHEN OTHERS THEN
  ...
END;
/
```

Standardizing Exception Handling

Consider writing a subprogram for common exception handling to:

- **Display errors based on `SQLCODE` and `SQLERRM` values for exceptions**
- **Track run-time errors easily by using parameters in your code to identify:**
  - **The procedure in which the error occurred**
  - **The location (line number) of the error**
  - **`RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`**

ORACLE

Standardizing Exception Handling

Standardized exception handling can be implemented either as a stand-alone subprogram or a subprogram added to the package that defines the standard exceptions. Consider creating a package with:

- Every named exception that is to be used in the application
- All unnamed, programmer-defined exceptions that are used in the application. These are error numbers –20000 through –20999.
- A program to call `RAISE_APPLICATION_ERROR` based on package exceptions
- A program to display an error based on the values of `SQLCODE` and `SQLERRM`
- Additional objects, such as error log tables, and programs to access the tables

A common practice is to use parameters that identify the name of the procedure and the location in which the error has occurred. This enables you to keep track of run-time errors more easily. An alternative is to use the `RAISE_APPLICATION_ERROR` built-in procedure to keep a stack trace of exceptions that can be used to track the call sequence leading to the error. To do this, set the third optional argument to `TRUE`. For example:

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

This is meaningful when more than one exception is raised in this manner.

Standardizing Constants

For programs that use local variables whose values should not change:

- **Convert the variables to constants to reduce maintenance and debugging**
- **Create one central package specification and place all constants in it**

```
CREATE OR REPLACE PACKAGE constant_pkg IS
  c_order_received CONSTANT VARCHAR(2) := 'OR';
  c_order_shipped  CONSTANT VARCHAR(2) := 'OS';
  c_min_sal        CONSTANT NUMBER(3)  := 900;
  ...
END constant_pkg;
```

ORACLE

Standardizing Constants

By definition, a variable's value changes, whereas a constant's value cannot be changed. If you have programs that use local variables whose values should not and do not change, then convert the variables to constants. This can help with the maintenance and debugging of your code.

Consider creating a single shared package with all your constants in it. This makes maintenance and change of the constants much easier. This procedure or package can be loaded on system startup for better performance.

The example in the slide shows the `constant_pkg` package containing a few constants. Refer to any of the package constants in your application as required. Here is an example:

```
BEGIN
  UPDATE employees
     SET salary  = salary + 200
  WHERE salary <= constant_pkg.c_min_sal;
...
END;
/
```

Local Subprograms

- **A local subprogram is a `PROCEDURE` or `FUNCTION` defined in the declarative section.**

```
CREATE PROCEDURE employee_sal(id NUMBER) IS
   emp employees%ROWTYPE;
   FUNCTION tax(salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO emp
   FROM EMPLOYEES WHERE employee_id = id;
   DBMS_OUTPUT.PUT_LINE('Tax: '||tax(emp.salary));
END;
```

- **The local subprogram must be defined at the end of the declarative section.**

ORACLE

Local Subprograms

Local subprograms can drive top-down design. They reduce the size of a module by removing redundant code. This is one of the main reasons for creating a local subprogram. If a module needs the same routine several times, but only this module needs the routine, then define it as a local subprogram.

You can define a named PL/SQL block in the declarative section of any PL/SQL program, procedure, function, or anonymous block provided that it is declared at the end of the `Declaration` section. Local subprograms have the following characteristics:

- They are only accessible to the block in which they are defined.
- They are compiled as part of their enclosing blocks.

The benefits of local subprograms are:

- Reduction of repetitive code
- Improved code readability and ease of maintenance
- Less administration because there is one program to maintain instead of two

The concept is simple. The example shown in the slide illustrates this with a basic example of an income tax calculation of an employee's salary.

Definer's Rights Versus Invoker's Rights

Definer's rights:

- **Used prior to Oracle8*i***
- **Programs execute with the privileges of the creating user.**
- **User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.**

Invoker's rights:

- **Introduced in Oracle8*i***
- **Programs execute with the privileges of the calling user.**
- **User requires privileges on the underlying objects that the procedure accesses.**

ORACLE

Definer's Rights Versus Invoker's Rights

Definer's Rights Model

Prior to Oracle8*i*, all programs executed with the privileges of the user who created the subprogram. This is known as the definer's rights model, which:

- Allows a caller of the program the privilege to execute the procedure, but no privileges on the underlying objects that the procedure accesses
- Requires the owner to have all the necessary object privileges for the objects that the procedure references

For example, if user Scott creates a PL/SQL subprogram `get_employees` that is subsequently invoked by Sarah, then the `get_employees` procedure runs with the privileges of the definer Scott.

Invoker's Rights Model

In the invoker's rights model, which was introduced in Oracle8*i*, programs are executed with the privileges of the calling user. A user of a procedure running with invoker's rights requires privileges on the underlying objects that the procedure references.

For example, if Scott's PL/SQL subprogram `get_employees` is invoked by Sarah, then the `get_employees` procedure runs with the privileges of the invoker Sarah.

Specifying Invoker's Rights

Set `AUTHID` to `CURRENT_USER`:

```
CREATE OR REPLACE PROCEDURE add_dept(
  id NUMBER, name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (id,name,NULL,NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- **Names used in queries, DML, Native Dynamic SQL, and `DBMS_SQL` package are resolved in the invoker's schema**
- **Calls to other packages, functions, and procedures are resolved in the definer's schema**

ORACLE

Specifying Invoker's Rights

Setting Invoker's Rights

The following is the syntax for setting invoker's rights for different PL/SQL subprogram constructs:

```
CREATE FUNCTION name RETURN type AUTHID CURRENT_USER
IS...
CREATE PROCEDURE name AUTHID CURRENT_USER IS…
CREATE PACKAGE name AUTHID CURRENT_USER IS…
CREATE TYPE name AUTHID CURRENT_USER IS OBJECT…
```

In these statements, set `AUTHID` to `DEFINER`, or do not use it for default behavior.

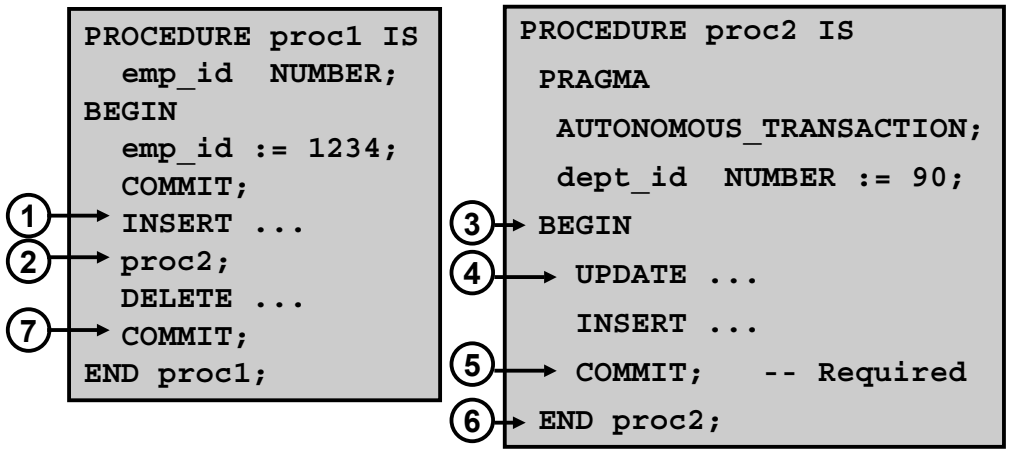
Name Resolution

For a definer's rights procedure, all external references are resolved in the definer's schema. For an invoker's rights procedure, the resolution of external references depends on the kind of statement in which they appear:

- Names used in queries, data manipulation language (DML) statements, dynamic SQL, and `DBMS_SQL` are resolved in the invoker's schema.
- All other statements, such as calls to packages, functions, and procedures, are resolved in the definer's schema.

Autonomous Transactions

- **Are independent transactions started by another main transaction.**
- **Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`**



ORACLE

Autonomous Transactions

A transaction is a series of statements that does a logical unit of work that completes or fails as an integrated unit. Often, one transaction starts another that may need to operate outside the scope of the transaction that started it. That is, in an existing transaction, a required independent transaction may need to commit or roll back changes without affecting the outcome of the starting transaction. For example, in a stock purchase transaction, the customer's information must be committed regardless of whether the overall stock purchase completes. Or, while running that same transaction, you want to log messages to a table even if the overall transaction rolls back.

Since Oracle8*i*, the autonomous transactions were added to make it possible to create an independent transaction. An autonomous transaction (AT) is an independent transaction started by another main transaction (MT). The slide depicts the behavior of an AT:

1. The main transaction begins.
2. A `proc2` procedure is called to start the autonomous transaction.
3. The main transaction is suspended.
4. The autonomous transactional operation begins.
5. The autonomous transaction ends with a commit or roll back operation.
6. The main transaction is resumed.
7. The main transaction ends.

Features of Autonomous Transactions

Autonomous transactions:

- **Are independent of the main transaction**
- **Suspend the calling transaction until it is completed**
- **Are not nested transactions**
- **Do not roll back if the main transaction rolls back**
- **Enable the changes to become visible to other transactions upon a commit**
- **Are demarcated (started and ended) by individual subprograms and not by nested or anonymous PL/SQL blocks**

ORACLE

Features of Autonomous Transactions

Autonomous transactions exhibit the following features:

- Although called within a transaction, autonomous transactions are independent of that transaction. That is, they are not nested transactions.
- If the main transaction rolls back, autonomous transactions do not.
- Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits.
- With their stack-like functionality, only the "top" transaction is accessible at any given time. After completion, the autonomous transaction is popped, and the calling transaction is resumed.
- There are no limits other than resource limits on how many autonomous transactions can be recursively called.
- Autonomous transactions must be explicitly committed or rolled back; otherwise, an error is returned when attempting to return from the autonomous block.
- You cannot use `PRAGMA` to mark all subprograms in a package as autonomous. Only individual routines can be marked autonomous.
- You cannot mark a nested or anonymous PL/SQL block as autonomous.

Using Autonomous Transactions

Example:

```
PROCEDURE bank_trans(cardnbr NUMBER,loc NUMBER) IS
BEGIN
  log_usage (cardnbr, loc);
  INSERT INTO txn VALUES (9001, 1000,...);
END bank_trans;
```

```
PROCEDURE log_usage (card_id NUMBER, loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage
  VALUES (card_id, loc);
  COMMIT;
END log_usage;
```

ORACLE

Using Autonomous Transactions

To define autonomous transactions, you use `PRAGMA AUTONOMOUS_TRANSACTION`. `PRAGMA` instructs the PL/SQL compiler to mark a routine as autonomous (independent). In this context, the term "routine" includes top-level (not nested) anonymous PL/SQL blocks; local, stand-alone, and packaged functions and procedures; methods of a SQL object type; and database triggers. You can code `PRAGMA` anywhere in the declarative section of a routine. However, for readability, it is best placed at the top of the `Declaration` section.

In the example in the slide, you track where the bankcard is used, regardless of whether the transaction is successful. The following are the benefits of autonomous transactions:

- After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit dependencies with the main transaction, so you can log events, increment retry counters, and so on even if the main transaction rolls back.
- More important, autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure's autonomous operations, and the procedure need not know about the application's transaction context. That makes autonomous transactions less error-prone than regular transactions and easier to use.

RETURNING Clause

The RETURNING clause:

- **Improves performance by returning column values with INSERT, UPDATE, and DELETE statements**
- **Eliminates the need for a SELECT statement**

```
CREATE PROCEDURE update_salary(emp_id NUMBER) IS
  name    employees.last_name%TYPE;
  new_sal employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * 1.1
  WHERE employee_id = emp_id
  RETURNING last_name, salary INTO name, new_sal;
END update_salary;
/
```

ORACLE

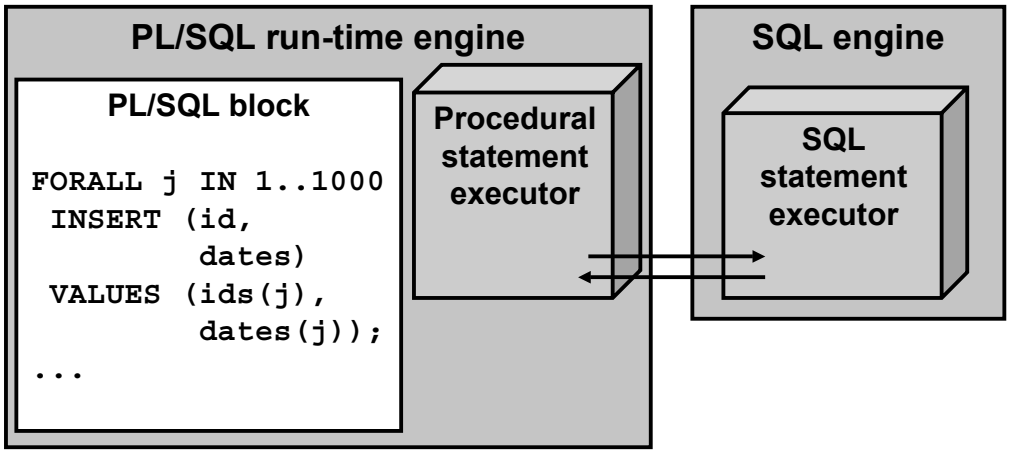
RETURNING Clause

Often, applications need information about the row affected by a SQL operation—for example, to generate a report or to take a subsequent action. The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to SELECT the row after an INSERT or UPDATE, or before a DELETE. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

The example in the slide shows how to update the salary of an employee and, at the same time, retrieve the employee's last name and new salary into a local PL/SQL variable.

Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



ORACLE

Bulk Binding

The Oracle server uses two engines to run PL/SQL blocks and subprograms:

- The PL/SQL run-time engine, which runs procedural statements but passes the SQL statements to the SQL engine
- The SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine

During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty for excessive amounts of SQL processing. This is typical of applications that have a SQL statement in a loop that uses indexed collection elements values. Collections include nested tables, varying arrays, index-by tables, and host arrays.

Performance can be substantially improved by minimizing the number of context switches through the use of bulk binding. Bulk binding causes an entire collection to be bound in one call, a context switch, to the SQL engine. That is, a bulk bind process passes the entire collection of values back and forth between the two engines in a single context switch, compared with incurring a context switch for each collection element in an iteration of a loop. The more rows affected by a SQL statement, the greater is the performance gain with bulk binding.

Using Bulk Binding

Keywords to support bulk binding:

- **The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine.**

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

- **The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections before returning them to the PL/SQL engine.**

```
... BULK COLLECT INTO
      collection_name[,collection_name] ...
```

ORACLE

Using Bulk Binding

Use bulk binds to improve the performance of:

- DML statements that reference collection elements
- `SELECT` statements that reference collection elements
- Cursor `FOR` loops that reference collections and the `RETURNING INTO` clause

Keywords to Support Bulk Binding

The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine. Although the `FORALL` statement contains an iteration scheme, it is not a `FOR` loop.

The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections, before returning them to the PL/SQL engine. This allows you to bind locations into which SQL can return the retrieved values in bulk. Thus, you can use these keywords in the `SELECT INTO`, `FETCH INTO`, and `RETURNING INTO` clauses.

The `SAVE EXCEPTIONS` keyword is optional. However, if some of the DML operations succeed and some fail, you will want to track or report on those that fail. Using the `SAVE EXCEPTIONS` keyword causes failed operations to be stored in a cursor attribute called `%BULK_EXCEPTIONS`, which is a collection of records indicating the bulk DML iteration number and corresponding error code.

Bulk Binding FORALL: Example

```
CREATE PROCEDURE raise_salary(percent NUMBER) IS
  TYPE numlist IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  id  numlist;
BEGIN
 id(1) := 100; id(2) := 102;
 id(3) := 104; id(4) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN id.FIRST .. id.LAST
    UPDATE employees
      SET salary = (1 + percent/100) * salary
      WHERE manager_id = id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

ORACLE

Bulk Binding FORALL: Example

In the example in the slide, the PL/SQL block increases the salary for employees whose manager's ID is 100, 102, 104, or 110. It uses the FORALL keyword to bulk bind the collection. Without bulk binding, the PL/SQL block would have sent a SQL statement to the SQL engine for each employee that is updated. If there are many employees to update, then the large number of context switches between the PL/SQL engine and the SQL engine can affect performance. However, the FORALL keyword bulk binds the collection to improve performance.

Note: A looping construct is no longer required when using this feature.

To manage exceptions and have the bulk bind complete despite errors, add the SAVE EXCEPTIONS keyword to your FORALL statement after the bounds, before the DML statement. Use the new cursor attribute %BULK_EXCEPTIONS, which stores a collection of records with two fields:

- %BULK_EXCEPTIONS(i).ERROR_INDEX holds the "iteration" of the statement during which the exception was raised.
- %BULK_EXCEPTIONS(i).ERROR_CODE holds the corresponding error code.

Values stored in %BULK_EXCEPTIONS refer to the most recently executed FORALL statement. Its subscripts range from 1 to %BULK_EXCEPTIONS.COUNT.

Bulk Binding FORALL: Example (continued)

An Additional Cursor Attribute for DML Operations

Another cursor attribute added to support bulk operations is %BULK_ROWCOUNT. The %BULK_ROWCOUNT attribute is a composite structure designed for use with the FORALL statement. This attribute acts like an index-by table. Its *i*th element stores the number of rows processed by the *i*th execution of an UPDATE or DELETE statement. If the *i*th execution affects no rows, then %BULK_ROWCOUNT(i) returns zero.

Here is an example:

```
CREATE TABLE num_table (n NUMBER);
DECLARE
  TYPE NumList IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  nums NumList;
BEGIN
  nums(1) := 1;
  nums(2) := 3;
  nums(3) := 5;
  nums(4) := 7;
  nums(5) := 11;
  FORALL i IN nums.FIRST .. nums.LAST
    INSERT INTO num_table (n) VALUES (nums(i));
  FOR i IN nums.FIRST .. nums.LAST
  LOOP
    dbms_output.put_line('Inserted ' ||
      SQL%BULK_ROWCOUNT(i) || ' row(s)'
      || ' on iteration ' || i);
  END LOOP;
END;
/
DROP TABLE num_table;
```

The following results are produced by this example:

```
Inserted 1 row(s) on iteration 1
Inserted 1 row(s) on iteration 2
Inserted 1 row(s) on iteration 3
Inserted 1 row(s) on iteration 4
Inserted 1 row(s) on iteration 5

PL/SQL procedure successfully completed.
```

Using BULK COLLECT INTO with Queries

The SELECT statement has been enhanced to support the BULK COLLECT INTO syntax.

Example:

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  TYPE dept_tabtype IS
    TABLE OF departments%ROWTYPE;
  depts dept_tabtype;
BEGIN
  SELECT * BULK COLLECT INTO depts
  FROM departments
  WHERE location_id = loc;
  FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_id
      ||' '||depts(i).department_name);
  END LOOP;
END;
```

ORACLE

Using BULK COLLECT INTO with Queries

In Oracle Database 10*g*, when using a SELECT statement in PL/SQL, you can use the bulk collection syntax shown in the example in the slide. Thus, you can quickly obtain a set of rows without using a cursor mechanism.

The example reads all the department rows for a specified region into a PL/SQL table, whose contents are displayed with the FOR loop that follows the SELECT statement.

Using BULK COLLECT INTO with Cursors

The FETCH statement has been enhanced to support the BULK COLLECT INTO syntax.

Example:

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  CURSOR dept_csr IS SELECT * FROM departments
                     WHERE location_id = loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts;
  CLOSE dept_csr;
 FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_id
      ||' '||depts(i).department_name);
  END LOOP;
END;
```

ORACLE

Using BULK COLLECT INTO with Cursors

In Oracle Database 10*g*, when using cursors in PL/SQL, you can use a form of the FETCH statement that supports the bulk collection syntax shown in the example in the slide.

This example shows how BULK COLLECT INTO can be used with cursors.

You can also add a LIMIT clause to control the number of rows fetched in each operation. The code example in the slide could be modified as follows:

```
CREATE PROCEDURE get_departments(loc NUMBER,
  nrows NUMBER) IS
  CURSOR dept_csr IS SELECT * FROM departments
                     WHERE location_id = loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts LIMIT nrows;
  CLOSE dept_csr;
  DBMS_OUTPUT.PUT_LINE(depts.COUNT||' rows read');
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

Example:

```
CREATE PROCEDURE raise_salary(rate NUMBER) IS
   TYPE emplist IS TABLE OF NUMBER;
   TYPE numlist IS TABLE OF employees.salary%TYPE
     INDEX BY BINARY_INTEGER;
   emp_ids  emplist := emplist(100,101,102,104);
   new_sals numlist;
BEGIN
  FORALL i IN emp_ids.FIRST .. emp_ids.LAST
    UPDATE employees
      SET commission_pct = rate * salary
    WHERE employee_id = emp_ids(i)
    RETURNING salary BULK COLLECT INTO new_sals;
  FOR i IN 1 .. new_sals.COUNT LOOP ...
END;
```

ORACLE

Using BULK COLLECT INTO with a RETURNING Clause

Bulk binds can be used to improve the performance of FOR loops that reference collections and return DML. If you have, or plan to have, PL/SQL code that does this, then you can use the FORALL keyword along with the RETURNING and BULK COLLECT INTO keywords to improve performance.

In the example shown in the slide, the salary information is retrieved from the EMPLOYEES table and collected into the new_sals array. The new_sals collection is returned in bulk to the PL/SQL engine.

The example in the slide shows an incomplete FOR loop that is used to iterate through the new salary data received from the UPDATE operation and then process the results.

Using the NOCOPY Hint

The NOCOPY hint:

- **Is a request to the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value**
- **Enhances performance by reducing overhead when passing parameters**

```
DECLARE
  TYPE emptabtype IS TABLE OF employees%ROWTYPE;
  emp_tab emptabtype;
  PROCEDURE populate(tab IN OUT NOCOPY emptabtype)
  IS BEGIN ... END;
BEGIN
  populate(emp_tab);
END;
/
```

ORACLE

Using the NOCOPY Hint

Note that PL/SQL subprograms support three parameter-passing modes: IN, OUT, and IN OUT. By default:

- The IN parameter is passed by reference. A pointer to the IN actual parameter is passed to the corresponding formal parameter. So both parameters reference the same memory location, which holds the value of the actual parameter.
- The OUT and IN OUT parameters are passed by value. The value of the OUT or IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

Copying parameters that represent large data structures (such as collections, records, and instances of object types) with OUT and IN OUT parameters slows down execution and uses up memory. To prevent this overhead, you can specify the NOCOPY hint, which enables the PL/SQL compiler to pass OUT and IN OUT parameters by reference.

The slide shows an example of declaring an IN OUT parameter with the NOCOPY hint.

Effects of the NOCOPY Hint

- **If the subprogram exits with an exception that is not handled:**
  - **You cannot rely on the values of the actual parameters passed to a NOCOPY parameter**
  - **Any incomplete modifications are not "rolled back"**
- **The remote procedure call (RPC) protocol enables you to pass parameters only by value.**

ORACLE

Effects of the NOCOPY Hint

As a trade-off for better performance, the NOCOPY hint enables you to trade well-defined exception semantics for better performance. Its use affects exception handling in the following ways:

- Because NOCOPY is a hint, not a directive, the compiler can pass NOCOPY parameters to a subprogram by value or by reference. So, if the subprogram exits with an unhandled exception, you cannot rely on the values of the NOCOPY actual parameters.
- By default, if a subprogram exits with an unhandled exception, the values assigned to its OUT and IN OUT formal parameters are not copied to the corresponding actual parameters, and changes appear to roll back. However, when you specify NOCOPY, assignments to the formal parameters immediately affect the actual parameters as well. So, if the subprogram exits with an unhandled exception, the (possibly unfinished) changes are not "rolled back."
- Currently, the RPC protocol enables you to pass parameters only by value. So exception semantics can change without notification when you partition applications. For example, if you move a local procedure with NOCOPY parameters to a remote site, those parameters are no longer passed by reference.

NOCOPY Hint Can Be Ignored

The NOCOPY hint has no effect if:

- **The actual parameter:**
  - **Is an element of an index-by table**
  - **Is constrained (for example, by scale or NOT NULL)**
  - **And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ**
  - **Requires an implicit data type conversion**
- **The subprogram is involved in an external or remote procedure call**

ORACLE

NOCOPY Hint Can Be Ignored

In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method (with no error generated):

- The actual parameter is an element of an index-by table. This restriction does not apply to entire index-by tables.
- The actual parameter is constrained (by scale or NOT NULL). This restriction does not extend to constrained elements or attributes. Also, it does not apply to size-constrained character strings.
- The actual and formal parameters are records; one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records; the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit data type conversion.
- The subprogram is involved in an external or remote procedure call.

PARALLEL_ENABLE Hint

The PARALLEL_ENABLE hint:

- Can be used in functions as an optimization hint

```
CREATE OR REPLACE FUNCTION f2 (p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p1 * 2;
END f2;
```

- Indicates that a function can be used in a parallelized query or parallelized DML statement

ORACLE

PARALLEL_ENABLE Hint

The PARALLEL_ENABLE keyword can be used in the syntax for declaring a function. It is an optimization hint that indicates that the function can be used in a parallelized query or parallelized DML statement. Oracle's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement that is run in parallel can 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.

For DML statements, prior to Oracle8*i*, the parallelization optimization looked to see whether 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 whether they were actually pure enough; parallelized execution might occur even though a PRAGMA cannot be specified on these functions.

The PARALLEL_ENABLE keyword is placed after the return value type in the declaration of the function, as shown in the example in the slide.

Note: The function should not use session state, such as package variables, because those variables may not be shared among the parallel execution servers.

Summary

In this lesson, you should have learned how to:

- **Create standardized constants and exceptions using packages**
- **Develop and invoke local subprograms**
- **Control the run-time privileges of a subprogram by setting the `AUTHID` directive**
- **Execute autonomous transactions**
- **Use the `RETURNING` clause with DML statements, and bulk binding collections with the `FORALL` and `BULK COLLECT INTO` clauses**
- **Pass parameters by reference using a `NOCOPY` hint**
- **Enable optimization with `PARALLEL ENABLE` hints**

ORACLE

Summary

The lesson provides insights into managing your PL/SQL code by defining constants and exceptions in a package specification. This enables a high degree of reuse and standardization of code.

Local subprograms can be used to simplify and modularize a block of code where the subprogram functionality is repeatedly used in the local block.

The run-time security privileges of a subprogram can be altered by using definer's or invoker's rights.

Autonomous transactions can be executed without affecting an existing main transaction.

You should understand how to obtain performance gains by using the `NOCOPY` hint, bulk binding and the `RETURNING` clauses in SQL statements, and the `PARALLEL_ENABLE` hint for optimization of functions.

Practice 7: Overview

This practice covers the following topics:

- **Creating a package that uses bulk fetch operations**
- **Creating a local subprogram to perform an autonomous transaction to audit a business operation**
- **Testing `AUTHID` functionality**

ORACLE

Practice 7: Overview

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table.

You add an `add_employee` procedure to the package that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the `add_employee` procedure is called, whether it successfully adds a record or not.

Finally, you make the package use `AUTHID` of `CURRENT_USER` and test the behavior with any other student. You test the code first with definer's rights and then with invoker's rights.

Practice 7

1. Update EMP_PKG with a new procedure to query employees in a specified department.
   a. In the specification, declare a get_employees procedure, with its parameter called dept_id based on the employees.department_id column type. Define an index-by PL/SQL type as a TABLE OF EMPLOYEES%ROWTYPE.
   b. In the body of the package, define a private variable called emp_table based on the type defined in the specification to hold employee records. Implement the get_employees procedure to bulk fetch the data into the table.
   c. Create a new procedure in the specification and body, called show_employees, that does not take arguments and displays the contents of the private PL/SQL table variable (if any data exists).
      Hint: Use the print_employee procedure.
   d. Invoke the emp_pkg.get_employees procedure for department 30, and then invoke emp_pkg.show_employees. Repeat this for department 60.
2. Your manager wants to keep a log whenever the add_employee procedure in the package is invoked to insert a new employee into the EMPLOYEES table.
   a. First, load and execute the E:\labs\PLPU\labs\lab_07_02_a.sql script to create a log table called LOG_NEWEMP, and a sequence called log_newemp_seq.
   b. In the package body, modify the add_employee procedure, which performs the actual INSERT operation, to have a local procedure called audit_newemp. The audit_newemp procedure must use an autonomous transaction to insert a log record into the LOG_NEWEMP table. Store the USER, the current time, and the new employee name in the log table row. Use log_newemp_seq to set the entry_id column.
      Note: Remember to perform a COMMIT operation in a procedure with an autonomous transaction.
   c. Modify the add_employee procedure to invoke audit_emp before it performs the insert operation.
   d. Invoke the add_employee procedure for these new employees: Max Smart in department 20 and Clark Kent in department 10. What happens?
   e. Query the two EMPLOYEES records added, and the records in LOG_NEWEMP table. How many log records are present?
   f. Execute a ROLLBACK statement to undo the insert operations that have not been committed. Use the same queries from Exercise 2e: the first to check whether the employee rows for Smart and Kent have been removed, and the second to check the log records in the LOG_NEWEMP table. How many log records are present? Why?

Practice 7 (continued)

If you have time, complete the following exercise:

3. Modify the `EMP_PKG` package to use `AUTHID` of `CURRENT_USER` and test the behavior with any other student.

   Note: Verify whether the `LOG_NEWEMP` table exists from Exercise 2 in this practice.

   a. Grant the `EXECUTE` privilege on your `EMP_PKG` package to another student.
   b. Ask the other student to invoke your `add_employee` procedure to insert employee `Jaco Pastorius` in department 10. Remember to prefix the package name with the owner of the package. The call should operate with definer's rights.
   c. Now, execute a query of the employees in department 10. In which user's employee table did the new record get inserted?
   d. Modify your package `EMP_PKG` specification to use an `AUTHID CURRENT_USER`. Compile the body of `EMP_PKG`.
   e. Ask the same student to execute the `add_employee` procedure again, to add employee `Joe Zawinal` in department 10.
   f. Query your employees in department 10. In which table was the new employee added?
   g. Write a query to display the records added in the `LOG_NEWEMP` tables. Ask the other student to query his or her own copy of the table.