
ORACLE
ACADEMY

Database Programming with PL/SQL

12-1
Using Dynamic SQL

ORACLE ACADEMY

Objectives

This lesson covers the following objectives:

- Recall the stages through which all SQL statements pass
- Describe the reasons for using dynamic SQL to create a SQL statement
- List four PL/SQL statements supporting Native Dynamic SQL
- Describe the benefits of `EXECUTE IMMEDIATE` over `DBMS_SQL` for Dynamic SQL

Purpose

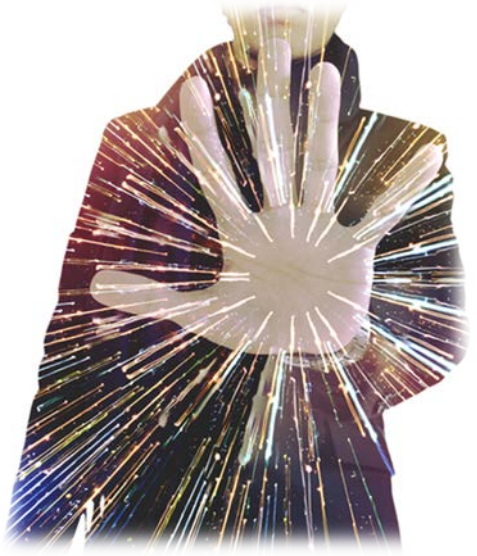
- In this lesson, you learn to construct and execute SQL statements dynamically—in other words, at run time using the Native Dynamic SQL statements in PL/SQL.
- Dynamically executing SQL and PL/SQL code extends the capabilities of PL/SQL beyond query and transactional operations.
- The lesson also compares Native Dynamic SQL to the DBMS_SQL package, which provides similar capabilities.

Execution Flow of SQL

- All SQL statements in the database go through various stages:
  - Parse: Pre-execution “is this possible?” checks syntax, object existence, privileges, and so on
  - Bind: Getting the actual values of any variables referenced in the statement
  - Execute: The statement is executed.
  - Fetch: Results are returned to the user.
- Some stages might not be relevant for all statements; for example, the fetch phase is applicable to queries but not DML.

Execution Flow of SQL in PL/SQL Subprograms

- When a SQL statement is included in a PL/SQL subprogram, the parse and bind phases are normally done at compile time, that is, when the procedure, function, or package body is CREATEd.
- What if the text of the SQL statement is not known when the procedure is created?



Execution Flow of SQL in PL/SQL Subprograms

- How could the Oracle server parse it?
- It couldn't.
- For example, suppose you want to DROP a table, but the user enters the table name at execution time:

```
CREATE PROCEDURE drop_any_table(p_table_name
VARCHAR2)
IS BEGIN
  DROP TABLE p_table_name; -- cannot be parsed
END;
```

Dynamic SQL

You use dynamic SQL to create a SQL statement whose text is not completely known in advance. Dynamic SQL:

- Is constructed and stored as a character string within a subprogram.
- Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables).
- Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL.

Native Dynamic SQL

- PL/SQL does not support DDL statements written directly in a program.
- Native Dynamic SQL (NDS) allows you to work around this by constructing and storing SQL as a character string within a subprogram.
- NDS:
  - Provides native support for Dynamic SQL directly in the PL/SQL language.
  - Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL.

Native Dynamic SQL

NDS:

- Is executed with Native Dynamic SQL statements (EXECUTE IMMEDIATE) or the DBMS_SQL package.
- Provides the ability to execute SQL statements whose structure is unknown until execution time.
- Can also use the OPEN-FOR, FETCH, and CLOSE PL/SQL statements.

Using the EXECUTE IMMEDIATE Statement

- Use the EXECUTE IMMEDIATE statement for NDS in PL/SQL anonymous blocks or subprograms:

```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
     [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
     [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.
- USING holds all bind arguments.
- The default parameter mode is IN, if not specified.

Using the EXECUTE IMMEDIATE Statement

```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
     [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
     [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- `dynamic_string` is a character variable or literal containing the text of a SQL statement.
- `define_variable` is a PL/SQL variable that stores a selected column value.
- *`record`* is a user-defined or %ROWTYPE record that stores a selected row.

Using the EXECUTE IMMEDIATE Statement

```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
     [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
     [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- *bind_argument* is an expression whose value is passed to the dynamic SQL statement at execution time.
- USING clause holds all bind arguments.
- The default parameter mode is IN.

Example 1: Dynamic SQL with a DDL Statement

- Constructing the dynamic statement in-line:

```
CREATE PROCEDURE drop_any_table(p_table_name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE ' || p_table_name;
END;
```

- Constructing the dynamic statement in a variable:

```
CREATE PROCEDURE drop_any_table(p_table_name VARCHAR2) IS
  v_dynamic_stmt  VARCHAR2(50);
BEGIN
  v_dynamic_stmt := 'DROP TABLE ' || p_table_name;
  EXECUTE IMMEDIATE v_dynamic_stmt;
END;
```

```
BEGIN  drop_any_table('EMPLOYEE_NAMES');   END;
```

Example 2: Dynamic SQL with a DML Statement

- Deleting all the rows from any table and returning a count:

```
CREATE FUNCTION del_rows(p_table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM ' || p_table_name;
  RETURN SQL%ROWCOUNT;
END;
```

- Invoking the function:

```
DECLARE
  v_count  NUMBER;
BEGIN
  v_count := del_rows('EMPLOYEE_NAMES');
  DBMS_OUTPUT.PUT_LINE(v_count || ' rows deleted.');
END;
```

Example 3: Dynamic SQL with a DML Statement

- Here is an example of inserting a row into a table with two columns and invoking the procedure.
- Note the use of escape single quotes.

```
CREATE PROCEDURE add_row(p_table_name VARCHAR2,
   p_id NUMBER, p_name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO ' || p_table_name ||
        'VALUES(' || p_id || ', ''' || p_name || ''')';
END;
```

```
BEGIN
  add_row('EMPLOYEE_NAMES', 250, 'Chang');
END;
```

Example 4: Using Native Dynamic SQL to Recompile PL/SQL Code

You can recompile PL/SQL objects without recreating them by using the following ALTER statements:

```
ALTER PROCEDURE procedure-name COMPILE;
ALTER FUNCTION  function-name  COMPILE;
ALTER PACKAGE   package_name   COMPILE SPECIFICATION;
ALTER PACKAGE   package-name   COMPILE BODY;
```

Example 4: Using Native Dynamic SQL to Recompile PL/SQL Code

- This example creates a procedure that recompiles a PL/SQL object whose name and type is entered at run time.

```
CREATE PROCEDURE compile_plsql
 (p_name VARCHAR2,p_type VARCHAR2,p_options VARCHAR2 := NULL) IS
  v_stmt VARCHAR2(200);
BEGIN
  v_stmt := 'ALTER ' || p_type || ' ' || p_name || ' COMPILE'
            || ' ' || p_options;
  EXECUTE IMMEDIATE v_stmt;
END;
```

```
BEGIN   compile_plsql('MYPACK','PACKAGE','BODY');  END;
```

Using the DBMS_SQL Package

Some of the procedures and functions of the DBMS_SQL package are:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

Using DBMS_SQL with a DML Statement

- Example of deleting rows:

```
CREATE OR REPLACE FUNCTION del_rows
 (p_table_name VARCHAR2) RETURN NUMBER IS
  v_csr_id      INTEGER;
  v_rows_del    NUMBER;
BEGIN
  v_csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_csr_id,
    'DELETE FROM ' || p_table_name, DBMS_SQL.NATIVE);
  v_rows_del := DBMS_SQL.EXECUTE(v_csr_id);
  DBMS_SQL.CLOSE_CURSOR(v_csr_id);
  RETURN v_rows_del;
END;
```

- Compare this with the `del_rows` function earlier in this lesson.
- They are functionally identical, but which is simpler?

Using DBMS_SQL with a Parameterized DML Statement

- Again, compare this with the `add_row` procedure earlier in this lesson.
- Which would you rather write?

```
CREATE PROCEDURE add_row (p_table_name VARCHAR2,
  p_id NUMBER, p_name VARCHAR2) IS
  v_csr_id     INTEGER;
  v_stmt       VARCHAR2(200);
  v_rows_added NUMBER;
BEGIN
  v_stmt := 'INSERT INTO ' || p_table_name ||
         ' VALUES(' || p_id || ', ''' || p_name || ''')';
  v_csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_csr_id, v_stmt, DBMS_SQL.NATIVE);
  v_rows_added := DBMS_SQL.EXECUTE(v_csr_id);
  DBMS_SQL.CLOSE_CURSOR(v_csr_id);
END;
```

Comparison of Native Dynamic SQL and the DBMS_SQL Package

Native Dynamic SQL:

- Is easier to use than DBMS_SQL
- Requires less code than DBMS_SQL
- Often executes faster than DBMS_SQL because there are fewer statements to execute.

Terminology

Key terms used in this lesson included:

- Native Dynamic SQL
- `EXECUTE IMMEDIATE`

Summary

In this lesson, you should have learned how to:

- Recall the stages through which all SQL statements pass
- Describe the reasons for using dynamic SQL to create a SQL statement
- List four PL/SQL statements supporting Native Dynamic SQL
- Describe the benefits of EXECUTE IMMEDIATE over DBMS_SQL for Dynamic SQL


ORACLE
ACADEMY