7

PL/SQL Dynamic SQL

Dynamic SQL is a programming methodology for generating and running SQL statements at run time. It is useful when writing general-purpose and flexible programs like ad hoc query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compilation time the full text of a SQL statement or the number or data types of its input and output variables.

PL/SQL provides two ways to write dynamic SQL:

- Native dynamic SQL, a PL/SQL language (that is, native) feature for building and running dynamic SQL statements
- `DBMS_SQL` package, an API for building, running, and describing dynamic SQL statements

Native dynamic SQL code is easier to read and write than equivalent code that uses the `DBMS_SQL` package, and runs noticeably faster (especially when it can be optimized by the compiler). However, to write native dynamic SQL code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement. If you do not know this information at compile time, you must use the `DBMS_SQL` package.

When you need both the `DBMS_SQL` package and native dynamic SQL, you can switch between them, using the "DBMS_SQL.TO_REFCURSOR Function" on page 7-7 and "DBMS_SQL.TO_CURSOR_NUMBER Function" on page 7-8.

Topics

- When You Need Dynamic SQL
- Native Dynamic SQL
- DBMS_SQL Package
- SQL Injection

When You Need Dynamic SQL

In PL/SQL, you need dynamic SQL to run:

- SQL whose text is unknown at compile time

  For example, a `SELECT` statement that includes an identifier that is unknown at compile time (such as a table name) or a `WHERE` clause in which the number of subclauses is unknown at compile time.

- SQL that is not supported as static SQL

That is, any SQL construct not included in "Description of Static SQL" on page 6-1.

If you do not need dynamic SQL, use static SQL, which has these advantages:

- Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects.
- Successful compilation creates schema object dependencies.

  For information about schema object dependencies, see *Oracle Database Advanced Application Developer's Guide*.

For information about using static SQL statements with PL/SQL, see Chapter 6, "PL/SQL Static SQL."

Native Dynamic SQL

Native dynamic SQL processes most dynamic SQL statements with the `EXECUTE IMMEDIATE` statement.

If the dynamic SQL statement is a `SELECT` statement that returns multiple rows, native dynamic SQL gives you these choices:

- Use the `EXECUTE IMMEDIATE` statement with the `BULK COLLECT INTO` clause.
- Use the `OPEN FOR`, `FETCH`, and `CLOSE` statements.

The SQL cursor attributes work the same way after native dynamic SQL `INSERT`, `UPDATE`, `DELETE`, `MERGE`, and single-row `SELECT` statements as they do for their static SQL counterparts. For more information about SQL cursor attributes, see "Cursors" on page 6-5.

Topics

- EXECUTE IMMEDIATE Statement
- OPEN FOR, FETCH, and CLOSE Statements
- Repeated Placeholder Names in Dynamic SQL Statements

EXECUTE IMMEDIATE Statement

The `EXECUTE IMMEDIATE` statement is the means by which native dynamic SQL processes most dynamic SQL statements.

If the dynamic SQL statement is **self-contained** (that is, if it has no placeholders for bind variables and the only result that it can possibly return is an error), then the `EXECUTE IMMEDIATE` statement needs no clauses.

If the dynamic SQL statement includes placeholders for bind variables, each placeholder must have a corresponding bind variable in the appropriate clause of the `EXECUTE IMMEDIATE` statement, as follows:

- If the dynamic SQL statement is a `SELECT` statement that can return at most one row, put out-bind variables (defines) in the `INTO` clause and in-bind variables in the `USING` clause.
- If the dynamic SQL statement is a `SELECT` statement that can return multiple rows, put out-bind variables (defines) in the `BULK COLLECT INTO` clause and in-bind variables in the `USING` clause.
- If the dynamic SQL statement is a DML statement without a `RETURNING INTO` clause, other than `SELECT`, put all bind variables in the `USING` clause.

- If the dynamic SQL statement is a DML statement with a RETURNING INTO clause, put in-bind variables in the USING clause and out-bind variables in the RETURNING INTO clause.
- If the dynamic SQL statement is an anonymous PL/SQL block or a CALL statement, put all bind variables in the USING clause.

  If the dynamic SQL statement invokes a subprogram, ensure that:

  – Every bind variable that corresponds to a placeholder for a subprogram parameter has the same parameter mode as that subprogram parameter (as in Example 7–1) and a data type that is compatible with that of the subprogram parameter. (For information about compatible data types, see "Formal and Actual Subprogram Parameters" on page 8-9.)

  – No bind variable has a data type that SQL does not support (such as BOOLEAN in Example 7–2).

The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 7–3.

For syntax details of the EXECUTE IMMEDIATE statement, see "EXECUTE IMMEDIATE Statement" on page 13-52.

Example 7–1 Invoking Subprogram from Dynamic PL/SQL Block

```
-- Subprogram that dynamic PL/SQL block invokes:
CREATE OR REPLACE PROCEDURE create_dept (
  deptid IN OUT NUMBER,
  dname  IN     VARCHAR2,
  mgrid  IN     NUMBER,
  locid  IN     NUMBER
) AS
BEGIN
  deptid := departments_seq.NEXTVAL;

  INSERT INTO departments (
    department_id,
    department_name,
    manager_id,
    location_id
  )
  VALUES (deptid, dname, mgrid, locid);
END;
/
DECLARE
  plsql_block VARCHAR2(500);
  new_deptid  NUMBER(4);
  new_dname   VARCHAR2(30) := 'Advertising';
  new_mgrid   NUMBER(6)    := 200;
  new_locid   NUMBER(4)    := 1700;
BEGIN
 -- Dynamic PL/SQL block invokes subprogram:
  plsql_block := 'BEGIN create_dept(:a, :b, :c, :d); END;';

 /* Specify bind variables in USING clause.
    Specify mode for first parameter.
    Modes of other parameters are correct by default. */

  EXECUTE IMMEDIATE plsql_block
    USING IN OUT new_deptid, new_dname, new_mgrid, new_locid;
END;
```

```
/
```

Example 7–2 Unsupported Data Type in Native Dynamic SQL

```
DECLARE
  dyn_stmt VARCHAR2(200);
  b1       BOOLEAN;

  FUNCTION f (x INTEGER)
    RETURN BOOLEAN
  AS
  BEGIN
    NULL;
  END f;

BEGIN
  dyn_stmt := 'BEGIN :b := f(5); END;';
  EXECUTE IMMEDIATE dyn_stmt USING OUT b1;
END;
/
```

Result:

```
  EXECUTE IMMEDIATE dyn_stmt USING OUT b1;
                                      *
ERROR at line 15:
ORA-06550: line 15, column 40:
PLS-00457: expressions have to be of SQL types
ORA-06550: line 15, column 3:
PL/SQL: Statement ignored
```

Example 7–3 Uninitialized Variable Represents NULL in USING Clause

```
CREATE TABLE employees_temp AS SELECT * FROM EMPLOYEES;

DECLARE
  a_null  CHAR(1);  -- Set to NULL automatically at run time
BEGIN
  EXECUTE IMMEDIATE 'UPDATE employees_temp SET commission_pct = :x'
    USING a_null;
END;
/
```

OPEN FOR, FETCH, and CLOSE Statements

If the dynamic SQL statement represents a `SELECT` statement that returns multiple rows, you can process it with native dynamic SQL as follows:

1. Use an `OPEN FOR` statement to associate a cursor variable with the dynamic SQL statement. In the `USING` clause of the `OPEN FOR` statement, specify a bind variable for each placeholder in the dynamic SQL statement.

   The `USING` clause cannot contain the literal `NULL`. To work around this restriction, use an uninitialized variable where you want to use `NULL`, as in Example 7–3.

   For syntax details, see "OPEN FOR Statement" on page 13-104.

2. Use the `FETCH` statement to retrieve result set rows one at a time, several at a time, or all at once.

   For syntax details, see "FETCH Statement" on page 13-71.

3. Use the CLOSE statement to close the cursor variable.

   For syntax details, see "CLOSE Statement" on page 13-23.

Example 7–4 lists all employees who are managers, retrieving result set rows one at a time.

Example 7–4 Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements

```
DECLARE
  TYPE EmpCurTyp  IS REF CURSOR;
  v_emp_cursor    EmpCurTyp;
  emp_record      employees%ROWTYPE;
  v_stmt_str      VARCHAR2(200);
  v_e_job         employees.job%TYPE;
BEGIN
  -- Dynamic SQL statement with placeholder:
  v_stmt_str := 'SELECT * FROM employees WHERE job_id = :j';

  -- Open cursor & specify bind variable in USING clause:
  OPEN v_emp_cursor FOR v_stmt_str USING 'MANAGER';

  -- Fetch rows from result set one at a time:
  LOOP
    FETCH v_emp_cursor INTO emp_record;
    EXIT WHEN v_emp_cursor%NOTFOUND;
  END LOOP;

  -- Close cursor:
  CLOSE v_emp_cursor;
END;
/
```

Repeated Placeholder Names in Dynamic SQL Statements

If you repeat placeholder names in dynamic SQL statements, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement.

Topics

- Dynamic SQL Statement is Not Anonymous Block or CALL Statement
- Dynamic SQL Statement is Anonymous Block or CALL Statement

Dynamic SQL Statement is Not Anonymous Block or CALL Statement

If the dynamic SQL statement does not represent an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is insignificant. Placeholders are associated with bind variables in the USING clause by position, not by name.

For example, in this dynamic SQL statement, the repetition of the name :x is insignificant:

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

In the corresponding USING clause, you must supply four bind variables. They can be different; for example:

```
EXECUTE IMMEDIATE sql_stmt USING a, b, c, d;
```

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

```
INSERT INTO payroll VALUES (a, b, c, d)
```

To associate the same bind variable with each occurrence of :x, you must repeat that bind variable; for example:

```
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

```
INSERT INTO payroll VALUES (a, a, b, a)
```

Dynamic SQL Statement is Anonymous Block or CALL Statement

If the dynamic SQL statement represents an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is significant. Each unique placeholder name must have a corresponding bind variable in the USING clause. If you repeat a placeholder name, you need not repeat its corresponding bind variable. All references to that placeholder name correspond to one bind variable in the USING clause.

In Example 7–5, all references to the first unique placeholder name, :x, are associated with the first bind variable in the USING clause, a, and the second unique placeholder name, :y, is associated with the second bind variable in the USING clause, b.

Example 7–5 Repeated Placeholder Names in Dynamic PL/SQL Block

```
CREATE PROCEDURE calc_stats (
  w NUMBER,
  x NUMBER,
  y NUMBER,
  z NUMBER )
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(w + x + y + z);
END;
/
DECLARE
  a NUMBER := 4;
  b NUMBER := 7;
  plsql_block VARCHAR2(100);
BEGIN
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';
  EXECUTE IMMEDIATE plsql_block USING a, b;  -- calc_stats(a, a, b, a)
END;
/
```

DBMS_SQL Package

The DBMS_SQL package defines an entity called a SQL cursor number. Because the SQL cursor number is a PL/SQL integer, you can pass it across call boundaries and store it.

You must use the DBMS_SQL package to run a dynamic SQL statement when you do not know either of these until run time:

- SELECT list
- What placeholders in a SELECT or DML statement must be bound

In these situations, you must use native dynamic SQL instead of the DBMS_SQL package:

- The dynamic SQL statement retrieves rows into records.

- You want to use the SQL cursor attribute %FOUND, %ISOPEN, %NOTFOUND, or %ROWCOUNT after issuing a dynamic SQL statement that is an INSERT, UPDATE, DELETE, MERGE, or single-row SELECT statement.

For information about native dynamic SQL, see "Native Dynamic SQL" on page 7-2.

When you need both the DBMS_SQL package and native dynamic SQL, you can switch between them, using:

- DBMS_SQL.TO_REFCURSOR Function
- DBMS_SQL.TO_CURSOR_NUMBER Function

Note: You can invoke DBMS_SQL subprograms remotely.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_SQL package, including instructions for running a dynamic SQL statement that has an unknown number of input or output variables ("Method 4")

DBMS_SQL.TO_REFCURSOR Function

The DBMS_SQL.TO_REFCURSOR function converts a SQL cursor number to a weak cursor variable, which you can use in native dynamic SQL statements.

Before passing a SQL cursor number to the DBMS_SQL.TO_REFCURSOR function, you must OPEN, PARSE, and EXECUTE it (otherwise an error occurs).

After you convert a SQL cursor number to a REF CURSOR variable, DBMS_SQL operations can access it only as the REF CURSOR variable, not as the SQL cursor number. For example, using the DBMS_SQL.IS_OPEN function to see if a converted SQL cursor number is still open causes an error.

Example 7–6 uses the DBMS_SQL.TO_REFCURSOR function to switch from the DBMS_SQL package to native dynamic SQL.

Example 7–6 Switching from DBMS_SQL Package to Native Dynamic SQL

```
CREATE OR REPLACE TYPE vc_array IS TABLE OF VARCHAR2(200);
/
CREATE OR REPLACE TYPE numlist IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE do_query_1 (
  placeholder vc_array,
  bindvars vc_array,
  sql_stmt VARCHAR2
)
IS
  TYPE curtype IS REF CURSOR;
  src_cur     curtype;
  curid       NUMBER;
  bindnames   vc_array;
  empnos      numlist;
  depts       numlist;
  ret         NUMBER;
  isopen      BOOLEAN;
BEGIN
  -- Open SQL cursor number:
  curid := DBMS_SQL.OPEN_CURSOR;
```

```
  -- Parse SQL cursor number:
  DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);

  bindnames := placeholder;

  -- Bind variables:
  FOR i IN 1 .. bindnames.COUNT LOOP
    DBMS_SQL.BIND_VARIABLE(curid, bindnames(i), bindvars(i));
  END LOOP;

  -- Run SQL cursor number:
  ret := DBMS_SQL.EXECUTE(curid);

  -- Switch from DBMS_SQL to native dynamic SQL:
  src_cur := DBMS_SQL.TO_REFCURSOR(curid);
  FETCH src_cur BULK COLLECT INTO empnos, depts;

  -- This would cause an error because curid was converted to a REF CURSOR:
  -- isopen := DBMS_SQL.IS_OPEN(curid);

  CLOSE src_cur;
END;
/
```

DBMS_SQL.TO_CURSOR_NUMBER Function

The DBMS_SQL.TO_CURSOR_NUMBER function converts a REF CURSOR variable (either strong or weak) to a SQL cursor number, which you can pass to DBMS_SQL subprograms.

Before passing a REF CURSOR variable to the DBMS_SQL.TO_CURSOR_NUMBER function, you must OPEN it.

After you convert a REF CURSOR variable to a SQL cursor number, native dynamic SQL operations cannot access it.

Example 7–7 uses the DBMS_SQL.TO_CURSOR_NUMBER function to switch from native dynamic SQL to the DBMS_SQL package.

Example 7–7 Switching from Native Dynamic SQL to DBMS_SQL Package

```
CREATE OR REPLACE PROCEDURE do_query_2 (
  sql_stmt VARCHAR2
)
IS
  TYPE curtype IS REF CURSOR;
  src_cur   curtype;
  curid     NUMBER;
  desctab   DBMS_SQL.DESC_TAB;
  colcnt    NUMBER;
  namevar   VARCHAR2(50);
  numvar    NUMBER;
  datevar   DATE;
  empno     NUMBER := 100;
BEGIN
  -- sql_stmt := SELECT ... FROM employees WHERE employee_id = :b1';

  -- Open REF CURSOR variable:
  OPEN src_cur FOR sql_stmt USING empno;

  -- Switch from native dynamic SQL to DBMS_SQL package:
```

```
  curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
  DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);

  -- Define columns:
  FOR i IN 1 .. colcnt LOOP
    IF desctab(i).col_type = 2 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
    ELSIF desctab(i).col_type = 12 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
      -- statements
    ELSE
      DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 50);
    END IF;
  END LOOP;

  -- Fetch rows with DBMS_SQL package:
  WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
    FOR i IN 1 .. colcnt LOOP
      IF (desctab(i).col_type = 1) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
      ELSIF (desctab(i).col_type = 2) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, numvar);
      ELSIF (desctab(i).col_type = 12) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, datevar);
        -- statements
      END IF;
    END LOOP;
  END LOOP;

  DBMS_SQL.CLOSE_CURSOR(curid);
END;
/
```

SQL Injection

SQL injection maliciously exploits applications that use client-supplied data in SQL statements, thereby gaining unauthorized access to a database to view or manipulate restricted data. This section describes SQL injection vulnerabilities in PL/SQL and explains how to guard against them.

To try the examples in this topic, connect to the `HR` schema and run the statements in Example 7–8.

Example 7–8 Setup for SQL Injection Examples

```
DROP TABLE secret_records;
CREATE TABLE secret_records (
  user_name    VARCHAR2(9),
  service_type VARCHAR2(12),
  value        VARCHAR2(30),
  date_created DATE
);

INSERT INTO secret_records (
  user_name, service_type, value, date_created
)
VALUES ('Andy', 'Waiter', 'Serve dinner at Cafe Pete', SYSDATE);

INSERT INTO secret_records (
  user_name, service_type, value, date_created
```

```
)
VALUES ('Chuck', 'Merger', 'Buy company XYZ', SYSDATE);
```

Topics

- SQL Injection Techniques
- Guarding Against SQL Injection

SQL Injection Techniques

All SQL injection techniques exploit a single vulnerability: String input is not correctly validated and is concatenated into a dynamic SQL statement.

Topics

- Statement Modification
- Statement Injection
- Data Type Conversion

Statement Modification

Statement modification means deliberately altering a dynamic SQL statement so that it runs in a way unintended by the application developer. Typically, the user retrieves unauthorized data by changing the `WHERE` clause of a `SELECT` statement or by inserting a `UNION ALL` clause. The classic example of this technique is bypassing password authentication by making a `WHERE` clause always `TRUE`.

Example 7–9 creates a procedure that is vulnerable to statement modification and then invokes that procedure with and without statement modification. With statement modification, the procedure returns a supposedly secret record.

Example 7–9 Procedure Vulnerable to Statement Modification

Create vulnerable procedure:

```
CREATE OR REPLACE PROCEDURE get_record (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  -- Following SELECT statement is vulnerable to modification
  -- because it uses concatenation to build WHERE clause.
  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || '''';
  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec ;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec );
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;
```

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record('Andy', 'Waiter', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter'
Rec: Serve dinner at Cafe Pete
```

Example of statement modification:

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record(
  'Anybody '' OR service_type=''Merger''--',
  'Anything',
  record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody ' OR
service_type='Merger'--' AND service_type='Anything'
Rec: Buy company XYZ

PL/SQL procedure successfully completed.
```

Statement Injection

Statement injection means that a user appends one or more SQL statements to a dynamic SQL statement. Anonymous PL/SQL blocks are vulnerable to this technique.

Example 7–10 creates a procedure that is vulnerable to statement injection and then invokes that procedure with and without statement injection. With statement injection, the procedure deletes the supposedly secret record exposed in Example 7–9.

Example 7–10 Procedure Vulnerable to Statement Injection

Create vulnerable procedure:

```
CREATE OR REPLACE PROCEDURE p (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2
)
IS
  block1 VARCHAR2(4000);
BEGIN
  -- Following block is vulnerable to statement injection
  -- because it is built by concatenation.
  block1 :=
    'BEGIN
    DBMS_OUTPUT.PUT_LINE(''user_name: ' || user_name || ''');'
    || 'DBMS_OUTPUT.PUT_LINE(''service_type: ' || service_type || ''');
    END;';

  DBMS_OUTPUT.PUT_LINE('Block1: ' || block1);
```

```
  EXECUTE IMMEDIATE block1;
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;

BEGIN
  p('Andy', 'Waiter');
END;
/
```

Result:

```
Block1: BEGIN
         DBMS_OUTPUT.PUT_LINE('user_name: Andy');
         DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
       END;
user_name: Andy
service_type: Waiter
```

SQL*Plus formatting command:

```
COLUMN date_created FORMAT A12;
```

Query:

```
SELECT * FROM secret_records ORDER BY user_name;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      28-APR-10
Chuck     Merger       Buy company XYZ                28-APR-10
```

Example of statement modification:

```
BEGIN
  p('Anybody', 'Anything'');
  DELETE FROM secret_records WHERE service_type=INITCAP(''Merger');
END;
/
```

Result:

```
Block1: BEGIN
       DBMS_OUTPUT.PUT_LINE('user_name: Anybody');
       DBMS_OUTPUT.PUT_LINE('service_type: Anything');
       DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
     END;
user_name: Anybody
service_type: Anything

PL/SQL procedure successfully completed.
```

Query:

```
SELECT * FROM secret_records;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      18-MAR-09

1 row selected.
```

Data Type Conversion

A less known SQL injection technique uses NLS session parameters to modify or inject SQL statements.

A datetime or numeric value that is concatenated into the text of a dynamic SQL statement must be converted to the `VARCHAR2` data type. The conversion can be either implicit (when the value is an operand of the concatentation operator) or explicit (when the value is the argument of the `TO_CHAR` function). This data type conversion depends on the NLS settings of the database session that runs the dynamic SQL statement. The conversion of datetime values uses format models specified in the parameters `NLS_DATE_FORMAT`, `NLS_TIMESTAMP_FORMAT`, or `NLS_TIMESTAMP_TZ_FORMAT`, depending on the particular datetime data type. The conversion of numeric values applies decimal and group separators specified in the parameter `NLS_NUMERIC_CHARACTERS`.

One datetime format model is `"text"`. The *text* is copied into the conversion result. For example, if the value of `NLS_DATE_FORMAT` is `'"Month:" Month'`, then in June, `TO_CHAR(SYSDATE)` returns `'Month: June'`. The datetime format model can be abused as shown in Example 7–11.

Example 7–11 Procedure Vulnerable to SQL Injection Through Data Type Conversion

```
SELECT * FROM secret_records;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATE
--------- ------------ ------------------------------ -----------
Andy      Waiter       Serve dinner at Cafe Pete      28-APR-2010
Chuck     Merger       Buy company XYZ                28-APR-2010
```

Create vulnerable procedure:

```
-- Return records not older than a month

CREATE OR REPLACE PROCEDURE get_recent_record (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  /* Following SELECT statement is vulnerable to modification
     because it uses concatenation to build WHERE clause
     and because SYSDATE depends on the value of NLS_DATE_FORMAT. */

  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || ''' AND date_created>'''
           || (SYSDATE - 30)
           || '''';
```

```
  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Andy', 'Waiter', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter' AND date_created>'29-MAR-2010'
Rec: Serve dinner at Cafe Pete
```

Example of statement modification:

```
ALTER SESSION SET NLS_DATE_FORMAT='"'' OR service_type=''Merger"';

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Anybody', 'Anything', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created>'' OR service_type='Merger'
Rec: Buy company XYZ

PL/SQL procedure successfully completed.
```

Guarding Against SQL Injection

If you use dynamic SQL in your PL/SQL applications, you must check the input text to ensure that it is exactly what you expected. You can use the following techniques:

- Bind Variables
- Validation Checks
- Explicit Format Models

Bind Variables

The most effective way to make your PL/SQL code invulnerable to SQL injection attacks is to use bind variables. The database uses the values of bind variables exclusively and does not interpret their contents in any way. (Bind variables also improve performance.)

The procedure in Example 7–12 is invulnerable to SQL injection because it builds the dynamic SQL statement with bind variables (not by concatenation as in the vulnerable procedure in Example 7–9). The same binding technique fixes the vulnerable procedure shown in Example 7–10.

Example 7–12 Bind Variables Guarding Against SQL Injection

Create invulnerable procedure:

```
CREATE OR REPLACE PROCEDURE get_record_2 (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  query := 'SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b';

  DBMS_OUTPUT.PUT_LINE('Query: ' || query);

  EXECUTE IMMEDIATE query INTO rec USING user_name, service_type;

  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record_2('Andy', 'Waiter', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b
Rec: Serve dinner at Cafe Pete

PL/SQL procedure successfully completed.
```

Attempt statement modification:

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record_2('Anybody '' OR service_type=''Merger''--',
               'Anything',
               record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records
```

```
            WHERE user_name=:a
            AND service_type=:b
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "HR.GET_RECORD_2", line 14
ORA-06512: at line 4
```

Validation Checks

Always have your program validate user input to ensure that it is what is intended. For example, if the user is passing a department number for a `DELETE` statement, check the validity of this department number by selecting from the `departments` table. Similarly, if a user enters the name of a table to be deleted, check that this table exists by selecting from the static data dictionary view `ALL_TABLES`.

> **Caution:** When checking the validity of a user name and its password, always return the same error regardless of which item is invalid. Otherwise, a malicious user who receives the error message "invalid password" but not "invalid user name" (or the reverse) can realize that he or she has guessed one of these correctly.

In validation-checking code, the subprograms in the `DBMS_ASSERT` package are often useful. For example, you can use the `DBMS_ASSERT.ENQUOTE_LITERAL` function to enclose a string literal in quotation marks, as Example 7–13 does. This prevents a malicious user from injecting text between an opening quotation mark and its corresponding closing quotation mark.

> **Caution:** Although the `DBMS_ASSERT` subprograms are useful in validation code, they do not replace it. For example, an input string can be a qualified SQL name (verified by `DBMS_ASSERT.QUALIFIED_SQL_NAME`) and still be a fraudulent password.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_ASSERT` subprograms

In Example 7–13, the procedure `raise_emp_salary` checks the validity of the column name that was passed to it before it updates the `employees` table, and then the anonymous block invokes the procedure from both a dynamic PL/SQL block and a dynamic SQL statement.

Example 7–13 Validation Checks Guarding Against SQL Injection

```
CREATE OR REPLACE PROCEDURE raise_emp_salary (
  column_value  NUMBER,
  emp_column    VARCHAR2,
  amount NUMBER )
IS
  v_column  VARCHAR2(30);
  sql_stmt  VARCHAR2(200);
BEGIN
  -- Check validity of column name that was given as input:
  SELECT column_name INTO v_column
  FROM USER_TAB_COLS
```

```
  WHERE TABLE_NAME = 'EMPLOYEES'
  AND COLUMN_NAME = emp_column;

  sql_stmt := 'UPDATE employees SET salary = salary + :1 WHERE '
    || DBMS_ASSERT.ENQUOTE_NAME(v_column,FALSE) || ' = :2';

  EXECUTE IMMEDIATE sql_stmt USING amount, column_value;

  -- If column name is valid:
  IF SQL%ROWCOUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Salaries were updated for: '
      || emp_column || ' = ' || column_value);
  END IF;

  -- If column name is not valid:
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Invalid Column: ' || emp_column);
END raise_emp_salary;
/

DECLARE
  plsql_block  VARCHAR2(500);
BEGIN
  -- Invoke raise_emp_salary from a dynamic PL/SQL block:
  plsql_block :=
    'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;';

  EXECUTE IMMEDIATE plsql_block
    USING 110, 'DEPARTMENT_ID', 10;

  -- Invoke raise_emp_salary from a dynamic SQL statement:
  EXECUTE IMMEDIATE 'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;'
    USING 112, 'EMPLOYEE_ID', 10;
END;
/
```

Result:

```
Salaries were updated for: DEPARTMENT_ID = 110
Salaries were updated for: EMPLOYEE_ID = 112
```

Explicit Format Models

If you use datetime and numeric values that are concatenated into the text of a SQL or PL/SQL statement, and you cannot pass them as bind variables, convert them to text using explicit format models that are independent from the values of the NLS parameters of the running session. Ensure that the converted values have the format of SQL datetime or numeric literals. Using explicit locale-independent format models to construct SQL is recommended not only from a security perspective, but also to ensure that the dynamic SQL statement runs correctly in any globalization environment.

The procedure in Example 7–14 is invulnerable to SQL injection because it converts the datetime parameter value, `SYSDATE - 30`, to a `VARCHAR2` value explicitly, using the `TO_CHAR` function and a locale-independent format model (not implicitly, as in the vulnerable procedure in Example 7–11).

Example 7–14 Explicit Format Models Guarding Against SQL Injection

Create invulnerable procedure:

```
-- Return records not older than a month

CREATE OR REPLACE PROCEDURE get_recent_record (
  user_name     IN  VARCHAR2,
  service_type  IN  VARCHAR2,
  rec           OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  /* Following SELECT statement is vulnerable to modification
     because it uses concatenation to build WHERE clause. */

  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || ''' AND date_created> DATE '''
           || TO_CHAR(SYSDATE - 30,'YYYY-MM-DD')
           || '''';

  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

Attempt statement modification:

```
ALTER SESSION SET NLS_DATE_FORMAT='"'' OR service_type=''Merger"';

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Anybody', 'Anything', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created> DATE '2010-03-29'
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "SYS.GET_RECENT_RECORD", line 21
ORA-06512: at line 4
```